

1 Front Matter

Copyright © 1992, 1993, 1994, 1995 Intermetrics, Inc.

Copyright © 2000 The MITRE Corporation, Inc.

Copyright © 2004, 2005, 2006 AXE Consultants

Copyright © 2004, 2005, 2006 Ada–Europe

Ada Reference Manual – Language and Standard Libraries

Copyright © 1992, 1993, 1994, 1995, Intermetrics, Inc.

This copyright is assigned to the U.S. Government. All rights reserved.

This document may be copied, in whole or in part, in any form or by any means, as is or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Compiled copies of standard library units and examples need not contain this copyright notice so long as the notice is included in all copies of source code and documentation.

Technical Corrigendum 1

Copyright © 2000, The MITRE Corporation. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of MITRE.

You use this document on the condition that you indemnify and hold harmless MITRE, its Board of Trustees, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

MITRE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL MITRE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF MITRE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Amendment 1

Copyright © 2004, 2005, 2006, AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this

copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Consolidated Standard

Copyright © 2004, 2005, 2006, Ada–Europe.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of Ada–Europe.

You use this document on the condition that you indemnify and hold harmless Ada–Europe and its Board from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

ADA–EUROPE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL ADA–EUROPE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF ADA–EUROPE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

1.1 0.1 Foreword to this version of the Ada Reference Manual

0.1/1

The International Standard for the programming language Ada is ISO/IEC 8652:1995(E).

0.2/1

The Ada Working Group ISO/IEC JTC 1/SC 22/WG 9 is tasked by ISO with the work item to interpret and maintain the International Standard and to produce Technical Corrigenda, as appropriate. The technical work on the International Standard is performed by the Ada Rapporteur Group (ARG) of WG 9. In September 2000, WG 9 approved and forwarded Technical Corrigendum 1 to SC 22 for ISO approval, which was granted in February 2001. Technical Corrigendum 1 was published in June 2001.

0.3/2

In October 2002, WG 9 approved a schedule and guidelines for the preparation of an Amendment to the International Standard. WG 9 approved the scope of the Amendment in June 2004. In April 2006, WG 9 approved and forwarded the Amendment to SC 22 for approval, which was granted in August 2006. Final ISO/IEC approval is expected by early 2007.

0.4/1

The Technical Corrigendum lists the individual changes that need to be made to the text of the International Standard to correct errors, omissions or inconsistencies. The corrections specified in Technical Corrigendum 1 are part of the International Standard ISO/IEC 8652:1995(E).

0.5/2

Similarly, Amendment 1 lists the individual changes that need to be made to the text of the International Standard to add new features as well as correct errors.

0.6/2

When ISO published Technical Corrigendum 1, it did not also publish a document that merges the changes from the Technical Corrigendum into the text of the International Standard. It is not known whether ISO will publish a document that merges the changes from Technical Corrigendum and Amendment 1 into the text of the International Standard. However, ISO rules require that the project editor for the International Standard be able to produce such a document on demand.

0.7/2

This version of the Ada Reference Manual is what the project editor would provide to ISO in response to such a request. It incorporates the changes specified in the Technical Corrigendum and Amendment into the text of ISO/IEC 8652:1995(E). It should be understood that the publication of any ISO document involves changes in general format, boilerplate, headers, etc., as well as a review by professional editors that may introduce editorial changes to the text. This version of the Ada Reference Manual is therefore neither an official ISO document, nor a version guaranteed to be identical to an official ISO document, should ISO decide to reprint the International Standard incorporating an approved Technical Corrigendum and Amendment. It is nevertheless a best effort to be as close as possible to the technical content of such an updated document. In the case of a conflict between this document and Amendment 1 as approved by ISO (or between this document and Technical Corrigendum 1 in the case of paragraphs not changed by Amendment 1; or between this document and the original 8652:1995 in the case of paragraphs not changed by either Amendment 1 or Technical Corrigendum 1), the other documents contain the official text of the International Standard ISO/IEC 8652:1995(E) and its Amendment.

0.8/2

As it is very inconvenient to have the Reference Manual for Ada specified in three documents, this consolidated version of the Ada Reference Manual is made available to the public.

1.2 0.2 Foreword

1

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of Interna-

tional Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

2

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

3

International Standard ISO/IEC 8652 was prepared by Joint Technical Committee ISO/IEC JTC 1, <Information Technology>.

4/2

This consolidated edition updates the second edition (ISO 8652:1995).

5/2

Annexes A to J form an integral part of this International Standard. Annexes K to Q are for information only.

1.3 0.3 Introduction

1

This is the Ada Reference Manual.

2

Other available Ada documents include:

3/2

- Ada 95 Rationale. This gives an introduction to the new features of Ada incorporated in the 1995 edition of this Standard, and explains the rationale behind them. Programmers unfamiliar with Ada 95 should read this first.

3.1/2

- Ada 2005 Rationale. This gives an introduction to the changes and new features in Ada 2005 (compared with the 1995 edition), and explains the rationale behind them. Programmers should read this rationale before reading this Standard in depth.

4/1

- <This paragraph was deleted.>

5/2

- The Annotated Ada Reference Manual (AARM). The AARM contains all of the text in the consolidated Ada Reference Manual, plus various annotations. It is intended primarily for compiler writers, validation test writers, and others who wish to study the fine details. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

Design Goals

6/2

Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. The 1995 revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on reliability, maintainability, and efficiency. This amended version provides further flexibility and adds more standardized packages within the framework provided by the 1995 revision.

7

The need for languages that promote reliability and simplify maintenance is well established. Hence emphasis was placed on program readability over ease of writing. For example, the rules of the language require that program variables be explicitly declared and that their type be specified. Since the type of a variable is invariant, compilers can ensure that operations on variables are compatible with the properties intended for objects of the type. Furthermore, error-prone notations have been avoided, and the syntax of the language avoids the use of encoded forms in favor of more English-like constructs. Finally, the language offers support for separate compilation of program units in a way that facilitates program development and maintenance, and which provides the same degree of checking between units as within a unit.

8

Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep to a relatively small number of underlying concepts integrated in a consistent and systematic way while continuing to avoid the pitfalls of excessive involution. The design especially aims to provide language constructs that correspond intuitively to the normal expectations of users.

9

Like many other human activities, the development of programs is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components continues to be a central idea in the design. The concepts of packages, of private types, and of generic units are directly related to this idea, which has ramifications in many other aspects of the language. An allied concern is the maintenance of programs to match changing requirements; type extension and the hierarchical library enable a program to be modified while minimizing disturbance to existing tested and trusted components.

10

No language can avoid the problem of efficiency. Languages that require over-elaborate compilers, or that lead to the inefficient use of storage or execution time, force these inefficiencies on all machines and on all programs. Every construct of the language was examined in the light of present implementation techniques. Any proposed construct whose implementation was unclear or that required excessive machine resources was rejected.

Language Summary

11

An Ada program is composed of one or more program units. Program units may be subprograms (which define executable algorithms), packages (which define collections of entities),

task units (which define concurrent computations), protected units (which define operations for the coordinated sharing of data between tasks), or generic units (which define parameterized forms of packages and subprograms). Each program unit normally consists of two parts: a specification, containing the information that must be visible to other units, and a body, containing the implementation details, which need not be visible to other units. Most program units can be compiled separately.

12

This distinction of the specification and body, and the ability to compile units separately, allows a program to be designed, written, and tested as a set of largely independent software components.

13

An Ada program will normally make use of a library of program units of general utility. The language provides means whereby individual organizations can construct their own libraries. All libraries are structured in a hierarchical manner; this enables the logical decomposition of a subsystem into individual components. The text of a separately compiled program unit must name the library units it requires.

14

<Program Units>

15

A subprogram is the basic unit for expressing an algorithm. There are two kinds of subprograms: procedures and functions. A procedure is the means of invoking a series of actions. For example, it may read data, update variables, or produce some output. It may have parameters, to provide a controlled means of passing information between the procedure and the point of call. A function is the means of invoking the computation of a value. It is similar to a procedure, but in addition will return a result.

16

A package is the basic unit for defining a collection of logically related entities. For example, a package can be used to define a set of type declarations and associated operations. Portions of a package can be hidden from the user, thus allowing access only to the logical properties expressed by the package specification.

17

Subprogram and package units may be compiled separately and arranged in hierarchies of parent and child units giving fine control over visibility of the logical properties and their detailed implementation.

18

A task unit is the basic unit for defining a task whose sequence of actions may be executed concurrently with those of other tasks. Such tasks may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. A task unit may define either a single executing task or a task type permitting the creation of any number of similar tasks.

19/2

A protected unit is the basic unit for defining protected operations for the coordinated use of data shared between tasks. Simple mutual exclusion is provided automatically, and more elaborate sharing protocols can be defined. A protected operation can either be a subprogram or an entry. A protected entry specifies a Boolean expression (an entry barrier)

that must be True before the body of the entry is executed. A protected unit may define a single protected object or a protected type permitting the creation of several similar objects.

20

<Declarations and Statements>

21

The body of a program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a sequence of statements, which defines the execution of the program unit.

22

The declarative part associates names with declared entities. For example, a name may denote a type, a constant, a variable, or an exception. A declarative part also introduces the names and parameters of other nested subprograms, packages, task units, protected units, and generic units to be used in the program unit.

23

The sequence of statements describes a sequence of actions that are to be performed. The statements are executed in succession (unless a transfer of control causes execution to continue from another place).

24

An assignment statement changes the value of a variable. A procedure call invokes execution of a procedure after associating any actual parameters provided at the call with the corresponding formal parameters.

25

Case statements and if statements allow the selection of an enclosed sequence of statements based on the value of an expression or on the value of a condition.

26

The loop statement provides the basic iterative mechanism in the language. A loop statement specifies that a sequence of statements is to be executed repeatedly as directed by an iteration scheme, or until an exit statement is encountered.

27

A block statement comprises a sequence of statements preceded by the declaration of local entities used by the statements.

28

Certain statements are associated with concurrent execution. A delay statement delays the execution of a task for a specified duration or until a specified time. An entry call statement is written as a procedure call statement; it requests an operation on a task or on a protected object, blocking the caller until the operation can be performed. A called task may accept an entry call by executing a corresponding accept statement, which specifies the actions then to be performed as part of the rendezvous with the calling task. An entry call on a protected object is processed when the corresponding entry barrier evaluates to true, whereupon the body of the entry is executed. The requeue statement permits the provision of a service as a number of related activities with preference control. One form of the select statement allows a selective wait for one of several alternative rendezvous. Other forms of the select statement allow conditional or timed entry calls and the asynchronous transfer of control in response to some triggering event.

29

Execution of a program unit may encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation may exceed the maximum allowed value of a number, or an attempt may be made to access an array component by using an incorrect index value. To deal with such error situations, the statements of a program unit can be textually followed by exception handlers that specify the actions to be taken when the error situation arises. Exceptions can be raised explicitly by a raise statement.

30

<Data Types>

31

Every object in the language has a type, which characterizes a set of values and a set of applicable operations. The main classes of types are elementary types (comprising enumeration, numeric, and access types) and composite types (including array and record types).

32/2

An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types Boolean, Character, Wide_Character, and Wide_Wide_Character are predefined.

33

Numeric types provide a means of performing exact or approximate numerical computations. Exact computations use integer types, which denote sets of consecutive integers. Approximate computations use either fixed point types, with absolute bounds on the error, or floating point types, with relative bounds on the error. The numeric types Integer, Float, and Duration are predefined.

34/2

Composite types allow definitions of structured objects with related components. The composite types in the language include arrays and records. An array is an object with indexed components of the same type. A record is an object with named components of possibly different types. Task and protected types are also forms of composite types. The array types String, Wide_String, and Wide_Wide_String are predefined.

35

Record, task, and protected types may have special components called discriminants which parameterize the type. Variant record structures that depend on the values of discriminants can be defined within a record type.

36

Access types allow the construction of linked data structures. A value of an access type represents a reference to an object declared as aliased or to an object created by the evaluation of an allocator. Several variables of an access type may designate the same object, and components of one object may designate the same or other objects. Both the elements in such linked data structures and their relation to other elements can be altered during program execution. Access types also permit references to subprograms to be stored, passed as parameters, and ultimately dereferenced as part of an indirect call.

37

Private types permit restricted views of a type. A private type can be defined in a package so that only the logically necessary properties are made visible to the users of the type.

The full structural details that are externally irrelevant are then only available within the package and any child units.

38

From any type a new type may be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations may be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives may be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension must be tagged, so that the specific type of an operand within a derivation class can be identified at run time. When an operation of a tagged type is applied to an operand whose specific type is not known until run time, implicit dispatching is performed based on the tag of the operand.

38.1/2

Interface types provide abstract models from which other interfaces and types may be composed and derived. This provides a reliable form of multiple inheritance. Interface types may also be implemented by task types and protected types thereby enabling concurrent programming and inheritance to be merged.

39

The concept of a type is further refined by the concept of a subtype, whereby a user can constrain the set of allowed values of a type. Subtypes can be used to define subranges of scalar types, arrays with a limited set of index values, and records and private types with particular discriminant values.

40

<Other Facilities>

41/2

Aspect clauses can be used to specify the mapping between types and features of an underlying machine. For example, the user can specify that objects of a given type must be represented with a given number of bits, or that the components of a record are to be represented using a given storage layout. Other features allow the controlled use of low level, nonportable, or implementation-dependent aspects, including the direct insertion of machine code.

42/2

The predefined environment of the language provides for input-output and other capabilities by means of standard library packages. Input-output is supported for values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided.

42.1/2

The predefined standard library packages provide facilities such as string manipulation, containers of various kinds (vectors, lists, maps, etc.), mathematical functions, random number generation, and access to the execution environment.

42.2/2

The specialized annexes define further predefined library packages and facilities with emphasis on areas such as real-time scheduling, interrupt handling, distributed systems, numerical computation, and high-integrity systems.

43

Finally, the language provides a powerful means of parameterization of program units, called generic program units. The generic parameters can be types and subprograms (as well as objects and packages) and so allow general algorithms and data structures to be defined that are applicable to all types of a given class.

Language Changes

44/2

This amended International Standard updates the edition of 1995 which replaced the first edition of 1987. In the 1995 edition, the following major language changes were incorporated:

45/2

- Support for standard 8-bit and 16-bit characters was added. See clauses Section 3.1 [2.1], page 32, Section 4.5.2 [3.5.2], page 93, Section 4.6.3 [3.6.3], page 122, Section 15.1 [A.1], page 556, Section 15.3 [A.3], page 565, and Section 15.4 [A.4], page 584.

46/2

- The type model was extended to include facilities for object-oriented programming with dynamic polymorphism. See the discussions of classes, derived types, tagged types, record extensions, and private extensions in clauses Section 4.4 [3.4], page 66, Section 4.9 [3.9], page 136, and Section 8.3 [7.3], page 283. Additional forms of generic formal parameters were allowed as described in clauses Section 13.5.1 [12.5.1], page 462, and Section 13.7 [12.7], page 474.

47/2

- Access types were extended to allow an access value to designate a subprogram or an object declared by an object declaration as opposed to just an object allocated on a heap. See clause Section 4.10 [3.10], page 156.

48/2

- Efficient data-oriented synchronization was provided by the introduction of protected types. See clause Section 10.4 [9.4], page 337.

49/2

- The library structure was extended to allow library units to be organized into a hierarchy of parent and child units. See clause Section 11.1 [10.1], page 394.

50/2

- Additional support was added for interfacing to other languages. See Chapter 16 [Annex B], page 894.

51/2

- The Specialized Needs Annexes were added to provide specific support for certain application areas:

52

- Chapter 17 [Annex C], page 949,
"Chapter 17 [Annex C], page 949,
Systems Programming"

53

- Chapter 18 [Annex D], page 974,
"Chapter 18 [Annex D], page 974,
Real-Time Systems"

54

- Chapter 19 [Annex E], page 1034,
"Chapter 19 [Annex E], page 1034,
Distributed Systems"

55

- Chapter 20 [Annex F], page 1054,
"Chapter 20 [Annex F], page 1054,
Information Systems"

56

- Chapter 21 [Annex G], page 1083,
"Chapter 21 [Annex G], page 1083,
Numerics"

57

- Chapter 22 [Annex H], page 1153,
"Chapter 22 [Annex H], page 1153,
High Integrity Systems"

57.1/2

Amendment 1 modifies the 1995 International Standard by making changes and additions that improve the capability of the language and the reliability of programs written in the language. In particular the changes were designed to improve the portability of programs, interfacing to other languages, and both the object-oriented and real-time capabilities.

57.2/2

The following significant changes with respect to the 1995 edition are incorporated:

57.3/2

- Support for program text is extended to cover the entire ISO/IEC 10646:2003 repertoire. Execution support now includes the 32-bit character set. See clauses Section 3.1 [2.1], page 32, Section 4.5.2 [3.5.2], page 93, Section 4.6.3 [3.6.3], page 122, Section 15.1 [A.1], page 556, Section 15.3 [A.3], page 565, and Section 15.4 [A.4], page 584.

57.4/2

- The object-oriented model has been improved by the addition of an interface facility which provides multiple inheritance and additional flexibility for type extensions. See clauses Section 4.4 [3.4], page 66, Section 4.9 [3.9], page 136, and Section 8.3 [7.3], page 283. An alternative notation for calling operations more akin to that used in other languages has also been added. See clause Section 5.1.3 [4.1.3], page 183.

57.5/2

- Access types have been further extended to unify properties such as the ability to access constants and to exclude null values. See clause Section 4.10 [3.10], page 156. Anonymous access types are now permitted more freely and anonymous access-to-subprogram types are introduced. See clauses Section 4.3 [3.3], page 58, Section 4.6 [3.6], page 114, Section 4.10 [3.10], page 156, and Section 9.5.1 [8.5.1], page 317.

57.6/2

- The control of structure and visibility has been enhanced to permit mutually dependent references between units and finer control over access from the private part of a package. See clauses Section 4.10.1 [3.10.1], page 160, and Section 11.1.2 [10.1.2], page 399. In addition, limited types have been made more useful by the provision of aggregates, constants, and constructor functions. See clauses Section 5.3 [4.3], page 190, Section 7.5 [6.5], page 272, and Section 8.5 [7.5], page 292.

57.7/2

- The predefined environment has been extended to include additional time and calendar operations, improved string handling, a comprehensive container library, file and directory management, and access to environment variables. See clauses Section 10.6.1 [9.6.1], page 363, Section 15.4 [A.4], page 584, Section 15.16 [A.16], page 757, Section 15.17 [A.17], page 775, and Section 15.18 [A.18], page 778.

57.8/2

- Two of the Specialized Needs Annexes have been considerably enhanced:

57.9/2

- The Real-Time Systems Annex now includes the Ravenscar profile

for high-integrity systems, further dispatching policies such as Round Robin and Earliest Deadline First, support for timing events, and support for control of CPU time utilization. See clauses Section 18.2 [D.2], page 978, Section 18.13 [D.13], page 1020, Section 18.14 [D.14], page 1021, and Section 18.15 [D.15], page 1031.

57.10/2

- The Numerics Annex now includes support for real and complex vectors and matrices as previously defined in ISO/IEC 13813:1997 plus further basic operations for linear algebra. See clause Section 21.3 [G.3], page 1119.

57.11/2

- The overall reliability of the language has been enhanced by a number of improvements. These include new syntax which detects accidental overloading, as well as pragmas for making assertions and giving better control over the suppression of checks. See clauses Section 7.1 [6.1], page 255, Section 12.4.2 [11.4.2], page 427, and Section 12.5 [11.5], page 431.

Instructions for Comment Submission

58/1

Informal comments on this International Standard may be sent via e-mail to ada-comment@ada-auth.org. If appropriate, the Project Editor will initiate the defect correction procedure.

59

Comments should use the following format:

60/2

```
!topic <Title summarizing comment>
!reference Ada 2005 RM<ss.ss(pp)>
!from <Author Name yy-mm-dd>
!keywords <keywords related to topic>
!discussion
```

```
<text of discussion>
```

61

where <ss.ss> is the section, clause or subclause number, <pp> is the paragraph number where applicable, and <yy-mm-dd> is the date the comment was sent. The date is optional, as is the !keywords line.

62/1

Please use a descriptive "Subject" in your e-mail message, and limit each message to a single comment.

63

When correcting typographical errors or making minor wording suggestions, please put the correction directly as the topic of the comment; use square brackets [] to indicate text to be omitted and curly braces { } to indicate text to be added, and provide enough context to make the nature of the suggestion self-evident or put additional information in the body of the comment, for example:

64

```
!topic [c]{C}haracter
!topic it[']s meaning is not defined
```

65

Formal requests for interpretations and for reporting defects in this International Standard may be made in accordance with the ISO/IEC JTC 1 Directives and the ISO/IEC JTC 1/SC 22 policy for interpretations. National Bodies may submit a Defect Report to ISO/IEC JTC 1/SC 22 for resolution under the JTC 1 procedures. A response will be provided and, if appropriate, a Technical Corrigendum will be issued in accordance with the procedures.

Acknowledgements for the Ada 95 edition of the Ada Reference Manual

66

This International Standard was prepared by the Ada 9X Mapping/Revision Team based at Intermetrics, Inc., which has included: W. Carlson, Program Manager; T. Taft, Technical Director; J. Barnes (consultant); B. Brosgol (consultant); R. Duff (Oak Tree Software); M. Edwards; C. Garrity; R. Hilliard; O. Pazy (consultant); D. Rosenfeld; L. Shafer; W. White; M. Woodger.

67

The following consultants to the Ada 9X Project contributed to the Specialized Needs Annexes: T. Baker (Real-Time/Systems Programming -- SEI, FSU); K. Dritz (Numerics -- Argonne National Laboratory); A. Gargaro (Distributed Systems -- Computer Sciences); J. Goodenough (Real-Time/Systems Programming -- SEI); J. McHugh (Secure Systems -- consultant); B. Wichmann (Safety-Critical Systems -- NPL: UK).

68

This work was regularly reviewed by the Ada 9X Distinguished Reviewers and the members of the Ada 9X Rapporteur Group (XRG): E. Ploedereder, Chairman of DRs and XRG (University of Stuttgart: Germany); B. Bardin (Hughes); J. Barnes (consultant: UK); B. Brett (DEC); B. Brosgol (consultant); R. Brukardt (RR Software); N. Cohen (IBM); R. Dewar (NYU); G. Dismukes (TeleSoft); A. Evans (consultant); A. Gargaro (Computer Sciences); M. Gerhardt (ESL); J. Goodenough (SEI); S. Heilbrunner (University of Salzburg: Austria); P. Hilfinger (UC/Berkeley); B. Källberg (CelsiusTech: Sweden); M. Kamrad II (Unisys); J. van Katwijk (Delft University of Technology: The Netherlands); V. Kaufman (Russia); P. Kruchten (Rational); R. Landwehr (CCI: Germany); C. Lester (Portsmouth Polytechnic: UK); L. Månsson (TELIA Research: Sweden); S. Michell (Multiprocessor Toolsmiths: Canada); M. Mills (US Air Force); D. Pogge (US Navy); K. Power (Boeing); O. Roubine (Verdix: France); A. Strohmeier (Swiss Fed Inst of Technology: Switzerland); W.

Taylor (consultant: UK); J. Tokar (Tartan); E. Vasilescu (Grumman); J. Vladik (Prospeks s.r.o.: Czech Republic); S. Van Vlierberghe (OFFIS: Belgium).

69

Other valuable feedback influencing the revision process was provided by the Ada 9X Language Precision Team (Odyssey Research Associates), the Ada 9X User/Implementer Teams (AETECH, Tartan, TeleSoft), the Ada 9X Implementation Analysis Team (New York University) and the Ada community—at-large.

70

Special thanks go to R. Mathis, Convenor of ISO/IEC JTC 1/SC 22 Working Group 9.

71

The Ada 9X Project was sponsored by the Ada Joint Program Office. Christine M. Anderson at the Air Force Phillips Laboratory (Kirtland AFB, NM) was the project manager.

Acknowledgements for the Corrigendum version of the Ada Reference Manual

71.1/1

The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this revision possible.

71.2/1

Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording corrections was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, E. Ploedereder (Germany), who kept the process moving; J. Barnes (UK) and K. Ishihata (Japan), whose extremely detailed reviews kept the editor on his toes; G. Dismukes (USA), M. Kamrad (USA), P. Leroy (France), S. Michell (Canada), T. Taft (USA), J. Tokar (USA), and other members too numerous to mention.

71.3/1

Special thanks go to R. Duff (USA) for his explanations of the previous system of formatting of these documents during the tedious conversion to more modern formats. Special thanks also go to the convener of ISO/IEC JTC 1/SC 22/WG 9, J. Moore (USA), without whose help and support the corrigendum and this consolidated reference manual would not have been possible.

Acknowledgements for the Amendment version of the Ada Reference Manual

71.4/2

The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this revision possible.

71.5/2

Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording corrections was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, P. Leroy (France), who kept the process on schedule; J. Barnes (UK) whose careful reviews found many typographical errors; T. Taft (USA), who always seemed to have a suggestion when we were stuck, and who also was usually able to provide the valuable service of explaining why things were as they are; S. Baird (USA), who found many obscure problems with the proposals; and A. Burns (UK), who pushed many of the real-time proposals to completion. Other ARG members who contributed were: R. Dewar (USA), G. Dismukes (USA), R. Duff

(USA), K. Ishihata (Japan), S. Michell (Canada), E. Ploedereder (Germany), J.P. Rosen (France), E. Schonberg (USA), J. Tokar (USA), and T. Vardanega (Italy).

71.6/2

Special thanks go to Ada–Europe and the Ada Resource Association, without whose help and support the Amendment and this consolidated reference manual would not have been possible. M. Heaney (USA) requires special thanks for his tireless work on the containers packages. Finally, special thanks go to the convener of ISO/IEC JTC 1/SC 22/WG 9, J. Moore (USA), who guided the document through the standardization process.

Changes

72

The International Standard is the same as this version of the Reference Manual, except:

73

- This list of Changes is not included in the International Standard.

74

- The "Acknowledgements" page is not included in the International Standard.

75

- The text in the running headers and footers on each page is slightly different in the International Standard.

76

- The title page(s) are different in the International Standard.

77

- This document is formatted for 8.5–by–11–inch paper, whereas the International Standard is formatted for A4 paper (210–by–297mm); thus, the page breaks are in different places.

77.1/1

- The "Foreword to this version of the Ada Reference Manual" clause is not included in the International Standard.

77.2/2

- The "Using this version of the Ada Reference Manual" clause is not included in the International Standard.

Using this version of the Ada Reference Manual

77.3/2

This document has been revised with the corrections specified in Technical Corrigendum

1 (ISO/IEC 8652:1995/COR.1:2001) and Amendment 1 (ISO/IEC 8652/AMD.1:2007). In addition, a variety of editorial errors have been corrected.

77.4/2

Changes to the original 8652:1995 can be identified by the version number following the paragraph number. Paragraphs with a version number of /1 were changed by Technical Corrigendum 1 or were editorial corrections at that time, while paragraphs with a version number of /2 were changed by Amendment 1 or were more recent editorial corrections. Paragraphs not so marked are unchanged by Amendment 1, Technical Corrigendum 1, or editorial corrections. Paragraph numbers of unchanged paragraphs are the same as in the original Ada Reference Manual. In addition, some versions of this document include revision bars near the paragraph numbers. Where paragraphs are inserted, the paragraph numbers are of the form pp.nn, where pp is the number of the preceding paragraph, and nn is an insertion number. For instance, the first paragraph inserted after paragraph 8 is numbered 8.1, the second paragraph inserted is numbered 8.2, and so on. Deleted paragraphs are indicated by the text <This paragraph was deleted.> Deleted paragraphs include empty paragraphs that were numbered in the original Ada Reference Manual.

1.4 0.99

===== INTERNATIONAL STANDARD ISO/IEC 8652:2007(E), Ed. 3

=====

Information technology -- Programming

Languages -- Ada

2 1 General

1

Ada is a programming language designed to support the construction of long-lived, highly reliable software systems. The language includes facilities to define packages of related types, objects, and operations. The packages may be parameterized and the types may be extended to support the construction of libraries of reusable, adaptable software components. The operations may be implemented as subprograms using conventional sequential control structures, or as entries that include synchronization of concurrent threads of control as part of their invocation. The language treats modularity in the physical sense as well, with a facility to support separate compilation.

2

The language includes a complete facility for the support of real-time, concurrent programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, and random number generation.

2.1 1.1 Scope

1

This International Standard specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs to a variety of data processing systems.

2.1.1 1.1.1 Extent

1

This International Standard specifies:

2

- The form of a program written in Ada;

3

- The effect of translating and executing such a program;

4

- The manner in which program units may be combined to form Ada programs;

5

- The language-defined library units that a conforming implementation is required to supply;

6

- The permissible variations within the standard, and the manner in which they are to be documented;

7

- Those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program containing such violations;

8

- Those violations of the standard that a conforming implementation is not required to detect.

9

This International Standard does not specify:

10

- The means whereby a program written in Ada is transformed into object code executable by a processor;

11

- The means whereby translation or execution of programs is invoked and the executing units are controlled;

12

- The size or speed of the object code, or the relative execution speed of different language constructs;

13

- The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages;

14

- The effect of unspecified execution.

15

- The size of a program or program unit that will exceed the capacity of a particular conforming implementation.

2.1.2 1.1.2 Structure

1

This International Standard contains thirteen sections, fourteen annexes, and an index.

2

The <core> of the Ada language consists of:

3

- Sections 1 through 13

4

- Chapter 15 [Annex A], page 553, "Chapter 15 [Annex A], page 553, Predefined Language Environment"

5

- Chapter 16 [Annex B], page 894, "Chapter 16 [Annex B], page 894, Interface to Other Languages"

6

- Chapter 23 [Annex J], page 1166, "Chapter 23 [Annex J], page 1166, Obsolescent Features"

7

The following <Specialized Needs Annexes> define features that are needed by certain application areas:

8

- Chapter 17 [Annex C], page 949, "Chapter 17 [Annex C], page 949, Systems Programming"

9

- Chapter 18 [Annex D], page 974, "Chapter 18 [Annex D], page 974, Real-Time Systems"

10

- Chapter 19 [Annex E], page 1034, "Chapter 19 [Annex E], page 1034, Distributed Systems"

11

- Chapter 20 [Annex F], page 1054, "Chapter 20 [Annex F], page 1054, Information Systems"

12

- Chapter 21 [Annex G], page 1083, "Chapter 21 [Annex G], page 1083, Numerics"

13

- Chapter 22 [Annex H], page 1153, "Chapter 22 [Annex H], page 1153, High Integrity Systems"

14

The core language and the Specialized Needs Annexes are normative, except that the material in each of the items listed below is informative:

15

- Text under a NOTES or Examples heading.

16

- Each clause or subclause whose title starts with the word "Example" or "Examples".

17

All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Specialized Needs Annexes.

18

The following Annexes are informative:

19

- Chapter 24 [Annex K], page 1179, "Chapter 24 [Annex K], page 1179, Language-Defined Attributes"

20

- Chapter 25 [Annex L], page 1242, "Chapter 25 [Annex L], page 1242, Language-Defined Pragmas"

21

- Section 26.2 [M.2], page 1250, "Section 26.2 [M.2], page 1250, Implementation-Defined Characteristics"

22

- Chapter 27 [Annex N], page 1283, "Chapter 27 [Annex N], page 1283, Glossary"

23

- Chapter 28 [Annex P], page 1289, "Chapter 28 [Annex P], page 1289, Syntax Summary"

24

Each section is divided into clauses and subclauses that have a common structure. Each section, clause, and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

Syntax

25

Syntax rules (indented).

Name Resolution Rules

26

Compile–time rules that are used in name resolution, including overload resolution.

Legality Rules

27

Rules that are enforced at compile time. A construct is <legal> if it obeys all of the Legality Rules.

Static Semantics

28

A definition of the compile–time effect of each construct.

Post-Compilation Rules

29

Rules that are enforced before running a partition. A partition is legal if its compilation units are legal and it obeys all of the Post–Compilation Rules.

Dynamic Semantics

30

A definition of the run–time effect of each construct.

Bounded (Run-Time) Errors

31

Situations that result in bounded (run–time) errors (see Section 2.1.5 [1.1.5], page 28).

Erroneous Execution

32

Situations that result in erroneous execution (see Section 2.1.5 [1.1.5], page 28).

Implementation Requirements

33

Additional requirements for conforming implementations.

Documentation Requirements

34

Documentation requirements for conforming implementations.

Metrics

35

Metrics that are specified for the time/space properties of the execution of certain language constructs.

Implementation Permissions

36

Additional permissions given to the implementer.

Implementation Advice

37

Optional advice given to the implementer. The word "should" is used to indicate that the advice is a recommendation, not a requirement. It is implementation defined whether or not a given recommendation is obeyed.

NOTES

38

1 Notes emphasize consequences of the rules described in the (sub)clause or elsewhere. This material is informative.

Examples

39

Examples illustrate the possible forms of the constructs described. This material is informative.

2.1.3 1.1.3 Conformity of an Implementation with the Standard

Implementation Requirements

1

A conforming implementation shall:

2

- Translate and correctly execute legal programs written in Ada, provided that they are not so large as to exceed the capacity of the implementation;

3

- Identify all programs or program units that are so large as to exceed the capacity of the implementation (or raise an appropriate exception at run time);

4

- Identify all programs or program units that contain errors whose detection is required by this International Standard;

5

- Supply all language-defined library units required by this International Standard;

6

- Contain no variations except those explicitly permitted by this International Standard, or those that are impossible or impractical to avoid given the implementation's execution environment;

7

- Specify all such variations in the manner prescribed by this International Standard.

8

The <external effect> of the execution of an Ada program is defined in terms of its interactions with its external environment. The following are defined as <external interactions>:

9

- Any interaction with an external file (see Section 15.7 [A.7], page 680);

10

- The execution of certain `code_statements` (see Section 14.8 [13.8], page 518); which `code_statements` cause external interactions is implementation defined.

11

- Any call on an imported subprogram (see Chapter 16 [Annex B], page 894), including any parameters passed to it;

12

- Any result returned or exception propagated from a main subprogram (see Section 11.2 [10.2], page 409) or an exported subprogram (see Chapter 16 [Annex B], page 894) to an external caller;

13

- Any read or update of an atomic or volatile object (see Section 17.6 [C.6], page 962);

14

- The values of imported and exported objects (see Chapter 16 [Annex B], page 894) at the time of any other interaction with the external environment.

15

A conforming implementation of this International Standard shall produce for the execution of a given Ada program a set of interactions with the external environment whose order and timing are consistent with the definitions and requirements of this International Standard for the semantics of the given program.

16

An implementation that conforms to this Standard shall support each capability required by the core language as specified. In addition, an implementation that conforms to this Standard may conform to one or more Specialized Needs Annexes (or to none). Conformance to a Specialized Needs Annex means that each capability required by the Annex is provided as specified.

17

An implementation conforming to this International Standard may provide additional attributes, library units, and pragmas. However, it shall not provide any attribute, library unit, or pragma having the same name as an attribute, library unit, or pragma (respectively) specified in a Specialized Needs Annex unless the provided construct is either as

specified in the Specialized Needs Annex or is more limited in capability than that required by the Annex. A program that attempts to use an unsupported capability of an Annex shall either be identified by the implementation before run time or shall raise an exception at run time.

Documentation Requirements

18

Certain aspects of the semantics are defined to be either <implementation defined> or <unspecified>. In such cases, the set of possible effects is specified, and the implementation may choose any effect in the set. Implementations shall document their behavior in implementation–defined situations, but documentation is not required for unspecified situations. The implementation–defined characteristics are summarized in Section 26.2 [M.2], page 1250.

19

The implementation may choose to document implementation–defined behavior either by documenting what happens in general, or by providing some mechanism for the user to determine what happens in a particular case.

Implementation Advice

20

If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise `Program_Error` if feasible.

21

If an implementation wishes to provide implementation–defined extensions to the functionality of a language–defined library unit, it should normally do so by adding children to the library unit.

NOTES

22

2 The above requirements imply that an implementation conforming to this Standard may support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities.

2.1.4 1.1.4 Method of Description and Syntax Notation

1

The form of an Ada program is described by means of a context–free syntax together with context–dependent requirements expressed by narrative rules.

2

The meaning of Ada programs is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs.

3

The context–free syntax of the language is described using a simple variant of Backus–Naur Form. In particular:

4

- Lower case words in a sans–serif font, some containing embedded underlines, are used to denote syntactic categories, for example:

5

`case_statement`

6

- Boldface words are used to denote reserved words, for example:

7

`array`

8

- Square brackets enclose optional items. Thus the two following rules are equivalent.

9/2

`simple_return_statement ::= return [expression];` ■

`simple_return_statement ::= return; | return expression;` ■

10

- Curly brackets enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left–recursive rule. Thus the two following rules are equivalent.

11

`term ::= factor {multiplying_operator factor}` ■

`term ::= factor | term multiplying_operator factor` ■

12

- A vertical line separates alternative items unless it occurs immediately after an opening curly bracket, in which case it stands for itself:

13

`constraint ::= scalar_constraint | composite_constraint` ■

`discrete_choice_list ::= discrete_choice { | discrete_choice }` ■

14

- If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey

some semantic information. For example <subtype_>name and <task_>name are both equivalent to name alone.

14.1/2

The delimiters, compound delimiters, reserved words, and numeric_literals are exclusively made of the characters whose code position is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this International Standard (see Section 3.1 [2.1], page 32) belong to the same range. For example, the character E in the definition of exponent is the character whose name is "LATIN CAPITAL LETTER E", not "GREEK CAPITAL LETTER EPSILON".

14.2/2

When this International Standard mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using locale-independent full case folding, as defined by documents referenced in the note in section 1 of ISO/IEC 10646:2003.

15

A <syntactic category> is a nonterminal in the grammar defined in BNF under "Syntax." Names of syntactic categories are set in a different font, like_this.

16

A <construct> is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under "Syntax".

17

A <constituent> of a construct is the construct itself, or any construct appearing within it.

18

Whenever the run-time semantics defines certain actions to happen in an <arbitrary order>, this means that the implementation shall arrange for these actions to occur in a way that is equivalent to some sequential order, following the rules that result from that sequential order. When evaluations are defined to happen in an arbitrary order, with conversion of the results to some subtypes, or with some run-time checks, the evaluations, conversions, and checks may be arbitrarily interspersed, so long as each expression is evaluated before converting or checking its value. Note that the effect of a program can depend on the order chosen by the implementation. This can happen, for example, if two actual parameters of a given call have side effects.

NOTES

19

3 The syntax rules describing structured constructs are presented in a form that corresponds to the recommended paragraphing. For example, an if_statement is defined as:

20

```
if_statement ::=
    if condition then
        sequence_of_statements
    {elsif condition then
```

```
        sequence_of_statements}
[else
    sequence_of_statements]
end if;
```

21

4 The line breaks and indentation in the syntax rules indicate the recommended line breaks and indentation in the corresponding constructs. The preferred places for other line breaks are after semicolons.

2.1.5 1.1.5 Classification of Errors

Implementation Requirements

1

The language definition classifies errors into several different categories:

2

- Errors that are required to be detected prior to run time by every Ada implementation;

3

These errors correspond to any violation of a rule given in this International Standard, other than those listed below. In particular, violation of any rule that uses the terms shall, allowed, permitted, legal, or illegal belongs to this category. Any program that contains such an error is not a legal Ada program; on the other hand, the fact that a program is legal does not mean, <per se>, that the program is free from other forms of error.

4

The rules are further classified as either compile time rules, or post compilation rules, depending on whether a violation has to be detected at the time a compilation unit is submitted to the compiler, or may be postponed until the time a compilation unit is incorporated into a partition of a program.

5

- Errors that are required to be detected at run time by the execution of an Ada program;

6

The corresponding error situations are associated with the names of the predefined exceptions. Every Ada compiler is required to generate code that raises the corresponding exception if such an error situation arises during program execution. If such an error situation

is certain to arise in every execution of a construct, then an implementation is allowed (although not required) to report this fact at compilation time.

7

- Bounded errors;

8

The language rules define certain kinds of errors that need not be detected either prior to or during run time, but if not detected, the range of possible effects shall be bounded. The errors of this category are called <bounded errors>. The possible effects of a given bounded error are specified for each such error, but in any case one possible effect of a bounded error is the raising of the exception `Program_Error`.

9

- Erroneous execution.

10

In addition to bounded errors, the language rules define certain kinds of errors as leading to <erroneous execution>. Like bounded errors, the implementation need not detect such errors either prior to or during run time. Unlike bounded errors, there is no language—specified bound on the possible effect of erroneous execution; the effect is in general not predictable.

Implementation Permissions

11

An implementation may provide <nonstandard modes> of operation. Typically these modes would be selected by a pragma or by a command line switch when the compiler is invoked. When operating in a nonstandard mode, the implementation may reject `compilation_units` that do not conform to additional requirements associated with the mode, such as an excessive number of warnings or violation of coding style guidelines. Similarly, in a nonstandard mode, the implementation may apply special optimizations or alternative algorithms that are only meaningful for programs that satisfy certain criteria specified by the implementation. In any case, an implementation shall support a <standard> mode that conforms to the requirements of this International Standard; in particular, in the standard mode, all legal `compilation_units` shall be accepted.

Implementation Advice

12

If an implementation detects a bounded error or erroneous execution, it should raise `Program_Error`.

2.2 1.2 Normative References

1

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

2

ISO/IEC 646:1991, <Information technology -- ISO 7-bit coded character set for information interchange>.

3/2

ISO/IEC 1539-1:2004, <Information technology -- Programming languages -- Fortran -- Part 1: Base language>.

4/2

ISO/IEC 1989:2002, <Information technology -- Programming languages -- COBOL>.

5

ISO/IEC 6429:1992, <Information technology -- Control functions for coded graphic character sets>.

5.1/2

ISO 8601:2004, <Data elements and interchange formats -- Information interchange -- Representation of dates and times>.

6

ISO/IEC 8859-1:1987, <Information processing -- 8-bit single-byte coded character sets -- Part 1: Latin alphabet No. 1>.

7/2

ISO/IEC 9899:1999, <Programming languages -- C>, supplemented by Technical Corrigendum 1:2001 and Technical Corrigendum 2:2004.

8/2

ISO/IEC 10646:2003, <Information technology -- Universal Multiple-Octet Coded Character Set (UCS)>.

9/2

ISO/IEC 14882:2003, <Programming languages -- C++>.

10/2

ISO/IEC TR 19769:2004, <Information technology -- Programming languages, their environments and system software interfaces -- Extensions for the programming language C to support new character data types>.

2.3 1.3 Definitions

1/2

Terms are defined throughout this International Standard, indicated by <italic> type. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Mathematical terms not defined in this International

Standard are to be interpreted according to the <CRC Concise Encyclopedia of Mathematics, Second Edition>. Other terms not defined in this International Standard are to be interpreted according to the <Webster's Third New International Dictionary of the English Language>. Informal descriptions of some terms are also given in Chapter 27 [Annex N], page 1283, "Chapter 27 [Annex N], page 1283, Glossary".

3 2 Lexical Elements

1

The text of a program consists of the texts of one or more compilations. The text of a compilation is a sequence of lexical elements, each composed of characters; the rules of composition are given in this section. Pragmas, which provide certain information for the compiler, are also described in this section.

3.1 2.1 Character Set

1/2

The character repertoire for the text of an Ada program consists of the entire coding space described by the ISO/IEC 10646:2003 Universal Multiple–Octet Coded Character Set. This coding space is organized in <planes>, each plane comprising 65536 characters.

Syntax

<Paragraphs 2 and 3 were deleted.>

3.1/2

A character is defined by this International Standard for each cell in the coding space described by ISO/IEC 10646:2003, regardless of whether or not ISO/IEC 10646:2003 allocates a character to that cell.

Static Semantics

4/2

The coded representation for characters is implementation defined (it need not be a representation defined within ISO/IEC 10646:2003). A character whose relative code position in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program.

4.1/2

The semantics of an Ada program whose text is not in Normalization Form KC (as defined by section 24 of ISO/IEC 10646:2003) is implementation defined.

5/2

The description of the language definition in this International Standard uses the character properties General Category, Simple Uppercase Mapping, Uppercase Mapping, and Special Case Condition of the documents referenced by the note in section 1 of ISO/IEC 10646:2003. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified.

6/2

Characters are categorized as follows:

7/2

<This paragraph was
deleted.>

8/2

letter_uppercase

9/2 letter_lowercase	Any character whose General Category is defined to be "Letter, Uppercase".
9.1/2 letter_titlecase	Any character whose General Category is defined to be "Letter, Lowercase".
9.2/2 letter_modifier	Any character whose General Category is defined to be "Letter, Titlecase".
9.3/2 letter_other	Any character whose General Category is defined to be "Letter, Modifier".
9.4/2 mark_non_spacing	Any character whose General Category is defined to be "Letter, Other".
9.5/2 mark_spacing_combining	Any character whose General Category is defined to be "Mark, Non–Spacing".
10/2 number_decimal	Any character whose General Category is defined to be "Mark, Spacing Combining".
	Any character whose General Category

10.1/2 number_letter	is defined to be "Number, Decimal".
10.2/2 punctuation_connector	Any character whose General Category is defined to be "Number, Letter".
10.3/2 other_format	Any character whose General Category is defined to be "Punctuation, Connector".
11/2 separator_space	Any character whose General Category is defined to be "Other, Format".
12/2 separator_line	Any character whose General Category is defined to be "Separator, Space".
12.1/2 separator_paragraph	Any character whose General Category is defined to be "Separator, Line".
13/2 format_effector	Any character whose General Category is defined to be "Separator, Paragraph".
	The characters whose code positions

are 16#09#
(CHARACTER
TABULATION),
16#0A# (LINE
FEED), 16#0B#
(LINE TABULA-
TION), 16#0C#
(FORM FEED),
16#0D# (CAR-
RIAGE RETURN),
16#85# (NEXT
LINE), and the char-
acters in categories
separator_line and
separator_paragraph.

13.1/2
other_control

Any character whose
General Category is
defined to be "Other,
Control", and which
is not defined to be a
format_effector.

13.2/2
other_private_use

Any character whose
General Category is
defined to be "Other,
Private Use".

13.3/2
other_surrogate

Any character whose
General Category is
defined to be "Other,
Surrogate".

14/2
graphic_character

Any character that is
not in the categories
other_control,
other_private_use,
other_surrogate,
format_effector, and
whose relative code

position in its plane is
neither 16#FFFE#
nor 16#FFFF#.

15/2

The following names are used when referring to certain characters (the first name is that given in ISO/IEC 10646:2003):

graphic symbol	name	graphic symbol	name
"	quotation mark	:	colon
#	number sign	;	semicolon
&	ampersand	<	less-than sign
'	apostrophe, tick	=	equals sign
(left parenthesis	>	greater-than sign
)	right parenthesis	-	low line, underscore
*	asterisk, multiply		vertical line
+	plus sign	/	solidus, division sign
,	comma	!	exclamation point
-	hyphen-minus, minus	%	percent sign
.	full stop, dot, point		

Implementation Permissions

16/2

<This paragraph was deleted.>

NOTES

17/2

1 The characters in categories `other_control`, `other_private_use`, and `other_surrogate` are only allowed in comments.

18

2 The language does not specify the source representation of programs.

3.2 2.2 Lexical Elements, Separators, and Delimiters

Static Semantics

1

The text of a program consists of the texts of one or more compilations. The text of each compilation is a sequence of separate <lexical elements>. Each lexical element is formed from a sequence of characters, and is either a delimiter, an identifier, a reserved word, a `numeric_literal`, a `character_literal`, a `string_literal`, or a comment. The meaning of a program depends only on the particular sequences of lexical elements that form its compilations, excluding comments.

2/2

The text of a compilation is divided into <lines>. In general, the representation for an end of line is implementation defined. However, a sequence of one or more `format_effectors`

other than the character whose code position is 16#09# (CHARACTER TABULATION) signifies at least one end of line.

3/2

In some cases an explicit <separator> is required to separate adjacent lexical elements. A separator is any of a separator_space, a format_effector, or the end of a line, as follows:

4/2

- A separator_space is a separator except within a comment, a string_literal, or a character_literal.

5/2

- The character whose code position is 16#09# (CHARACTER TABULATION) is a separator except within a comment.

6

- The end of a line is always a separator.

7

One or more separators are allowed between any two adjacent lexical elements, before the first of each compilation, or after the last. At least one separator is required between an identifier, a reserved word, or a numeric_literal and an adjacent identifier, reserved word, or numeric_literal.

8/2

A <delimiter> is either one of the following characters:

9

& ' () * + , - . / : ; < = > |

10

or one of the following <compound delimiters> each composed of two adjacent special characters

11

=> .. ** := /= >= <= << >> <>

12

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string_literal, character_literal, or numeric_literal.

13

The following names are used when referring to compound delimiters:

delimiter	name
=>	arrow
..	double dot

**	double star, exponentiate
:=	assignment (pronounced: "becomes")
/=	inequality (pronounced: "not equal")
>=	greater than or equal
<=	less than or equal
<<	left label bracket
>>	right label bracket
<>	box

Implementation Requirements

14

An implementation shall support lines of at least 200 characters in length, not counting any characters used to signify the end of a line. An implementation shall support lexical elements of at least 200 characters in length. The maximum supported line length and lexical element length are implementation defined.

3.3 2.3 Identifiers

1

Identifiers are used as names.

Syntax

2/2

```
identifier ::=
    identifier_start {identifier_start | identifier_extend}
```

3/2

```
identifier_start ::=
    letter_uppercase
    | letter_lowercase
    | letter_titlecase
    | letter_modifier
    | letter_other
    | number_letter
```

3.1/2

```
identifier_extend ::=
    mark_non_spacing
    | mark_spacing_combining
    | number_decimal
    | punctuation_connector
    | other_format
```

4/2

After eliminating the characters in category `other_format`, an identifier shall not contain two consecutive characters in category `punctuation_connector`, or end with a character in that category.

Static Semantics

5/2

Two identifiers are considered the same if they consist of the same sequence of characters after applying the following transformations (in this order):

5.1/2

- The characters in category `other_format` are eliminated.

5.2/2

- The remaining sequence of characters is converted to upper case.

5.3/2

After applying these transformations, an identifier shall not be identical to a reserved word (in upper case).

Implementation Permissions

6

In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers, to accommodate local conventions.

NOTES

6.1/2

3 Identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

Examples

7

<Examples of identifiers:>

8/2

```
Count      X   Get_Symbol  Ethelyn  Marion
Snobol_4   X1  Page_Count  Store_Next_Item
[Unicode 928] [Unicode 955] [Unicode 940] [Unicode 964] [Unicode 969] [Uni-█
code 957]      --< Plato>
[Unicode 1063] [Unicode 1072] [Unicode 1081] [Unicode 1082] [Unicode 1086] [Uni-█
code 1074] [Unicode 1089] [Unicode 1082] [Unicode 1080] [Unicode 1081]  --< Tchaikovs
[Unicode 952]  [Unicode 966]          --< Angles>
```

3.4 2.4 Numeric Literals

1

There are two kinds of `numeric_literals`, `<real literals>` and `<integer literals>`. A real literal is a `numeric_literal` that includes a point; an integer literal is a `numeric_literal` without a point.

Syntax

2

numeric_literal ::= decimal_literal | based_literal

NOTES

3

4 The type of an integer literal is <universal_integer>. The type of a real literal is <universal_real>.

3.4.1 2.4.1 Decimal Literals

1

A decimal_literal is a numeric_literal in the conventional decimal notation (that is, the base is ten).

Syntax

2

decimal_literal ::= numeral [.numeral] [exponent]

3

numeral ::= digit {[underline] digit}

4

exponent ::= E [+] numeral | E - numeral

4.1/2

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

5

An exponent for an integer literal shall not have a minus sign.

Static Semantics

6

An underline character in a numeric_literal does not affect its meaning. The letter E of an exponent can be written either in lower case or in upper case, with the same meaning.

7

An exponent indicates the power of ten by which the value of the decimal_literal without the exponent is to be multiplied to obtain the value of the decimal_literal with the exponent.

Examples

8

<Examples of decimal literals:>

9

12 0 1E6 123_456 --< integer literals>

12.0 0.0 0.456 3.14159_26 --< real literals>

3.4.2 2.4.2 Based Literals

1

A `based_literal` is a `numeric_literal` expressed in a form that specifies the base explicitly.

Syntax

2

```
based_literal ::=  
    base # based_numeral [.,based_numeral] # [exponent]
```

3

```
base ::= numeral
```

4

```
based_numeral ::=  
    extended_digit {[underline] extended_digit}
```

5

```
extended_digit ::= digit | A | B | C | D | E | F
```

Legality Rules

6

The `<base>` (the numeric value of the decimal numeral preceding the first `#`) shall be at least two and at most sixteen. The extended_digits A through F represent the digits ten through fifteen, respectively. The value of each extended_digit of a `based_literal` shall be less than the base.

Static Semantics

7

The conventional meaning of based notation is assumed. An exponent indicates the power of the base by which the value of the `based_literal` without the exponent is to be multiplied to obtain the value of the `based_literal` with the exponent. The base and the exponent, if any, are in decimal notation.

8

The extended_digits A through F can be written either in lower case or in upper case, with the same meaning.

Examples

9

`<Examples of based literals:>`

10

```
2#1111_1111# 16#FF#      016#0ff#    --< integer literals of value 255>■  
16#E#E1      2#1110_0000#    --< integer literals of value 224>■  
16#F.FF#E+2  2#1.1111_1111_1110#E11 --< real literals of value 4095.0>■
```

3.5 2.5 Character Literals

1
A `character_literal` is formed by enclosing a graphic character between two apostrophe characters.

Syntax

2

```
character_literal ::= 'graphic_character'
```

NOTES

3

5 A `character_literal` is an enumeration literal of a character type.
See Section 4.5.2 [3.5.2], page 93.

Examples

4

<Examples of character literals:>

5/2

```
'A'      '*'      ''      ' , '
```

```
'L'      '[Unicode 1051]'      '[Unicode 923]'      --< Various els.>
```

```
'[Unicode 8734]'      '[Unicode 1488]'      --< Big numbers - infinity and .
```

3.6 2.6 String Literals

1
A `string_literal` is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets. They are used to represent operator_symbols (see Section 7.1 [6.1], page 255), values of a string type (see Section 5.2 [4.2], page 189), and array subaggregates (see Section 5.3.3 [4.3.3], page 196).

Syntax

2

```
string_literal ::= "{string_element}"
```

3

```
string_element ::= "" | <non_quotation_mark_>graphic_character
```

4

A `string_element` is either a pair of quotation marks (""), or a single `graphic_character` other than a quotation mark.

Static Semantics

5

The <sequence of characters> of a `string_literal` is formed from the sequence of

string_elements between the bracketing quotation marks, in the given order, with a string_element that is "" becoming a single quotation mark in the sequence of characters, and any other string_element being reproduced in the sequence.

6

A <null string literal> is a string_literal with no string_elements between the quotation marks.

NOTES

7

6 An end of line cannot appear in a string_literal.

7.1/2

7 No transformation is performed on the sequence of characters of a string_literal.

Examples

8

<Examples of string literals:>

9/2

```
"Message of the day:"
```

```
""                --< a null string literal>
" " "A" """"     --< three string literals of length 1>
```

```
"Characters such as $, %, and } are allowed in string literals"
```

```
"Archimedes said ""[Unicode 917][Unicode 973][Unicode 961][Unicode 951][Uni-█
code 954][Unicode 945]""
```

```
"Volume of cylinder (PIr2h) = "
```

3.7 2.7 Comments

1

A comment starts with two adjacent hyphens and extends up to the end of the line.

Syntax

2

```
comment ::= --{<non_end_of_line_>character}
```

3

A comment may appear on any line of a program.

Static Semantics

4

The presence or absence of comments has no influence on whether a program is legal or

illegal. Furthermore, comments do not influence the meaning of a program; their sole purpose is the enlightenment of the human reader.

Examples

5

<Examples of comments:>

6

```
--< the last sentence above echoes the Algol 68 report >

end;  --< processing of Line is complete >

--< a long comment may be split onto>
--< two or more consecutive lines  >

-----< the first two hyphens start the comment >■
```

3.8 2.8 Pragma

1

A pragma is a compiler directive. There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation-defined) pragmas.

Syntax

2

```
pragma ::=
  pragma identifier [(pragma_argument_association { , pragma_argument_association})];
```

3

```
pragma_argument_association ::=
  [<pragma_argument_>identifier =>] name
  | [<pragma_argument_>identifier =>] expression
```

4

In a pragma, any pragma_argument_associations without a <pragma_argument_>identifier shall precede any associations with a <pragma_argument_>identifier.

5

Pragmas are only allowed at the following places in a program:

6

- After a semicolon delimiter, but not within a formal_part or discriminant_part.

7

- At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "statement", "clause", or "alternative", or one of the syntactic categories `variant` or `exception_handler`; but not in place of such a construct. Also at any place where a `compilation_unit` would be allowed.

8

Additional syntax rules and placement restrictions exist for specific pragmas.

9

The `<name>` of a pragma is the identifier following the reserved word `pragma`. The name or expression of a `pragma_argument_association` is a `<pragma argument>`.

10

An `<identifier specific to a pragma>` is an identifier that is used in a pragma argument with special meaning for that pragma.

Static Semantics

11

If an implementation does not recognize the name of a pragma, then it has no effect on the semantics of the program. Inside such a pragma, the only rules that apply are the Syntax Rules.

Dynamic Semantics

12

Any pragma that appears at the place of an executable construct is executed. Unless otherwise specified for a particular pragma, this execution consists of the evaluation of each evaluable pragma argument in an arbitrary order.

Implementation Requirements

13

The implementation shall give a warning message for an unrecognized pragma name.

Implementation Permissions

14

An implementation may provide implementation-defined pragmas; the name of an implementation-defined pragma shall differ from those of the language-defined pragmas.

15

An implementation may ignore an unrecognized pragma even if it violates some of the Syntax Rules, if detecting the syntax error is too complex.

Implementation Advice

16

Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas are removed from a working program, the program should still be legal, and should still have the same semantics.

17

Normally, an implementation should not define pragmas that can make an illegal program legal, except as follows:

18

- A pragma used to complete a declaration, such as a pragma Import;

19

- A pragma used to configure the environment by adding, removing, or replacing library_items.

Syntax

20

The forms of List, Page, and Optimize pragmas are as follows:

21

```
pragma List(identifier);
```

22

```
pragma Page;
```

23

```
pragma Optimize(identifier);
```

24

Other pragmas are defined throughout this International Standard, and are summarized in Chapter 25 [Annex L], page 1242.

Static Semantics

25

A pragma List takes one of the identifiers On or Off as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a List pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.

26

A pragma Page is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).

27

A pragma Optimize takes one of the identifiers Time, Space, or Off as the single argument. This pragma is allowed anywhere a pragma is allowed, and it applies until the end of the immediately enclosing declarative region, or for a pragma at the place of a compilation_unit, to the end of the compilation. It gives advice to the implementation as to whether time or

space is the primary optimization criterion, or that optional optimizations should be turned off. It is implementation defined how this advice is followed.

Examples

28

<Examples of pragmas:>

29/2

```
pragma List(Off); --< turn off listing generation>
pragma Optimize(Off); --< turn off optional optimizations>
pragma Inline(Set_Mask); --< generate code for Set_Mask inline>
pragma Import(C, Put_Char, External_Name => "putchar"); --< import C putchar func
```

3.9 2.9 Reserved Words

Syntax

1/1

<This paragraph was deleted.>

2/2

The following are the <reserved words>. Within a program, some or all of the letters of a reserved word may be in upper case, and one or more characters in category other_format may be inserted within or at the end of the reserved word.

abort	else	new	return
abs	elsif	not	reverse
abstract	end	null	
accept	entry		select
access	exception	of	separate
aliased	exit	or	subtype
all		others	synchronized
and	for	out	
array	function	overriding	tagged
at			task
	generic	package	terminate
begin	goto	pragma	then
body		private	type
	if	procedure	
case	in	protected	until
constant	interface		use
	is	raise	
declare		range	when
delay	limited	record	while
delta	loop	rem	with
digits		renames	

do

mod

requeue

xor

NOTES

3

8 The reserved words appear in lower case boldface in this International Standard, except when used in the designator of an attribute (see Section 5.1.4 [4.1.4], page 187). Lower case boldface is also used for a reserved word in a string_literal used as an operator_symbol. This is merely a convention — programs may be written in whatever typeface is desired and available.

4 3 Declarations and Types

1

This section describes the types in the language and the rules for declaring constants, variables, and named numbers.

4.1 3.1 Declarations

1

The language defines several kinds of named <entities> that are declared by declarations. The entity's <name> is defined by the declaration, usually by a `defining_identifier` (see [S0022], page 49), but sometimes by a `defining_character_literal` (see [S0040], page 92) or `defining_operator_symbol` (see [S0156], page 256).

2

There are several forms of declaration. A `basic_declaration` is a form of declaration defined as follows.

Syntax

3/2

```
basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | null_procedure_declaration | package_declaration
  | renaming_declaration | exception_declaration
  | generic_declaration | generic_instantiation
```

4

```
defining_identifier ::= identifier
```

Static Semantics

5

A <declaration> is a language construct that associates a name with (a view of) an entity. A declaration may appear explicitly in the program text (an <explicit> declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an <implicit> declaration).

6/2

Each of the following is defined to be a declaration: any `basic_declaration` (see [S0021], page 49); an `enumeration_literal_specification` (see [S0039], page 92); a `discriminant_specification` (see [S0062], page 123); a `component_declaration` (see [S0070], page 130); a `loop_parameter_specification` (see [S0144], page 249); a `parameter_specification` (see [S0160], page 256); a `subprogram_body` (see [S0162], page 261); an `entry_declaration` (see [S0200], page 347); an `entry_index_specification` (see [S0206], page 348); a `choice_parameter_specification` (see [S0249], page 420); a `generic_formal_parameter_declaration` (see [S0256], page 451). In addition, an `extended_return_statement` is a declaration of its `defining_identifier`.

7

All declarations contain a <definition> for a <view> of an entity. A view consists of an identification of the entity (the entity <of> the view), plus view-specific characteristics that affect the use of the entity through that view (such as mode of access to an object, formal parameter names and defaults for a subprogram, or visibility to components of a type). In most cases, a declaration also contains the definition for the entity itself (a renaming_declaration is an example of a declaration that does not define a new entity, but instead defines a view of an existing entity (see Section 9.5 [8.5], page 316)).

8

For each declaration, the language rules define a certain region of text called the <scope> of the declaration (see Section 9.2 [8.2], page 306). Most declarations associate an identifier with a declared entity. Within its scope, and only there, there are places where it is possible to use the identifier to refer to the declaration, the view it defines, and the associated entity; these places are defined by the visibility rules (see Section 9.3 [8.3], page 308). At such places the identifier is said to be a <name> of the entity (the direct_name or selector_name); the name is said to <denote> the declaration, the view, and the associated entity (see Section 9.6 [8.6], page 324). The declaration is said to <declare> the name, the view, and in most cases, the entity itself.

9

As an alternative to an identifier, an enumeration literal can be declared with a character_literal as its name (see Section 4.5.1 [3.5.1], page 92), and a function can be declared with an operator_symbol as its name (see Section 7.1 [6.1], page 255).

10

The syntax rules use the terms defining_identifier, defining_character_literal (see [S0040], page 92), and defining_operator_symbol (see [S0156], page 256) for the defining occurrence of a name; these are collectively called <defining names>. The terms direct_name and selector_name are used for usage occurrences of identifiers, character_literals, and operator_symbols. These are collectively called <usage names>.

Dynamic Semantics

11

The process by which a construct achieves its run-time effect is called <execution>. This process is also called <elaboration> for declarations and <evaluation> for expressions. One of the terms execution, elaboration, or evaluation is defined by this International Standard for each construct that has a run-time effect.

NOTES

12

- 1 At compile time, the declaration of an entity <declares> the entity.
At run time, the elaboration of the declaration <creates> the entity.

4.2 3.2 Types and Subtypes

Static Semantics

1

A <type> is characterized by a set of values, and a set of <primitive operations> which

implement the fundamental aspects of its semantics. An <object> of a given type is a run-time entity that contains (has) a value of the type.

2/2

Types are grouped into <categories> of types. There exist several <language-defined categories> of types (see NOTES below), reflecting the similarity of their values and primitive operations. Most categories of types form <classes> of types. <Elementary> types are those whose values are logically indivisible; <composite> types are those whose values are composed of <component> values.

3

The elementary types are the <scalar> types (<discrete> and <real>) and the <access> types (whose values provide access to objects or subprograms). Discrete types are either <integer> types or are defined by enumeration of their values (<enumeration> types). Real types are either <floating point> types or <fixed point> types.

4/2

The composite types are the <record> types, <record extensions>, <array> types, <interface> types, <task> types, and <protected> types.

4.1/2

There can be multiple views of a type with varying sets of operations. An <incomplete> type represents an incomplete view (see Section 4.10.1 [3.10.1], page 160) of a type with a very restricted usage, providing support for recursive data structures. A <private> type or <private extension> represents a partial view (see Section 8.3 [7.3], page 283) of a type, providing support for data abstraction. The full view (see Section 4.2.1 [3.2.1], page 53) of a type represents its complete definition. An incomplete or partial view is considered a composite type, even if the full view is not.

5/2

Certain composite types (and views thereof) have special components called <discriminants> whose values affect the presence, constraints, or initialization of other components. Discriminants can be thought of as parameters of the type.

6/2

The term <subcomponent> is used in this International Standard in place of the term component to indicate either a component, or a component of another subcomponent. Where other subcomponents are excluded, the term component is used instead. Similarly, a <part> of an object or value is used to mean the whole object or value, or any set of its subcomponents. The terms component, subcomponent, and part are also applied to a type meaning the component, subcomponent, or part of objects and values of the type.

7/2

The set of possible values for an object of a given type can be subjected to a condition that is called a <constraint> (the case of a <null constraint> that specifies no restriction is also included); the rules for which values satisfy a given kind of constraint are given in Section 4.5 [3.5], page 76, for range_constraints, Section 4.6.1 [3.6.1], page 117, for index_constraints, and Section 4.7.1 [3.7.1], page 127, for discriminant_constraints. The set of possible values for an object of an access type can also be subjected to a condition that excludes the null value (see Section 4.10 [3.10], page 156).

8/2

A <subtype> of a given type is a combination of the type, a constraint on values of the

type, and certain attributes specific to the subtype. The given type is called the <type of the subtype>. Similarly, the associated constraint is called the <constraint of the subtype>.

The set of values of a subtype consists of the values of its type that satisfy its constraint and any exclusion of the null value. Such values <belong> to the subtype.

9

A subtype is called an <unconstrained> subtype if its type has unknown discriminants, or if its type allows range, index, or discriminant constraints, but the subtype does not impose such a constraint; otherwise, the subtype is called a <constrained> subtype (since it has no unconstrained characteristics).

NOTES

10/2

2 Any set of types can be called a "category" of types, and any set of types that is closed under derivation (see Section 4.4 [3.4], page 66) can be called a "class" of types. However, only certain categories and classes are used in the description of the rules of the language — generally those that have their own particular set of primitive operations (see Section 4.2.3 [3.2.3], page 57), or that correspond to a set of types that are matched by a given kind of generic formal type (see Section 13.5 [12.5], page 460). The following are examples of "interesting" <language-defined classes>: elementary, scalar, discrete, enumeration, character, boolean, integer, signed integer, modular, real, floating point, fixed point, ordinary fixed point, decimal fixed point, numeric, access, access-to-object, access-to-subprogram, composite, array, string, (untagged) record, tagged, task, protected, nonlimited. Special syntax is provided to define types in each of these classes. In addition to these classes, the following are examples of "interesting" <language-defined categories>: abstract, incomplete, interface, limited, private, record.

11/2

These language-defined categories are organized like this:

12/2

- all types
 - elementary
 - scalar
 - discrete
 - enumeration
 - character
 - boolean
 - other enumeration
 - integer
 - signed integer
 - modular integer

- real
 - floating point
 - fixed point
 - ordinary fixed point
 - decimal fixed point
- access
 - access-to-object
 - access-to-subprogram
- composite
 - untagged
 - array
 - string
 - other array
 - record
 - task
 - protected
 - tagged (including interfaces)
 - nonlimited tagged record
 - limited tagged
 - limited tagged record
 - synchronized tagged
 - tagged task
 - tagged protected

13/2

There are other categories, such as "numeric" and "discriminated", which represent other categorization dimensions, but do not fit into the above strictly hierarchical picture.

4.2.1 3.2.1 Type Declarations

1

A `type_declaration` declares a type and its first subtype.

Syntax

2

```

type_declaration ::= full_type_declaration
                  | incomplete_type_declaration
                  | private_type_declaration
                  | private_extension_declaration
  
```

3

```

full_type_declaration ::=
  type_defining_identifier [known_discriminant_part] is type_definition;

  | task_type_declaration
  | protected_type_declaration
  
```



4/2

```
type_definition ::=
    enumeration_type_definition | integer_type_definition
    | real_type_definition | array_type_definition
    | record_type_definition | access_type_definition
    | derived_type_definition | interface_type_definition
```

Legality Rules

5

A given type shall not have a subcomponent whose type is the given type itself.

Static Semantics

6

The defining_identifier (see [S0022], page 49) of a type_declaration (see [S0023], page 53) denotes the <first subtype> of the type. The known_discriminant_part (see [S0061], page 123), if any, defines the discriminants of the type (see Section 4.7 [3.7], page 123, "Section 4.7 [3.7], page 123, Discriminants"). The remainder of the type_declaration (see [S0023], page 53) defines the remaining characteristics of (the view of) the type.

7/2

A type defined by a type_declaration (see [S0023], page 53) is a <named> type; such a type has one or more nameable subtypes. Certain other forms of declaration also include type definitions as part of the declaration for an object. The type defined by such a declaration is <anonymous> -- it has no nameable subtypes. For explanatory purposes, this International Standard sometimes refers to an anonymous type by a pseudo-name, written in italics, and uses such pseudo-names at places where the syntax normally requires an identifier. For a named type whose first subtype is T, this International Standard sometimes refers to the type of T as simply "the type T".

8/2

A named type that is declared by a full_type_declaration (see [S0024], page 53), or an anonymous type that is defined by an access_definition or as part of declaring an object of the type, is called a <full type>. The declaration of a full type also declares the <full view> of the type. The type_definition (see [S0025], page 54), task_definition (see [S0190], page 329), protected_definition (see [S0195], page 338), or access_definition (see [S0084], page 156) that defines a full type is called a <full type definition>. Types declared by other forms of type_declaration (see [S0023], page 53) are not separate types; they are partial or incomplete views of some full type.

9

The definition of a type implicitly declares certain <predefined operators> that operate on the type, according to what classes the type belongs, as specified in Section 5.5 [4.5], page 203, "Section 5.5 [4.5], page 203, Operators and Expression Evaluation".

10

The <predefined types> (for example the types Boolean, Wide_Character, Integer, <root_integer>, and <universal_integer>) are the types that are defined in a predefined library package called Standard; this package also includes the (implicit) declarations of their predefined operators. The package Standard is described in Section 15.1 [A.1], page 556.

Dynamic Semantics

11

The elaboration of a `full_type_declaration` consists of the elaboration of the full type definition. Each elaboration of a full type definition creates a distinct type and its first subtype.

Examples

12

<Examples of type definitions:>

13

```
(White, Red, Yellow, Green, Blue, Brown, Black)
range 1 .. 72
array(1 .. 10) of Integer
```

14

<Examples of type declarations:>

15

```
type Color is (White, Red, Yellow, Green, Blue, Brown, Black);
type Column is range 1 .. 72;
type Table is array(1 .. 10) of Integer;
NOTES
```

16

3 Each of the above examples declares a named type. The identifier given denotes the first subtype of the type. Other named subtypes of the type can be declared with `subtype_declarations` (see Section 4.2.2 [3.2.2], page 55). Although names do not directly denote types, a phrase like "the type Column" is sometimes used in this International Standard to refer to the type of Column, where Column denotes the first subtype of the type. For an example of the definition of an anonymous type, see the declaration of the array `Color_Table` in Section 4.3.1 [3.3.1], page 61; its type is anonymous — it has no nameable subtypes.

4.2.2 3.2.2 Subtype Declarations

1

A `subtype_declaration` declares a subtype of some previously declared type, as defined by a `subtype_indication`.

Syntax

2

```
subtype_declaration ::=
    subtype defining_identifier is subtype_indication;
```

3/2

4 subtype_indication ::= [null_exclusion] subtype_mark [constraint]

5 subtype_mark ::= <subtype_>name

6 constraint ::= scalar_constraint | composite_constraint

7 scalar_constraint ::=
 range_constraint | digits_constraint | delta_constraint

 composite_constraint ::=
 index_constraint | discriminant_constraint
 Name Resolution Rules

8 A subtype_mark shall resolve to denote a subtype. The type <determined by> a subtype_mark is the type of the subtype denoted by the subtype_mark.

Dynamic Semantics

9 The elaboration of a subtype_declaration consists of the elaboration of the subtype_indication. The elaboration of a subtype_indication creates a new subtype. If the subtype_indication does not include a constraint, the new subtype has the same (possibly null) constraint as that denoted by the subtype_mark. The elaboration of a subtype_indication that includes a constraint proceeds as follows:

10

- The constraint is first elaborated.

11

- A check is then made that the constraint is <compatible> with the subtype denoted by the subtype_mark.

12

The condition imposed by a constraint is the condition obtained after elaboration of the constraint. The rules defining compatibility are given for each form of constraint in the appropriate subclause. These rules are such that if a constraint is <compatible> with a subtype, then the condition imposed by the constraint cannot contradict any condition already imposed by the subtype on its values. The exception Constraint_Error is raised if any check of compatibility fails.

NOTES

13

4 A `scalar_constraint` may be applied to a subtype of an appropriate scalar type (see Section 4.5 [3.5], page 76, Section 4.5.9 [3.5.9], page 106, and Section 23.3 [J.3], page 1167), even if the subtype is already constrained. On the other hand, a `composite_constraint` may be applied to a composite subtype (or an `access-to-composite` subtype) only if the composite subtype is unconstrained (see Section 4.6.1 [3.6.1], page 117, and Section 4.7.1 [3.7.1], page 127).

Examples

14

<Examples of subtype declarations:>

15/2

```

subtype Rainbow    is Color range Red .. Blue;           --< see Section 4.2.1
[3.2.1], page 53>
subtype Red_Blue   is Rainbow;
subtype Int        is Integer;
subtype Small_Int  is Integer range -10 .. 10;
subtype Up_To_K    is Column range 1 .. K;             --< see Section 4.2.1
[3.2.1], page 53>
subtype Square     is Matrix(1 .. 10, 1 .. 10);        --< see Section 4.6
[3.6], page 114>
subtype Male       is Person(Sex => M);                --< see Section 4.10.1
[3.10.1], page 160>
subtype Binop_Ref  is not null Binop_Ptr;             --< see Section 4.10
[3.10], page 156>

```

4.2.3 3.2.3 Classification of Operations

Static Semantics

1/2

An operation <operates on a type> <T> if it yields a value of type <T>, if it has an operand whose expected type (see Section 9.6 [8.6], page 324) is <T>, or if it has an access parameter or access result type (see Section 7.1 [6.1], page 255) designating <T>. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a <predefined operation> of the type. The <primitive operations> of a type are the predefined operations of the type, plus any user-defined primitive subprograms.

2

The <primitive subprograms> of a specific type are defined as follows:

3

- The predefined operators of the type (see Section 5.5 [4.5], page 203);

4

- For a derived type, the inherited (see Section 4.4 [3.4], page 66) user-defined subprograms;

5

- For an enumeration type, the enumeration literals (which are considered parameterless functions — see Section 4.5.1 [3.5.1], page 92);

6

- For a specific type declared immediately within a `package_specification`, any subprograms (in addition to the enumeration literals) that are explicitly declared immediately within the same `package_specification` and that operate on the type;

7/2

- For a nonformal type, any subprograms not covered above that are explicitly declared immediately within the same declarative region as the type and that override (see Section 9.3 [8.3], page 308) other implicitly declared primitive subprograms of the type.

8

A primitive subprogram whose designator is an `operator_symbol` is called a `<primitive operator>`.

4.3 3.3 Objects and Named Numbers

1

Objects are created at run time and contain a value of a given type. An object can be created and initialized as part of elaborating a declaration, evaluating an allocator, aggregate, or `function_call`, or passing a parameter by copy. Prior to reclaiming the storage for an object, it is finalized if necessary (see Section 8.6.1 [7.6.1], page 299).

Static Semantics

2

All of the following are objects:

3

- the entity declared by an `object_declaration`;

4

- a formal parameter of a subprogram, entry, or generic subprogram;

5

- a generic formal object;

6

- a loop parameter;

7

- a choice parameter of an `exception_handler`;

8

- an entry index of an `entry_body`;

9

- the result of dereferencing an `access-to-object` value (see Section 5.1 [4.1], page 179);

10/2

- the return object created as the result of evaluating a `function_call` (or the equivalent operator invocation — see Section 7.6 [6.6], page 276);

11

- the result of evaluating an aggregate;

12

- a component, slice, or view conversion of another object.

13

An object is either a `<constant>` object or a `<variable>` object. The value of a constant object cannot be changed between its initialization and its finalization, whereas the value of a variable object can be changed. Similarly, a view of an object is either a `<constant>` or a `<variable>`. All views of a constant object are constant. A constant view of a variable object cannot be used to modify the value of the variable. The terms `constant` and `variable` by themselves refer to constant and variable views of objects.

14

The value of an object is `<read>` when the value of any part of the object is evaluated, or when the value of an enclosing object is evaluated. The value of a variable is `<updated>` when an assignment is performed to any part of the variable, or when an assignment is performed to an enclosing object.

15

Whether a view of an object is constant or variable is determined by the definition of the view. The following (and no others) represent constants:

16

- an object declared by an `object_declaration` with the reserved word `constant`;

17

- a formal parameter or generic formal object of mode `in`;

18

- a discriminant;

19

- a loop parameter, choice parameter, or entry index;

20

- the dereference of an access-to-constant value;

21

- the result of evaluating a function_call or an aggregate;

22

- a selected_component, indexed_component, slice, or view conversion of a constant.

23

At the place where a view of an object is defined, a <nominal subtype> is associated with the view. The object's <actual subtype> (that is, its subtype) can be more restrictive than the nominal subtype of the view; it always is if the nominal subtype is an <indefinite subtype>. A subtype is an indefinite subtype if it is an unconstrained array subtype, or if it has unknown discriminants or unconstrained discriminants without defaults (see Section 4.7 [3.7], page 123); otherwise the subtype is a <definite> subtype (all elementary subtypes are definite subtypes). A class-wide subtype is defined to have unknown discriminants, and is therefore an indefinite subtype. An indefinite subtype does not by itself provide enough information to create an object; an additional constraint or explicit initialization expression is necessary (see Section 4.3.1 [3.3.1], page 61). A component cannot have an indefinite nominal subtype.

24

A <named number> provides a name for a numeric value known at compile time. It is declared by a number_declaration.

NOTES

25

5 A constant cannot be the target of an assignment operation, nor be passed as an in out or out parameter, between its initialization and finalization, if any.

26

6 The nominal and actual subtypes of an elementary object are always the same. For a discriminated or array object, if the nominal subtype is constrained then so is the actual subtype.

4.3.1 3.3.1 Object Declarations

1

An `object_declaration` declares a <stand-alone> object with a given nominal subtype and, optionally, an explicit initial value given by an initialization expression. For an array, task, or protected object, the `object_declaration` may include the definition of the (anonymous) type of the object.

Syntax

2/2

```
object_declaration ::=
    defining_identifier_list : [aliased] [constant] subtype_indication [:= expression];
    | defining_identifier_list : [aliased] [constant] access_definition [:= expression];
    | defining_identifier_list : [aliased] [constant] array_type_definition [:= expression];
    | single_task_declaration
    | single_protected_declaration
```

3

```
defining_identifier_list ::=
    defining_identifier {, defining_identifier}
    Name Resolution Rules
```

4

For an `object_declaration` with an expression following the compound delimiter `:=`, the type expected for the expression is that of the object. This expression is called the <initialization expression>.

Legality Rules

5/2

An `object_declaration` without the reserved word `constant` declares a variable object. If it has a `subtype_indication` or an `array_type_definition` that defines an indefinite subtype, then there shall be an initialization expression.

Static Semantics

6

An `object_declaration` with the reserved word `constant` declares a constant object. If it has an initialization expression, then it is called a <full constant declaration>. Otherwise it is called a <deferred constant declaration>. The rules for deferred constant declarations are given in clause Section 8.4 [7.4], page 290. The rules for full constant declarations are given in this subclause.

7

Any declaration that includes a `defining_identifier_list` with more than one `defining_identifier` is equivalent to a series of declarations each containing one `defining_identifier` from the list, with the rest of the text of the declaration copied for each declaration in the series, in the same order as the list. The remainder of this International Standard relies on this equivalence; explanations are given for declarations with a single `defining_identifier`.

8/2

The `subtype_indication`, `access_definition`, or full type definition of an `object_declaration` defines the nominal subtype of the object. The `object_declaration` declares an object of the type of the nominal subtype.

8.1/2

A component of an object is said to `<require late initialization>` if it has an access discriminant value constrained by a `per-object` expression, or if it has an initialization expression that includes a name denoting the current instance of the type or denoting an access discriminant.

Dynamic Semantics

9/2

If a composite object declared by an `object_declaration` has an unconstrained nominal subtype, then if this subtype is indefinite or the object is constant the actual subtype of this object is constrained. The constraint is determined by the bounds or discriminants (if any) of its initial value; the object is said to be `<constrained by its initial value>`. When not constrained by its initial value, the actual and nominal subtypes of the object are the same. If its actual subtype is constrained, the object is called a `<constrained object>`.

10

For an `object_declaration` without an initialization expression, any initial values for the object or its subcomponents are determined by the `<implicit initial values>` defined for its nominal subtype, as follows:

11

- The implicit initial value for an access subtype is the null value of the access type.

12

- The implicit initial (and only) value for each discriminant of a constrained discriminated subtype is defined by the subtype.

13

- For a (definite) composite subtype, the implicit initial value of each component with a `default_expression` is obtained by evaluation of this expression and conversion to the component's nominal subtype (which might raise `Constraint_Error` — see Section 5.6 [4.6], page 219, "Section 5.6 [4.6], page 219, Type Conversions"), unless the component is a discriminant of a constrained subtype (the previous case), or is in an excluded variant (see Section 4.8.1 [3.8.1], page 134). For each component that does not have a `default_expression`, any implicit initial values are those determined by the component's nominal subtype.

14

- For a protected or task subtype, there is an implicit component (an entry queue) corresponding to each entry, with its implicit initial value being an empty queue.

15

The elaboration of an `object_declaration` proceeds in the following sequence of steps:

16/2

1. The `subtype_indication` (see [S0027], page 56), `access_definition` (see [S0084], page 156), `array_type_definition` (see [S0051], page 114), `single_task_declaration` (see [S0189], page 329), or `single_protected_declaration` (see [S0194], page 338) is first elaborated. This creates the nominal subtype (and the anonymous type in the last four cases).

17

2. If the `object_declaration` includes an initialization expression, the (explicit) initial value is obtained by evaluating the expression and converting it to the nominal subtype (which might raise `Constraint_Error` — see Section 5.6 [4.6], page 219).

18/2

3. The object is created, and, if there is not an initialization expression, the object is <initialized by default>. When an object is initialized by default, any per-object constraints (see Section 4.8 [3.8], page 130) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype. Any initial values (whether explicit or implicit) are assigned to the object or to the corresponding subcomponents. As described in Section 6.2 [5.2], page 242, and Section 8.6 [7.6], page 295, `Initialize` and `Adjust` procedures can be called.

19/2

<This paragraph was deleted.>

20/2

For the third step above, evaluations and assignments are performed in an arbitrary order subject to the following restrictions:

20.1/2

- Assignment to any part of the object is preceded by the evaluation of the value that is to be assigned.

20.2/2

- The evaluation of a `default_expression` that includes the name of a discriminant is preceded by the assignment to that discriminant.

20.3/2

- The evaluation of the `default_expression` for any component that depends on a discriminant is preceded by the assignment to that discriminant.

20.4/2

- The assignments to any components, including implicit components, not requiring late initialization must precede the initial value evaluations for any components requiring

late initialization; if two components both require late initialization, then assignments to parts of the component occurring earlier in the order of the component declarations must precede the initial value evaluations of the component occurring later.

21

There is no implicit initial value defined for a scalar subtype. In the absence of an explicit initialization, a newly created scalar object might have a value that does not belong to its subtype (see Section 14.9.1 [13.9.1], page 522, and Section 22.1 [H.1], page 1153).

NOTES

22

7 Implicit initial values are not defined for an indefinite subtype, because if an object's nominal subtype is indefinite, an explicit initial value is required.

23

8 As indicated above, a stand-alone object is an object declared by an `object_declaration`. Similar definitions apply to "stand-alone constant" and "stand-alone variable." A subcomponent of an object is not a stand-alone object, nor is an object that is created by an allocator. An object declared by a `loop_parameter_specification`, `parameter_specification`, `entry_index_specification`, `choice_parameter_specification`, or a `formal_object_declaration` is not called a stand-alone object.

24

9 The type of a stand-alone object cannot be abstract (see Section 4.9.3 [3.9.3], page 149).

Examples

25

<Example of a multiple object declaration:>

26

```
--< the multiple object declaration >
```

27/2

```
John, Paul : not null Person_Name := new Person(Sex => M); --< see Section 4.10  
[3.10.1], page 160>
```

28

```
--< is equivalent to the two single object declarations in the order given>■
```

29/2

```
John : not null Person_Name := new Person(Sex => M);
```



```
Paul : not null Person_Name := new Person(Sex => M);
```

30

<Examples of variable declarations:>

31/2

```
Count, Sum : Integer;
Size       : Integer range 0 .. 10_000 := 0;
Sorted     : Boolean := False;
Color_Table : array(1 .. Max) of Color;
Option     : Bit_Vector(1 .. 10) := (others => True);
Hello      : aliased String := "Hi, world.";
[Unicode 952], [Unicode 966] : Float range -PI .. +PI;
```

32

<Examples of constant declarations:>

33/2

```
Limit      : constant Integer := 10_000;
Low_Limit  : constant Integer := Limit/10;
Tolerance  : constant Real := Dispersion(1.15);
Hello_Msg  : constant access String := Hello'Access; --< see Section 4.10.2
[3.10.2], page 164>
```

4.3.2 3.3.2 Number Declarations

1

A number_declaration declares a named number.

Syntax

2

```
number_declaration ::=
    defining_identifier_list : constant := <static_>expression;
Name Resolution Rules
```

3

The <static_>expression given for a number_declaration is expected to be of any numeric type.

Legality Rules

4

The <static_>expression given for a number declaration shall be a static expression, as defined by clause Section 5.9 [4.9], page 234.

Static Semantics

5

The named number denotes a value of type <universal_integer> if the type of the <static_>-expression is an integer type. The named number denotes a value of type <universal_real> if the type of the <static_>expression is a real type.

6

The value denoted by the named number is the value of the <static_>expression, converted to the corresponding universal type.

Dynamic Semantics

7

The elaboration of a number_declaration has no effect.

Examples

8

<Examples of number declarations:>

9

```
Two_Pi      : constant := 2.0*Ada.Numerics.Pi;  --< a real number (see Section
[A.5], page 648)>
```

10/2

```
Max          : constant := 500;                --< an integer number>■
Max_Line_Size : constant := Max/6;            --< the integer 83>■
Power_16     : constant := 2**16;            --< the integer 65_536>■
One, Un, Eins : constant := 1;                --< three different names for
```

4.4 3.4 Derived Types and Classes

1/2

A derived_type_definition defines a <derived type> (and its first subtype) whose characteristics are <derived> from those of a parent type, and possibly from progenitor types.

1.1/2

A <class of types> is a set of types that is closed under derivation; that is, if the parent or a progenitor type of a derived type belongs to a class, then so does the derived type. By saying that a particular group of types forms a class, we are saying that all derivatives of a type in the set inherit the characteristics that define that set. The more general term <category of types> is used for a set of types whose defining characteristics are not necessarily inherited by derivatives; for example, limited, abstract, and interface are all categories of types, but not classes of types.

Syntax

2/2

```
derived_type_definition ::=
  [abstract] [limited] new <parent_>subtype_indication [[and interface_list] record_extension_part]
```

Legality Rules

3/2

The <parent_>subtype_indication defines the <parent subtype>; its type is the <parent type>. The interface_list defines the progenitor types (see Section 4.9.4 [3.9.4], page 152). A derived type has one parent type and zero or more progenitor types.

4

A type shall be completely defined (see Section 4.11.1 [3.11.1], page 177) prior to being specified as the parent type in a `derived_type_definition` — the `full_type_declarations` for the parent type and any of its subcomponents have to precede the `derived_type_definition`.

5/2

If there is a `record_extension_part`, the derived type is called a <record extension> of the parent type. A `record_extension_part` shall be provided if and only if the parent type is a tagged type. An `interface_list` shall be provided only if the parent type is a tagged type.

5.1/2

If the reserved word `limited` appears in a `derived_type_definition`, the parent type shall be a limited type.

Static Semantics

6

The first subtype of the derived type is unconstrained if a `known_discriminant_part` is provided in the declaration of the derived type, or if the parent subtype is unconstrained. Otherwise, the constraint of the first subtype <corresponds> to that of the parent subtype in the following sense: it is the same as that of the parent subtype except that for a range constraint (implicit or explicit), the value of each bound of its range is replaced by the corresponding value of the derived type.

6.1/2

The first subtype of the derived type excludes null (see Section 4.10 [3.10], page 156) if and only if the parent subtype excludes null.

7

The characteristics of the derived type are defined as follows:

8/2

- If the parent type or a progenitor type belongs to a class of types, then the derived type also belongs to that class. The following sets of types, as well as any higher-level sets composed from them, are classes in this sense, and hence the characteristics defining these classes are inherited by derived types from their parent or progenitor types: signed integer, modular integer, ordinary fixed, decimal fixed, floating point, enumeration, boolean, character, `access-to-constant`, `general access-to-variable`, `pool-specific access-to-variable`, `access-to-subprogram`, array, string, non-array composite, non-limited, untagged record, tagged, task, protected, and synchronized tagged.

9

- If the parent type is an elementary type or an array type, then the set of possible values of the derived type is a copy of the set of possible values of the parent type. For a scalar type, the base range of the derived type is the same as that of the parent type.

10

- If the parent type is a composite type other than an array type, then the components, protected subprograms, and entries that are declared for the derived type are as follows:

11

- The discriminants specified by a new `known_discriminant_part`, if there is one; otherwise, each discriminant of the parent type (implicitly declared in the same order with the same specifications) — in the latter case, the discriminants are said to be `<inherited>`, or if unknown in the parent, are also unknown in the derived type;

12

- Each nondiscriminant component, entry, and protected subprogram of the parent type, implicitly declared in the same order with the same declarations; these components, entries, and protected subprograms are said to be `<inherited>`;

13

- Each component declared in a `record_extension_part`, if any.

14

Declarations of components, protected subprograms, and entries, whether implicit or explicit, occur immediately within the declarative region of the type, in the order indicated above, following the `parent_subtype_indication`.

15/2

- `<This paragraph was deleted.>`

16

- For each predefined operator of the parent type, there is a corresponding predefined operator of the derived type.

17/2

- For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent type or of a progenitor type that already exists at the place of the `derived_type_definition`, there exists a corresponding `<inherited>` primitive subprogram of the derived type with the same defining name. Primitive user-defined equality operators of the parent type and any progenitor types are also inherited by

the derived type, except when the derived type is a nonlimited record extension, and the inherited operator would have a profile that is type conformant with the profile of the corresponding predefined equality operator; in this case, the user-defined equality operator is not inherited, but is rather incorporated into the implementation of the predefined equality operator of the record extension (see Section 5.5.2 [4.5.2], page 206).

18/2

The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent or progenitor type, after systematic replacement of each subtype of its profile (see Section 7.1 [6.1], page 255) that is of the parent or progenitor type with a <corresponding subtype> of the derived type. For a given subtype of the parent or progenitor type, the corresponding subtype of the derived type is defined as follows:

19

- If the declaration of the derived type has neither a `known_discriminant_part` nor a `record_extension_part`, then the corresponding subtype has a constraint that corresponds (as defined above for the first subtype of the derived type) to that of the given subtype.

20

- If the derived type is a record extension, then the corresponding subtype is the first subtype of the derived type.

21

- If the derived type has a new `known_discriminant_part` but is not a record extension, then the corresponding subtype is constrained to those values that when converted to the parent type belong to the given subtype (see Section 5.6 [4.6], page 219).

22/2

The same formal parameters have `default_expressions` in the profile of the inherited subprogram. Any type mismatch due to the systematic replacement of the parent or progenitor type by the derived

type is handled as part of the normal type conversion associated with parameter passing — see Section 7.4.1 [6.4.1], page 270.

23/2

If a primitive subprogram of the parent or progenitor type is visible at the place of the `derived_type_definition`, then the corresponding inherited subprogram is implicitly declared immediately after the `derived_type_definition`. Otherwise, the inherited subprogram is implicitly declared later or not at all, as explained in Section 8.3.1 [7.3.1], page 287.

24

A derived type can also be defined by a `private_extension_declaration` (see [S0177], page 283) (see Section 8.3 [7.3], page 283) or a `formal_derived_type_definition` (see [S0265], page 463) (see Section 13.5.1 [12.5.1], page 462). Such a derived type is a partial view of the corresponding full or actual type.

25

All numeric types are derived types, in that they are implicitly derived from a corresponding root numeric type (see Section 4.5.4 [3.5.4], page 95, and Section 4.5.6 [3.5.6], page 102).

Dynamic Semantics

26

The elaboration of a `derived_type_definition` creates the derived type and its first subtype, and consists of the elaboration of the `subtype_indication` (see [S0027], page 56) and the `record_extension_part` (see [S0075], page 143), if any. If the `subtype_indication` (see [S0027], page 56) depends on a discriminant, then only those expressions that do not depend on a discriminant are evaluated.

27/2

For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent or progenitor type is performed; the normal conversion of each actual parameter to the subtype of the corresponding formal parameter (see Section 7.4.1 [6.4.1], page 270) performs any necessary type conversion as well. If the result type of the inherited subprogram is the derived type, the result of calling the subprogram of the parent or progenitor is converted to the derived type, or in the case of a null extension, extended to the derived type using the equivalent of an `extension_aggregate` with the original result as the `ancestor_part` and null record as the `record_component_association_list`.

NOTES

28

10 Classes are closed under derivation — any class that contains a type also contains its derivatives. Operations available for a given class of types are available for the derived types in that class.

29

11 Evaluating an inherited enumeration literal is equivalent to evaluating the corresponding enumeration literal of the parent type, and then converting the result to the derived type. This follows from their equivalence to parameterless functions.

30

12 A generic subprogram is not a subprogram, and hence cannot be a primitive subprogram and cannot be inherited by a derived type. On the other hand, an instance of a generic subprogram can be a primitive subprogram, and hence can be inherited.

31

13 If the parent type is an access type, then the parent and the derived type share the same storage pool; there is a null access value for the derived type and it is the implicit initial value for the type. See Section 4.10 [3.10], page 156.

32

14 If the parent type is a boolean type, the predefined relational operators of the derived type deliver a result of the predefined type Boolean (see Section 5.5.2 [4.5.2], page 206). If the parent type is an integer type, the right operand of the predefined exponentiation operator is of the predefined type Integer (see Section 5.5.6 [4.5.6], page 217).

33

15 Any discriminants of the parent type are either all inherited, or completely replaced with a new set of discriminants.

34

16 For an inherited subprogram, the subtype of a formal parameter of the derived type need not have any value in common with the first subtype of the derived type.

35

17 If the reserved word `abstract` is given in the declaration of a type, the type is abstract (see Section 4.9.3 [3.9.3], page 149).

35.1/2

18 An interface type that has a progenitor type "is derived from" that type. A `derived_type_definition`, however, never defines an interface type.

35.2/2

19 It is illegal for the parent type of a `derived_type_definition` to be a synchronized tagged type.

Examples

36

<Examples of derived type declarations:>

37

```
type Local_Coordinate is new Coordinate;    --< two different types>■
type Midweek is new Day range Tue .. Thu;  --< see Section 4.5.1
[3.5.1], page 92>
type Counter is new Positive;              --< same range as Positive >■
```

38

```
type Special_Key is new Key_Manager.Key;   --< see Section 8.3.1
[7.3.1], page 287>
--< the inherited subprograms have the following specifications: >■
--<     procedure Get_Key(K : out Special_Key);>
--<     function "<(X,Y : Special_Key) return Boolean;>
```

4.4.1 3.4.1 Derivation Classes

1

In addition to the various language–defined classes of types, types can be grouped into <derivation classes>.

Static Semantics

2/2

A derived type is <derived from> its parent type <directly>; it is derived <indirectly> from any type from which its parent type is derived. A derived type, interface type, type extension, task type, protected type, or formal derived type is also derived from every ancestor of each of its progenitor types, if any. The derivation class of types for a type <T> (also called the class <rooted> at <T>) is the set consisting of <T> (the <root type> of the class) and all types derived from <T> (directly or indirectly) plus any associated universal or class–wide types (defined below).

3/2

Every type is either a <specific> type, a <class–wide> type, or a <universal> type. A specific type is one defined by a type–declaration, a formal–type–declaration, or a full type definition embedded in another construct. Class–wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

4

Class–wide types

Class–wide types are defined for (and belong to) each derivation class rooted at a tagged type (see Section 4.9 [3.9], page 136). Given a subtype S of a tagged type <T>, S'Class is the subtype_mark for

a corresponding subtype of the tagged class-wide type `<T>'Class`. Such types are called "class-wide" because when a formal parameter is defined to be of a class-wide type `<T>'Class`, an actual parameter of any type in the derivation class rooted at `<T>` is acceptable (see Section 9.6 [8.6], page 324).

5

The set of values for a class-wide type `<T>'Class` is the discriminated union of the set of values of each specific type in the derivation class rooted at `<T>` (the tag acts as the implicit discriminant -- see Section 4.9 [3.9], page 136). Class-wide types have no primitive subprograms of their own. However, as explained in Section 4.9.2 [3.9.2], page 145, operands of a class-wide type `<T>'Class` can be used as part of a dispatching call on a primitive subprogram of the type `<T>`. The only components (including discrimi-

nants) of `<T>'Class` that are visible are those of `<T>`. If `S` is a first subtype, then `S'Class` is a first subtype.

6/2

Universal types

Universal types are defined for (and belong to) the integer, real, fixed point, and access classes, and are referred to in this standard as `integer`, `real`, `fixed`, and `access` respectively, `<universal_integer>`, `<universal_real>`, `<universal_fixed>`, and `<universal_access>`. These are analogous to class-wide types for these language-defined elementary classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the corresponding class is acceptable. In addition, a value of a universal type (including an integer or real `numeric_literal`, or the literal `null`) is "universal" in that it is acceptable where some particular type in the class is expected (see

Section 9.6 [8.6],
page 324).

7

The set of values of a universal type is the undiscriminated union of the set of values possible for any definable type in the associated class. Like class-wide types, universal types have no primitive subprograms of their own. However, their "universality" allows them to be used as operands with the primitive subprograms of any type in the corresponding class.

8

The integer and real numeric classes each have a specific root type in addition to their universal type, named respectively <root_integer> and <root_real>.

9

A class-wide or universal type is said to <cover> all of the types in its class. A specific type covers only itself.

10/2

A specific type <T2> is defined to be a <descendant> of a type <T1> if <T2> is the same as <T1>, or if <T2> is derived (directly or indirectly) from <T1>. A class-wide type <T2>'Class is defined to be a descendant of type <T1> if <T2> is a descendant of <T1>. Similarly, the numeric universal types are defined to be descendants of the root types of their classes. If a type <T2> is a descendant of a type <T1>, then <T1> is called an <ancestor> of <T2>. An <ultimate ancestor> of a type is an ancestor of that type that is not itself a descendant of any other type. Every untagged type has a unique ultimate ancestor.

11

An inherited component (including an inherited discriminant) of a derived type is inherited <from> a given ancestor of the type if the corresponding component was inherited by each derived type in the chain of derivations going back to the given ancestor.

NOTES

12

20 Because operands of a universal type are acceptable to the predefined operators of any type in their class, ambiguity can result. For `<universal_integer>` and `<universal_real>`, this potential ambiguity is resolved by giving a preference (see Section 9.6 [8.6], page 324) to the predefined operators of the corresponding root types (`<root_integer>` and `<root_real>`, respectively). Hence, in an apparently ambiguous expression like

13

$$1 + 4 < 7$$

14

where each of the literals is of type `<universal_integer>`, the predefined operators of `<root_integer>` will be preferred over those of other specific integer types, thereby resolving the ambiguity.

4.5 3.5 Scalar Types

1

`<Scalar>` types comprise enumeration types, integer types, and real types. Enumeration types and integer types are called `<discrete>` types; each value of a discrete type has a `<position number>` which is an integer value. Integer types and real types are called `<numeric>` types. All scalar types are ordered, that is, all relational operators are predefined for their values.

Syntax

2

```
range_constraint ::= range range
```

3

```
range ::= range_attribute_reference  
       | simple_expression .. simple_expression
```

4

A `<range>` has a `<lower bound>` and an `<upper bound>` and specifies a subset of the values of some scalar type (the `<type of the range>`). A range with lower bound L and upper bound R is described by "L .. R". If R is less than L, then the range is a `<null range>`, and specifies an empty set of values. Otherwise, the range specifies the values of the type from the lower bound to the upper bound, inclusive. A value `<belongs>` to a range if it is of the type of the range, and is in the subset of values specified by the range. A value `<satisfies>` a range constraint if it belongs to the associated range. One range is `<included>` in another if all values that belong to the first range also belong to the second.

Name Resolution Rules

5

For a `subtype_indication` containing a `range_constraint`, either directly or as part of some other `scalar_constraint`, the type of the range shall resolve to that of the type determined

by the `subtype_mark` of the `subtype_indication`. For a range of a given type, the `simple_expressions` of the range (likewise, the `simple_expressions` of the equivalent range for a `range_attribute_reference`) are expected to be of the type of the range.

Static Semantics

6

The `<base range>` of a scalar type is the range of finite values of the type that can be represented in every unconstrained object of the type; it is also the range supported at a minimum for intermediate values during the evaluation of expressions involving predefined operators of the type.

7

A constrained scalar subtype is one to which a range constraint applies. The `<range>` of a constrained scalar subtype is the range associated with the range constraint of the subtype. The `<range>` of an unconstrained scalar subtype is the base range of its type.

Dynamic Semantics

8

A range is `<compatible>` with a scalar subtype if and only if it is either a null range or each bound of the range belongs to the range of the subtype. A `range_constraint` is `<compatible>` with a scalar subtype if and only if its range is compatible with the subtype.

9

The elaboration of a `range_constraint` consists of the evaluation of the range. The evaluation of a range determines a lower bound and an upper bound. If `simple_expressions` are given to specify bounds, the evaluation of the range evaluates these `simple_expressions` in an arbitrary order, and converts them to the type of the range. If a `range_attribute_reference` is given, the evaluation of the range consists of the evaluation of the `range_attribute_reference`.

10

`<Attributes>`

11

For every scalar subtype S, the following attributes are defined:

12

S'First

S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S.

13

S'Last

S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S.

14

S'Range

S'Range is equivalent to the range S'First .. S'Last.

15
S'Base

S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the <base subtype> of the type.

16
S'Min

S'Min denotes a function with the following specification:

17

```
function S'Min(<Left>, <Right> : S'Base)
return S'Base
```

18

The function returns the lesser of the values of the two parameters.

19
S'Max

S'Max denotes a function with the following specification:

20

```
function S'Max(<Left>, <Right> : S'Base)
return S'Base
```

21

The function returns the greater of the values of the two parameters.

22

S'Succ

S'Succ denotes a function with the following specification:

23

```
function S'Succ(<Arg> : S'Base)
  return S'Base
```

24

For an enumeration type, the function returns the value whose position number is one more than that of the value of <Arg>; `Constraint_Error` is raised if there is no such value of the type. For an integer type, the function returns the result of adding one to the value of <Arg>. For a fixed point type, the function returns the result of adding <small> to the value of <Arg>. For a floating point type, the function returns the machine number (as defined in Section 4.5.7 [3.5.7], page 103) immediately above the value of <Arg>; `Constraint_Error` is raised if there is no such machine number.

25

S'Pred

S'Pred denotes a function with the following specification:

26

```
function S'Pred(<Arg> : S'Base)
return S'Base
```

27

For an enumeration type, the function returns the value whose position number is one less than that of the value of <Arg>; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of <Arg>. For a fixed point type, the function returns the result of subtracting <small> from the value of <Arg>. For a floating point type, the function returns the machine number (as defined in Section 4.5.7 [3.5.7], page 103) immediately below the value of <Arg>; Constraint_Error is raised if there is no such machine number.

27.1/2
S'Wide_Wide_Image

S'Wide_Wide_Image
denotes a function
with the following
specification:

27.2/2

```
function S'Wide_Wide_Image(<Arg> : S'Base)■  
    return Wide_Wide_String
```

27.3/2

The function returns
an <image> of the
value of <Arg>,
that is, a sequence
of characters
representing the value
in display form. The
lower bound of the
result is one.

27.4/2

The image of an
integer value is
the corresponding
decimal literal,
without underlines,
leading zeros,
exponent, or trailing
spaces, but with
a single leading
character that is
either a minus sign or
a space.

27.5/2

The image of an
enumeration value
is either the corre-
sponding identifier
in upper case or
the corresponding
character literal
(including the two
apostrophes); neither
leading nor trailing

spaces are included. For a <nongraphic character> (a value of a character type that has no enumeration literal associated with it), the result is a corresponding language-defined name in upper case (for example, the image of the nongraphic character identified as <nul> is "NUL" -- the quotes are not part of the image).

27.6/2

The image of a floating point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, a single digit (that is nonzero unless the value is zero), a decimal point, S'Digits-1 (see Section 4.5.8 [3.5.8], page 105) digits after the decimal point (but one if S'Digits is one), an upper case E, the sign of the exponent (either + or -), and two or more digits (with leading zeros if necessary) representing the exponent. If

27.7/2

S'Signed_Zeros is True, then the leading character is a minus sign for a negatively signed zero.

The image of a fixed point value is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, one or more digits before the decimal point (with no redundant leading zeros), a decimal point, and S'Aft (see Section 4.5.10 [3.5.10], page 109) digits after the decimal point.

28
S'Wide_Image

S'Wide_Image denotes a function with the following specification:

29

```
function S'Wide_Image(<Arg> : S'Base)
  return Wide_String
```

30/2

The function returns an image of the value of <Arg> as a Wide_String. The lower bound of the result is one. The

image has the same sequence of character as defined for S'Wide_Wide_Image if all the graphic characters are defined in Wide_Character; otherwise the sequence of characters is implementation defined (but no shorter than that of S'Wide_Wide_Image for the same value of Arg).

<Paragraphs 31 through 34 were moved to Wide_Wide_Image.>

35
S'Image

S'Image denotes a function with the following specification:

36

```
function S'Image(<Arg> : S'Base)
    return String
```

37/2

The function returns an image of the value of <Arg> as a String. The lower bound of the result is one. The image has the same sequence of graphic characters as that defined for S'Wide_Wide_Image if all the graphic characters are defined in Character; otherwise the

sequence of characters
is implementation
defined (but no
shorter than that of
S'Wide_Wide_Image
for the same value of
<Arg>).

37.1/2
S'Wide_Wide_Width

S'Wide_Wide_Width
denotes the
maximum length of
a Wide_Wide_String
returned by
S'Wide_Wide_Image
over all values of
the subtype S. It
denotes zero for a
subtype that has a
null range. Its type is
<universal_integer>.

38
S'Wide_Width

S'Wide_Width
denotes the
maximum length of a
Wide_String returned
by S'Wide_Image
over all values of
the subtype S. It
denotes zero for a
subtype that has a
null range. Its type is
<universal_integer>.

39
S'Width

S'Width denotes the
maximum length of
a String returned
by S'Image over all
values of the subtype
S. It denotes zero for
a subtype that has a
null range. Its type is
<universal_integer>.

39.1/2
S'Wide_Wide_Value

S'Wide_Wide_Value
denotes a function
with the following
specification:

39.2/2

```
function S'Wide_Wide_Value(<Arg> : Wide_Wide_String)
  return S'Base
```

39.3/2

This function returns
a value given an
image of the value as
a Wide_Wide_String,
ignoring any leading
or trailing spaces.

39.4/2

For the evaluation
of a call on
S'Wide_Wide_Value
for an enumeration
subtype S, if the
sequence of characters
of the parameter
(ignoring leading
and trailing spaces)
has the syntax of an
enumeration literal
and if it corresponds
to a literal of the type
of S (or corresponds
to the result of
S'Wide_Wide_Image
for a nongraphic
character of the
type), the result is
the corresponding
enumeration
value; otherwise
Constraint_Error is
raised.

39.5/2

For the evaluation of a call on `S'Wide_Wide_Value` for an integer subtype `S`, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an integer literal, with an optional leading sign character (plus or minus for a signed type; only plus for a modular type), and the corresponding numeric value belongs to the base range of the type of `S`, then that value is the result; otherwise `Constraint_Error` is raised.

39.6/2

For the evaluation of a call on `S'Wide_Wide_Value` for a real subtype `S`, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of one of the following:

39.7/2

- `numeric_literal`

39.8/2

- `numeral.[exponent]`



39.9/2

- `.numeral[exponent]` ■

39.10/2

- `base#based_numeral.#[exponent]` ■

39.11/2

- `base#.based_numeral#[exponent]` ■

39.12/2

with an optional leading sign character (plus or minus), and if the corresponding numeric value belongs to the base range of the type of `S`, then that value is the result; otherwise `Constraint_Error` is raised. The sign of a zero value is preserved (positive if none has been specified) if `S'Signed_Zeros` is `True`.

40

`S'Wide_Value`

`S'Wide_Value` denotes a function with the following specification:

41

```
function S'Wide_Value(<Arg> : Wide_String) ■  
    return S'Base
```

42

This function returns a value given an image of the value as a `Wide_String`, ignoring

any leading or trailing spaces.

43/2

For the evaluation of a call on `S'Wide_Value` for an enumeration subtype `S`, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of `S` (or corresponds to the result of `S'Wide_Image` for a value of the type), the result is the corresponding enumeration value; otherwise `Constraint_Error` is raised. For a numeric subtype `S`, the evaluation of a call on `S'Wide_Value` with `<Arg>` of type `Wide_String` is equivalent to a call on `S'Wide_Wide_Value` for a corresponding `<Arg>` of type `Wide_Wide_String`.

<Paragraphs 44 through 51 were moved to `Wide_Wide_Value`.>

52
`S'Value`

`S'Value` denotes a function with

the following
specification:

53

```
function S'Value(<Arg> : String)
    return S'Base
```

54

This function returns
a value given an
image of the value as
a String, ignoring any
leading or trailing
spaces.

55/2

For the evaluation
of a call on S'Value
for an enumeration
subtype S, if the
sequence of characters
of the parameter
(ignoring leading
and trailing spaces)
has the syntax of an
enumeration literal
and if it corresponds
to a literal of the type
of S (or corresponds
to the result of
S'Image for a value of
the type), the result
is the corresponding
enumeration
value; otherwise
Constraint_Error is
raised. For a numeric
subtype S, the
evaluation of a call on
S'Value with <Arg>
of type String is
equivalent to a call on
S'Wide_Wide_Value
for a corresponding
<Arg> of type
Wide_Wide_String.

Implementation Permissions

56/2

An implementation may extend the `Wide_Wide_Value`, `Wide_Value`, `Value`, `Wide_Wide_Image`, `Wide_Image`, and `Image` attributes of a floating point type to support special values such as infinities and NaNs.

NOTES

57

21 The evaluation of `S'First` or `S'Last` never raises an exception. If a scalar subtype `S` has a nonnull range, `S'First` and `S'Last` belong to this range. These values can, for example, always be assigned to a variable of subtype `S`.

58

22 For a subtype of a scalar type, the result delivered by the attributes `Succ`, `Pred`, and `Value` might not belong to the subtype; similarly, the actual parameters of the attributes `Succ`, `Pred`, and `Image` need not belong to the subtype.

59

23 For any value `V` (including any nongraphic character) of an enumeration subtype `S`, `S'Value(S'Image(V))` equals `V`, as do `S'Wide_Value(S'Wide_Image(V))` and `S'Wide_Wide_Value(S'Wide_Wide_Image(V))`. None of these expressions ever raise `Constraint_Error`.

Examples

60

<Examples of ranges:>

61

```
-10 .. 10
X .. X + 1
0.0 .. 2.0*Pi
Red .. Green    --< see Section 4.5.1 [3.5.1], page 92>
1 .. 0          --< a null range>
Table'Range     --< a range attribute reference (see Section 4.6
[3.6], page 114)>
```

62

<Examples of range constraints:>

63

```
range -999.0 .. +999.0
range S'First+1 .. S'Last-1
```

4.5.1 3.5.1 Enumeration Types

1

An enumeration_type_definition defines an enumeration type.

Syntax

2

```
enumeration_type_definition ::=  
    (enumeration_literal_specification {, enumeration_literal_specification})
```

3

```
enumeration_literal_specification ::= defining_identifier | defining_character_literal
```

4

```
defining_character_literal ::= character_literal  
Legality Rules
```

5

The defining_identifiers and defining_character_literals listed in an enumeration_type_definition shall be distinct.

Static Semantics

6

Each enumeration_literal_specification is the explicit declaration of the corresponding <enumeration literal>: it declares a parameterless function, whose defining name is the defining_identifier (see [S0022], page 49) or defining_character_literal (see [S0040], page 92), and whose result type is the enumeration type.

7

Each enumeration literal corresponds to a distinct value of the enumeration type, and to a distinct position number. The position number of the value of the first listed enumeration literal is zero; the position number of the value of each subsequent enumeration literal is one more than that of its predecessor in the list.

8

The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.

9

If the same defining_identifier or defining_character_literal is specified in more than one enumeration_type_definition (see [S0038], page 92), the corresponding enumeration literals are said to be <overloaded>. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal has to be determinable from the context (see Section 9.6 [8.6], page 324).

Dynamic Semantics

10

The elaboration of an enumeration_type_definition creates the enumeration type and its first subtype, which is constrained to the base range of the type.

11

When called, the parameterless function associated with an enumeration literal returns the corresponding value of the enumeration type.

NOTES

12

24 If an enumeration literal occurs in a context that does not otherwise suffice to determine the type of the literal, then qualification by the name of the enumeration type is one way to resolve the ambiguity (see Section 5.7 [4.7], page 229).

Examples

13

<Examples of enumeration types and subtypes: >

14

```
type Day    is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Suit   is (Clubs, Diamonds, Hearts, Spades);
type Gender is (M, F);
type Level  is (Low, Medium, Urgent);
type Color  is (White, Red, Yellow, Green, Blue, Brown, Black);
type Light  is (Red, Amber, Green); --< Red and Green are overloaded>
```

15

```
type Hexa   is ('A', 'B', 'C', 'D', 'E', 'F');
type Mixed  is ('A', 'B', '*', B, None, '?', '%');
```

16

```
subtype Weekday is Day    range Mon .. Fri;
subtype Major   is Suit   range Hearts .. Spades;
subtype Rainbow is Color range Red .. Blue; --< the Color Red, not the Light>
```

4.5.2 3.5.2 Character Types

Static Semantics

1

An enumeration type is said to be a <character type> if at least one of its enumeration literals is a `character_literal`.

2/2

The predefined type `Character` is a character type whose values correspond to the 256 code positions of Row 00 (also known as Latin-1) of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of Row 00 of the BMP has a corresponding `character_literal` in `Character`. Each of the nongraphic positions of Row 00 (0000–001F and 007F–009F) has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes `Image`, `Wide_Image`, `Wide_Wide_Image`, `Value`, `Wide_Value`, and `Wide_Wide_Value`; these names are given in

the definition of type `Character` in Section 15.1 [A.1], page 556, "Section 15.1 [A.1], page 556, The Package Standard", but are set in <italics>.

3/2

The predefined type `Wide_Character` is a character type whose values correspond to the 65536 code positions of the ISO/IEC 10646:2003 Basic Multilingual Plane (BMP). Each of the graphic characters of the BMP has a corresponding `character_literal` in `Wide_Character`. The first 256 values of `Wide_Character` have the same `character_literal` or language-defined name as defined for `Character`. Each of the graphic characters has a corresponding `character_literal`.

3.1/2

The predefined type `Wide_Wide_Character` is a character type whose values correspond to the 2147483648 code positions of the ISO/IEC 10646:2003 character set. Each of the graphic characters has a corresponding `character_literal` in `Wide_Wide_Character`. The first 65536 values of `Wide_Wide_Character` have the same `character_literal` or language-defined name as defined for `Wide_Character`.

3.2/2

The characters whose code position is larger than 16#FF# and which are not graphic characters have language-defined names which are formed by appending to the string "Hex-" the representation of their code position in hexadecimal as eight extended digits. As with other language-defined names, these names are usable only with the attributes `(Wide_)Wide_Image` and `(Wide_)Wide_Value`; they are not usable as enumeration literals.

Implementation Permissions

4/2

<This paragraph was deleted.>

Implementation Advice

5/2

<This paragraph was deleted.>

NOTES

6

25 The language-defined library package `Characters.Latin_1` (see Section 15.3.3 [A.3.3], page 573) includes the declaration of constants denoting control characters, lower case characters, and special characters of the predefined type `Character`.

7

26 A conventional character set such as `<EBCDIC>` can be declared as a character type; the internal codes of the characters can be specified by an `enumeration_representation_clause` as explained in clause Section 14.4 [13.4], page 500.

Examples

8

<Example of a character type: >

```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
```

4.5.3 3.5.3 Boolean Types

Static Semantics

1

There is a predefined enumeration type named Boolean, declared in the visible part of package Standard. It has the two enumeration literals False and True ordered with the relation False < True. Any descendant of the predefined type Boolean is called a <boolean> type.

4.5.4 3.5.4 Integer Types

1

An integer_type_definition defines an integer type; it defines either a <signed> integer type, or a <modular> integer type. The base range of a signed integer type includes at least the values of the specified range. A modular type is an integer type with all arithmetic modulo a specified positive <modulus>; such a type corresponds to an unsigned type with wrap-around semantics.

Syntax

2

```
integer_type_definition ::= signed_integer_type_definition | modular_type_definition
```

3

```
signed_integer_type_definition ::= range <static_>simple_expression .. <static_>simple_expression
```

4

```
modular_type_definition ::= mod <static_>expression
```

Name Resolution Rules

5

Each simple_expression in a signed_integer_type_definition is expected to be of any integer type; they need not be of the same type. The expression in a modular_type_definition is likewise expected to be of any integer type.

Legality Rules

6

The simple_expressions of a signed_integer_type_definition shall be static, and their values shall be in the range System.Min_Int .. System.Max_Int.

7

The expression of a modular_type_definition shall be static, and its value (the <modulus>) shall be positive, and shall be no greater than System.Max_Binary_Modulus if a power of 2, or no greater than System.Max_Nonbinary_Modulus if not.

Static Semantics

8

The set of values for a signed integer type is the (infinite) set of mathematical integers, though only values of the base range of the type are fully supported for run-time operations. The set of values for a modular integer type are the values from 0 to one less than the modulus, inclusive.

9

A `signed_integer_type_definition` defines an integer type whose base range includes at least the values of the `simple_expressions` and is symmetric about zero, excepting possibly an extra negative value. A `signed_integer_type_definition` also defines a constrained first subtype of the type, with a range whose bounds are given by the values of the `simple_expressions`, converted to the type being defined.

10

A `modular_type_definition` defines a modular type whose base range is from zero to one less than the given modulus. A `modular_type_definition` also defines a constrained first subtype of the type with a range that is the same as the base range of the type.

11

There is a predefined signed integer subtype named `Integer`, declared in the visible part of package `Standard`. It is constrained to the base range of its type.

12

`Integer` has two predefined subtypes, declared in the visible part of package `Standard`:

13

```
subtype Natural is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

14

A type defined by an `integer_type_definition` is implicitly derived from `<root_integer>`, an anonymous predefined (specific) integer type, whose base range is `System.Min_Int .. System.Max_Int`. However, the base range of the new type is not inherited from `<root_integer>`, but is instead determined by the range or modulus specified by the `integer_type_definition`. Integer literals are all of the type `<universal_integer>`, the universal type (see Section 4.4.1 [3.4.1], page 72) for the class rooted at `<root_integer>`, allowing their use with the operations of any integer type.

15

The `<position number>` of an integer value is equal to the value.

16/2

For every modular subtype `S`, the following attributes are defined:

16.1/2

`S'Mod`

`S'Mod` denotes
a function with
the following
specification:

16.2/2

```
function S'Mod (<Arg> : <universal_integer>)■
```


return S'Base

16.3/2

This function returns $\langle \text{Arg} \rangle \bmod \text{S'Modulus}$, as a value of the type of S.

17

S'Modulus

S'Modulus yields the modulus of the type of S, as a value of the type $\langle \text{universal_integer} \rangle$.

Dynamic Semantics

18

The elaboration of an integer_type_definition creates the integer type and its first subtype.

19

For a modular type, if the result of the execution of a predefined operator (see Section 5.5 [4.5], page 203) is outside the base range of the type, the result is reduced modulo the modulus of the type to a value that is within the base range of the type.

20

For a signed integer type, the exception Constraint_Error is raised by the execution of an operation that cannot deliver the correct result because it is outside the base range of the type. For any integer type, Constraint_Error is raised by the operators "/", "rem", and "mod" if the right operand is zero.

Implementation Requirements

21

In an implementation, the range of Integer shall include the range $-2^{15}+1 .. +2^{15}-1$.

22

If Long_Integer is predefined for an implementation, then its range shall include the range $-2^{31}+1 .. +2^{31}-1$.

23

System.Max_Binary_Modulus shall be at least 2^{16} .

Implementation Permissions

24

For the execution of a predefined operation of a signed integer type, the implementation need not raise Constraint_Error if the result is outside the base range of the type, so long as the correct result is produced.

25

An implementation may provide additional predefined signed integer types, declared in the visible part of Standard, whose first subtypes have names of the form Short_Integer, Long_Integer, Short_Short_Integer, Long_Long_Integer, etc. Different predefined integer types are allowed to have the same base range. However, the range of Integer should be no

wider than that of `Long_Integer`. Similarly, the range of `Short_Integer` (if provided) should be no wider than `Integer`. Corresponding recommendations apply to any other predefined integer types. There need not be a named integer type corresponding to each distinct base range supported by an implementation. The range of each first subtype should be the base range of its type.

26

An implementation may provide `<nonstandard integer types>`, descendants of `<root_integer>` that are declared outside of the specification of package `Standard`, which need not have all the standard characteristics of a type defined by an `integer_type_definition`. For example, a nonstandard integer type might have an asymmetric base range or it might not be allowed as an array or loop index (a very long integer). Any type descended from a nonstandard integer type is also nonstandard. An implementation may place arbitrary restrictions on the use of such types; it is implementation defined whether operators that are predefined for "any integer type" are defined for a particular nonstandard integer type. In any case, such types are not permitted as `explicit_generic_actual_parameters` for formal scalar types — see Section 13.5.2 [12.5.2], page 466.

27

For a one's complement machine, the high bound of the base range of a modular type whose modulus is one less than a power of 2 may be equal to the modulus, rather than one less than the modulus. It is implementation defined for which powers of 2, if any, this permission is exercised.

27.1/1

For a one's complement machine, implementations may support non-binary modulus values greater than `System.Max_Nonbinary_Modulus`. It is implementation defined which specific values greater than `System.Max_Nonbinary_Modulus`, if any, are supported.

Implementation Advice

28

An implementation should support `Long_Integer` in addition to `Integer` if the target machine supports 32-bit (or longer) arithmetic. No other named integer subtypes are recommended for package `Standard`. Instead, appropriate named integer subtypes should be provided in the library package `Interfaces` (see Section 16.2 [B.2], page 900).

29

An implementation for a two's complement machine should support modular types with a binary modulus up to `System.Max_Int*2+2`. An implementation should support a nonbinary modulus up to `Integer'Last`.

NOTES

30

27 Integer literals are of the anonymous predefined integer type `<universal_integer>`. Other integer types have no literals. However, the overload resolution rules (see Section 9.6 [8.6], page 324, "Section 9.6 [8.6], page 324, The Context of Overload Resolution") allow expressions of the type `<universal_integer>` whenever an integer type is expected.

31

28 The same arithmetic operators are predefined for all signed integer types defined by a `signed_integer_type_definition` (see Section 5.5 [4.5], page 203, "Section 5.5 [4.5], page 203, Operators and Expression Evaluation"). For modular types, these same operators are predefined, plus bit-wise logical operators (`and`, `or`, `xor`, and `not`). In addition, for the unsigned types declared in the language-defined package `Interfaces` (see Section 16.2 [B.2], page 900), functions are defined that provide bit-wise shifting and rotating.

32

29 Modular types match a `generic_formal_parameter_declaration` of the form `"type T is mod <>;"`; signed integer types match `"type T is range <>,"` (see Section 13.5.2 [12.5.2], page 466).

Examples

33

<Examples of integer types and subtypes: >

34

```
type Page_Num is range 1 .. 2_000;
type Line_Size is range 1 .. Max_Line_Size;
```

35

```
subtype Small_Int is Integer range -10 .. 10;
subtype Column_Ptr is Line_Size range 1 .. 10;
subtype Buffer_Size is Integer range 0 .. Max;
```

36

```
type Byte is mod 256; --< an unsigned byte>
type Hash_Index is mod 97; --< modulus is prime>
```

4.5.5 3.5.5 Operations of Discrete Types

Static Semantics

1

For every discrete subtype `S`, the following attributes are defined:

2

`S'Pos`

`S'Pos` denotes
a function with
the following
specification:

3

```
function S'Pos(<Arg> : S'Base)
  return <universal_integer>
```

4

This function returns the position number of the value of <Arg>, as a value of type <universal_integer>.

5

S'Val

S'Val denotes a function with the following specification:

6

```
function S'Val(<Arg> : <universal_integer>)
  return S'Base
```

7

This function returns a value of the type of S whose position number equals the value of <Arg>. For the evaluation of a call on S'Val, if there is no value in the base range of its type with the given position number, Constraint_Error is raised.

Implementation Advice

8

For the evaluation of a call on S'Pos for an enumeration subtype, if the value of the operand does not correspond to the internal code for any enumeration literal of its type (perhaps due to an uninitialized variable), then the implementation should raise Program_Error. This is particularly important for enumeration types with noncontiguous internal codes specified by an enumeration_representation_clause (see [S0287], page 500).

NOTES

9

30 Indexing and loop iteration use values of discrete types.

10

31 The predefined operations of a discrete type include the assignment operation, qualification, the membership tests, and the relational operators; for a boolean type they include the short-circuit control forms and the logical operators; for an integer type they include type conversion to and from other numeric types, as well as the binary and unary adding operators $-$ and $+$, the multiplying operators, the unary operator abs , and the exponentiation operator. The assignment operation is described in Section 6.2 [5.2], page 242. The other predefined operations are described in Section 4.

11

32 As for all types, objects of a discrete type have `Size` and `Address` attributes (see Section 14.3 [13.3], page 486).

12

33 For a subtype of a discrete type, the result delivered by the attribute `Val` might not belong to the subtype; similarly, the actual parameter of the attribute `Pos` need not belong to the subtype. The following relations are satisfied (in the absence of an exception) by these attributes:

13

$$\begin{aligned} S'Val(S'Pos(X)) &= X \\ S'Pos(S'Val(N)) &= N \end{aligned}$$

Examples

14

<Examples of attributes of discrete subtypes: >

15

--< For the types and subtypes declared in subclause Section 4.5.1 [3.5.1], page 92 the following hold: >

16

```
-- Color'First   = White,   Color'Last   = Black
-- Rainbow'First = Red,     Rainbow'Last = Blue
```

17

```
-- Color'Succ(Blue) = Rainbow'Succ(Blue) = Brown
```

```
-- Color'Pos(Blue) = Rainbow'Pos(Blue) = 4
-- Color'Val(0)   = Rainbow'Val(0)   = White
```

4.5.6 3.5.6 Real Types

1

Real types provide approximations to the real numbers, with relative bounds on errors for floating point types, and with absolute bounds for fixed point types.

Syntax

2

```
real_type_definition ::=
  floating_point_definition | fixed_point_definition
```

Static Semantics

3

A type defined by a `real_type_definition` is implicitly derived from `<root_real>`, an anonymous predefined (specific) real type. Hence, all real types, whether floating point or fixed point, are in the derivation class rooted at `<root_real>`.

4

Real literals are all of the type `<universal_real>`, the universal type (see Section 4.4.1 [3.4.1], page 72) for the class rooted at `<root_real>`, allowing their use with the operations of any real type. Certain multiplying operators have a result type of `<universal_fixed>` (see Section 5.5.5 [4.5.5], page 213), the universal type for the class of fixed point types, allowing the result of the multiplication or division to be used where any specific fixed point type is expected.

Dynamic Semantics

5

The elaboration of a `real_type_definition` consists of the elaboration of the `floating_point_definition` or the `fixed_point_definition`.

Implementation Requirements

6

An implementation shall perform the run-time evaluation of a use of a predefined operator of `<root_real>` with an accuracy at least as great as that of any floating point type definable by a `floating_point_definition`.

Implementation Permissions

7/2

For the execution of a predefined operation of a real type, the implementation need not raise `Constraint_Error` if the result is outside the base range of the type, so long as the correct result is produced, or the `Machine_Overflows` attribute of the type is `False` (see Section 21.2 [G.2], page 1103).

8

An implementation may provide `<nonstandard real types>`, descendants of `<root_real>` that are declared outside of the specification of package `Standard`, which need not have all the standard characteristics of a type defined by a `real_type_definition`. For example, a nonstandard real type might have an asymmetric or unsigned base range, or its predefined operations might wrap around or "saturate" rather than overflow (modular or saturating arithmetic), or it might not conform to the accuracy model (see Section 21.2 [G.2], page 1103). Any

type descended from a nonstandard real type is also nonstandard. An implementation may place arbitrary restrictions on the use of such types; it is implementation defined whether operators that are predefined for "any real type" are defined for a particular nonstandard real type. In any case, such types are not permitted as `explicit_generic_actual_parameters` for formal scalar types — see Section 13.5.2 [12.5.2], page 466.

NOTES

9

34 As stated, real literals are of the anonymous predefined real type `<universal_real>`. Other real types have no literals. However, the overload resolution rules (see Section 9.6 [8.6], page 324) allow expressions of the type `<universal_real>` whenever a real type is expected.

4.5.7 3.5.7 Floating Point Types

1

For floating point types, the error bound is specified as a relative precision by giving the required minimum number of significant decimal digits.

Syntax

2

```
floating_point_definition ::=  
  digits <static_>expression [real_range_specification]
```

3

```
real_range_specification ::=  
  range <static_>simple_expression .. <static_>simple_expression
```

Name Resolution Rules

4

The `<requested decimal precision>`, which is the minimum number of significant decimal digits required for the floating point type, is specified by the value of the expression given after the reserved word `digits`. This expression is expected to be of any integer type.

5

Each `simple_expression` of a `real_range_specification` is expected to be of any real type; the types need not be the same.

Legality Rules

6

The requested decimal precision shall be specified by a static expression whose value is positive and no greater than `System.Max_Base_Digits`. Each `simple_expression` of a `real_range_specification` shall also be static. If the `real_range_specification` is omitted, the requested decimal precision shall be no greater than `System.Max_Digits`.

7

A `floating_point_definition` is illegal if the implementation does not support a floating point type that satisfies the requested decimal precision and range.

Static Semantics

8

The set of values for a floating point type is the (infinite) set of rational numbers. The <machine numbers> of a floating point type are the values of the type that can be represented exactly in every unconstrained variable of the type. The base range (see Section 4.5 [3.5], page 76) of a floating point type is symmetric around zero, except that it can include some extra negative values in some implementations.

9

The <base decimal precision> of a floating point type is the number of decimal digits of precision representable in objects of the type. The <safe range> of a floating point type is that part of its base range for which the accuracy corresponding to the base decimal precision is preserved by all predefined operations.

10

A `floating_point_definition` defines a floating point type whose base decimal precision is no less than the requested decimal precision. If a `real_range_specification` is given, the safe range of the floating point type (and hence, also its base range) includes at least the values of the simple expressions given in the `real_range_specification`. If a `real_range_specification` is not given, the safe (and base) range of the type includes at least the values of the range $-10.0^{*(4*D)} .. +10.0^{*(4*D)}$ where D is the requested decimal precision. The safe range might include other values as well. The attributes `Safe_First` and `Safe_Last` give the actual bounds of the safe range.

11

A `floating_point_definition` also defines a first subtype of the type. If a `real_range_specification` is given, then the subtype is constrained to a range whose bounds are given by a conversion of the values of the simple expressions of the `real_range_specification` to the type being defined. Otherwise, the subtype is unconstrained.

12

There is a predefined, unconstrained, floating point subtype named `Float`, declared in the visible part of package `Standard`.

Dynamic Semantics

13

The elaboration of a `floating_point_definition` creates the floating point type and its first subtype.

Implementation Requirements

14

In an implementation that supports floating point types with 6 or more digits of precision, the requested decimal precision for `Float` shall be at least 6.

15

If `Long.Float` is predefined for an implementation, then its requested decimal precision shall be at least 11.

Implementation Permissions

16

An implementation is allowed to provide additional predefined floating point types, declared in the visible part of `Standard`, whose (unconstrained) first subtypes have names of the form `Short_Float`, `Long_Float`, `Short_Short_Float`, `Long_Long_Float`, etc. Different predefined

floating point types are allowed to have the same base decimal precision. However, the precision of Float should be no greater than that of Long_Float. Similarly, the precision of Short_Float (if provided) should be no greater than Float. Corresponding recommendations apply to any other predefined floating point types. There need not be a named floating point type corresponding to each distinct base decimal precision supported by an implementation.

Implementation Advice

17

An implementation should support Long_Float in addition to Float if the target machine supports 11 or more digits of precision. No other named floating point subtypes are recommended for package Standard. Instead, appropriate named floating point subtypes should be provided in the library package Interfaces (see Section 16.2 [B.2], page 900).

NOTES

18

35 If a floating point subtype is unconstrained, then assignments to variables of the subtype involve only Overflow_Checks, never Range_Checks.

Examples

19

<Examples of floating point types and subtypes:>

20

```
type Coefficient is digits 10 range -1.0 .. 1.0;
```

21

```
type Real is digits 8;
type Mass is digits 7 range 0.0 .. 1.0E35;
```

22

```
subtype Probability is Real range 0.0 .. 1.0;  --< a subtype with a smaller ra
```

4.5.8 3.5.8 Operations of Floating Point Types

Static Semantics

1

The following attribute is defined for every floating point subtype S:

2/1

S'Digits

S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type <universal_integer>.

The requested decimal precision of the base subtype of a floating point type `<T>` is defined to be the largest value of `<d>` for which

$$\text{ceiling}(\langle d \rangle * \log(10) / \log(T.\text{Machine_Radix})) + \langle g \rangle \leq T.\text{Model_Mantissa}$$

where `g` is 0 if `Machine_Radix` is a positive power of 10 and 1 otherwise.

NOTES

3

36 The predefined operations of a floating point type include the assignment operation, qualification, the membership tests, and explicit conversion to and from other numeric types. They also include the relational operators and the following predefined arithmetic operators: the binary and unary adding operators `-` and `+`, certain multiplying operators, the unary operator `abs`, and the exponentiation operator.

4

37 As for all types, objects of a floating point type have `Size` and `Address` attributes (see Section 14.3 [13.3], page 486). Other attributes of floating point types are defined in Section 15.5.3 [A.5.3], page 663.

4.5.9 3.5.9 Fixed Point Types

1

A fixed point type is either an ordinary fixed point type, or a decimal fixed point type. The error bound of a fixed point type is specified as an absolute value, called the `<delta>` of the fixed point type.

Syntax

2

`fixed_point_definition ::= ordinary_fixed_point_definition | decimal_fixed_point_definition`

3

`ordinary_fixed_point_definition ::=`
`delta <static_>expression real_range_specification`

4

```
decimal_fixed_point_definition ::=  
    delta <static_>expression digits <static_>expression [real_range_specification]
```

5

```
digits_constraint ::=  
    digits <static_>expression [range_constraint]  
    Name Resolution Rules
```

6

For a type defined by a `fixed_point_definition`, the `<delta>` of the type is specified by the value of the expression given after the reserved word `delta`; this expression is expected to be of any real type. For a type defined by a `decimal_fixed_point_definition` (a `<decimal>` fixed point type), the number of significant decimal digits for its first subtype (the `<digits>` of the first subtype) is specified by the expression given after the reserved word `digits`; this expression is expected to be of any integer type.

Legality Rules

7

In a `fixed_point_definition` or `digits_constraint`, the expressions given after the reserved words `delta` and `digits` shall be static; their values shall be positive.

8/2

The set of values of a fixed point type comprise the integral multiples of a number called the `<small>` of the type. The `<machine numbers>` of a fixed point type are the values of the type that can be represented exactly in every unconstrained variable of the type. For a type defined by an `ordinary_fixed_point_definition` (an `<ordinary>` fixed point type), the `<small>` may be specified by an `attribute_definition_clause` (see [S0286], page 487) (see Section 14.3 [13.3], page 486); if so specified, it shall be no greater than the `<delta>` of the type. If not specified, the `<small>` of an ordinary fixed point type is an implementation-defined power of two less than or equal to the `<delta>`.

9

For a decimal fixed point type, the `<small>` equals the `<delta>`; the `<delta>` shall be a power of 10. If a `real_range_specification` is given, both bounds of the range shall be in the range $-(10^{**<digits>-1})^{*}<delta> .. +(10^{**<digits>-1})^{*}<delta>$.

10

A `fixed_point_definition` is illegal if the implementation does not support a fixed point type with the given `<small>` and specified range or `<digits>`.

11

For a `subtype_indication` with a `digits_constraint`, the `subtype_mark` shall denote a decimal fixed point subtype.

Static Semantics

12

The base range (see Section 4.5 [3.5], page 76) of a fixed point type is symmetric around zero, except possibly for an extra negative value in some implementations.

13

An `ordinary_fixed_point_definition` defines an ordinary fixed point type whose base range

includes at least all multiples of `<small>` that are between the bounds specified in the `real_range_specification`. The base range of the type does not necessarily include the specified bounds themselves. An `ordinary_fixed_point_definition` (see [S0048], page 106) also defines a constrained first subtype of the type, with each bound of its range given by the closer to zero of:

14

- the value of the conversion to the fixed point type of the corresponding expression of the `real_range_specification`;

15

- the corresponding bound of the base range.

16

A `decimal_fixed_point_definition` defines a decimal fixed point type whose base range includes at least the range $-(10^{**}<digits>-1)^{*}<delta> .. +(10^{**}<digits>-1)^{*}<delta>$. A `decimal_fixed_point_definition` also defines a constrained first subtype of the type. If a `real_range_specification` is given, the bounds of the first subtype are given by a conversion of the values of the expressions of the `real_range_specification`. Otherwise, the range of the first subtype is $-(10^{**}<digits>-1)^{*}<delta> .. +(10^{**}<digits>-1)^{*}<delta>$.

Dynamic Semantics

17

The elaboration of a `fixed_point_definition` creates the fixed point type and its first subtype.

18

For a `digits_constraint` on a decimal fixed point subtype with a given `<delta>`, if it does not have a `range_constraint`, then it specifies an implicit range $-(10^{**}<D>-1)^{*}<delta> .. +(10^{**}<D>-1)^{*}<delta>$, where `<D>` is the value of the expression. A `digits_constraint` is `<compatible>` with a decimal fixed point subtype if the value of the expression is no greater than the `<digits>` of the subtype, and if it specifies (explicitly or implicitly) a range that is compatible with the subtype.

19

The elaboration of a `digits_constraint` consists of the elaboration of the `range_constraint`, if any. If a `range_constraint` is given, a check is made that the bounds of the range are both in the range $-(10^{**}<D>-1)^{*}<delta> .. +(10^{**}<D>-1)^{*}<delta>$, where `<D>` is the value of the (static) expression given after the reserved word `digits`. If this check fails, `Constraint_Error` is raised.

Implementation Requirements

20

The implementation shall support at least 24 bits of precision (including the sign bit) for fixed point types.

Implementation Permissions

21

Implementations are permitted to support only `<small>`s that are a power of two. In particular, all decimal fixed point type declarations can be disallowed. Note however that

conformance with the Information Systems Annex requires support for decimal <small>s, and decimal fixed point type declarations with <digits> up to at least 18.

NOTES

22

38 The base range of an ordinary fixed point type need not include the specified bounds themselves so that the range specification can be given in a natural way, such as:

23

```
type Fraction is delta 2.0**(-15) range -1.0 .. 1.0;
```

24

With 2's complement hardware, such a type could have a signed 16-bit representation, using 1 bit for the sign and 15 bits for fraction, resulting in a base range of $-1.0 .. 1.0 - 2.0^{(-15)}$.

Examples

25

<Examples of fixed point types and subtypes:>

26

```
type Volt is delta 0.125 range 0.0 .. 255.0;
```

27

```
-- <A pure fraction which requires all the available>
-- <space in a word can be declared as the type Fraction:>
type Fraction is delta System.Fine_Delta range -1.0 .. 1.0;
-- <Fraction'Last = 1.0 - System.Fine_Delta>
```

28

```
type Money is delta 0.01 digits 15; -- <decimal fixed point>
subtype Salary is Money digits 10;
-- <Money'Last = 10.0**13 - 0.01, Salary'Last = 10.0**8 - 0.01>
```

4.5.10 3.5.10 Operations of Fixed Point Types

Static Semantics

1

The following attributes are defined for every fixed point subtype S:

2/1

S'Small

S'Small denotes the
<small> of the type of

S. The value of this attribute is of the type `<universal_real>`. Small may be specified for nonderived ordinary fixed point types via an `attribute_definition_clause` (see [S0286], page 487) (see Section 14.3 [13.3], page 486); the expression of such a clause shall be static.

3 S'Delta

S'Delta denotes the `<delta>` of the fixed point subtype S. The value of this attribute is of the type `<universal_real>`.

4 S'Fore

S'Fore yields the minimum number of characters needed before the decimal point for the decimal representation of any value of the subtype S, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute

is of the type
<universal_integer>.

5
S'Aft

S'Aft yields the number of decimal digits needed after the decimal point to accommodate the <delta> of the subtype S, unless the <delta> of the subtype S is greater than 0.1, in which case the attribute yields the value one. (S'Aft is the smallest positive integer N for which $(10^{**}N)*S'Delta$ is greater than or equal to one.) The value of this attribute is of the type <universal_integer>.

6
The following additional attributes are defined for every decimal fixed point subtype S:

7
S'Digits

S'Digits denotes the <digits> of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type <universal_integer>. Its value is determined as follows:

8

- For a first subtype or a subtype defined by a subtype_indication with a digits_constraint, the digits is the value of the expression given after the reserved word digits;

9

- For a subtype defined by a subtype_indication without a digits_constraint, the digits of the subtype is the same as that of the subtype denoted by the subtype_mark in the subtype_indication.

10

- The digits of a base subtype is the largest integer $\langle D \rangle$ such that the range $-(10^{**}\langle D \rangle - 1)^{\langle \text{delta} \rangle}$
 \dots
 $+(10^{**}\langle D \rangle - 1)^{\langle \text{delta} \rangle}$ is included in the base range of the type.

11
 S'Scale

S'Scale denotes the $\langle \text{scale} \rangle$ of the subtype S, defined as the value

N such that S'Delta = 10.0**(-N). The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S. The value of this attribute is of the type <universal_integer>.

12
S'Round

S'Round denotes a function with the following specification:

13

```
function S'Round(<X> : <universal_real>)■  
    return S'Base
```

14

The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S).

NOTES

15

39 All subtypes of a fixed point type will have the same value for the Delta attribute, in the absence of delta_constraints (see Section 23.3 [J.3], page 1167).

16

40 S'Scale is not always the same as S'Aft for a decimal subtype; for example, if S'Delta = 1.0 then S'Aft is 1 while S'Scale is 0.

17

41 The predefined operations of a fixed point type include the assignment operation, qualification, the membership tests, and explicit conversion to and from other numeric types. They also include the

relational operators and the following predefined arithmetic operators: the binary and unary adding operators $-$ and $+$, multiplying operators, and the unary operator `abs`.

18

42 As for all types, objects of a fixed point type have `Size` and `Address` attributes (see Section 14.3 [13.3], page 486). Other attributes of fixed point types are defined in Section 15.5.4 [A.5.4], page 679.

4.6 3.6 Array Types

1

An `<array>` object is a composite object consisting of components which all have the same subtype. The name for a component of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of the components.

Syntax

2

```
array_type_definition ::=
    unconstrained_array_definition | constrained_array_definition
```

3

```
unconstrained_array_definition ::=
    array(index_subtype_definition {, index_subtype_definition}) of component_definition
```

4

```
index_subtype_definition ::= subtype_mark range <>
```

5

```
constrained_array_definition ::=
    array (discrete_subtype_definition {, discrete_subtype_definition}) of component_definition
```

6

```
discrete_subtype_definition ::= <discrete_>subtype_indication | range
```

7/2

```
component_definition ::=
    [aliased] subtype_indication
    | [aliased] access_definition
```

Name Resolution Rules

8

For a `discrete_subtype_definition` that is a range, the range shall resolve to be of some specific discrete type; which discrete type shall be determined without using any context

other than the bounds of the range itself (plus the preference for `<root_integer>` — see Section 9.6 [8.6], page 324).

Legality Rules

9

Each `index_subtype_definition` or `discrete_subtype_definition` in an `array_type_definition` defines an `<index subtype>`; its type (the `<index type>`) shall be discrete.

10

The subtype defined by the `subtype_indication` of a `component_definition` (the `<component subtype>`) shall be a definite subtype.

11/2

`<This paragraph was deleted.>`

Static Semantics

12

An array is characterized by the number of indices (the `<dimensionality>` of the array), the type and position of each index, the lower and upper bounds for each index, and the subtype of the components. The order of the indices is significant.

13

A one-dimensional array has a distinct component for each possible index value. A multidimensional array has a distinct component for each possible sequence of index values that can be formed by selecting one value for each index position (in the given order). The possible values for a given index are all the values between the lower and upper bounds, inclusive; this range of values is called the `<index range>`. The `<bounds>` of an array are the bounds of its index ranges. The `<length>` of a dimension of an array is the number of values of the index range of the dimension (zero for a null range). The `<length>` of a one-dimensional array is the length of its only dimension.

14

An `array_type_definition` defines an array type and its first subtype. For each object of this array type, the number of indices, the type and position of each index, and the subtype of the components are as in the type definition; the values of the lower and upper bounds for each index belong to the corresponding index subtype of its type, except for null arrays (see Section 4.6.1 [3.6.1], page 117).

15

An `unconstrained_array_definition` defines an array type with an unconstrained first subtype. Each `index_subtype_definition` (see [S0053], page 114) defines the corresponding index subtype to be the subtype denoted by the `subtype_mark` (see [S0028], page 56). The compound delimiter `<>` (called a `<box>`) of an `index_subtype_definition` stands for an undefined range (different objects of the type need not have the same bounds).

16

A `constrained_array_definition` defines an array type with a constrained first subtype. Each `discrete_subtype_definition` (see [S0055], page 114) defines the corresponding index subtype, as well as the corresponding index range for the constrained first subtype. The `<constraint>` of the first subtype consists of the bounds of the index ranges.

17

The discrete subtype defined by a `discrete_subtype_definition` (see [S0055], page 114) is

either that defined by the `subtype_indication` (see [S0027], page 56), or a subtype determined by the range as follows:

18

- If the type of the range resolves to `<root_integer>`, then the `discrete_subtype_definition` defines a subtype of the predefined type `Integer` with bounds given by a conversion to `Integer` of the bounds of the range;

19

- Otherwise, the `discrete_subtype_definition` defines a subtype of the type of the range, with the bounds given by the range.

20

The `component_definition` of an `array_type_definition` defines the nominal subtype of the components. If the reserved word `aliased` appears in the `component_definition`, then each component of the array is aliased (see Section 4.10 [3.10], page 156).

Dynamic Semantics

21

The elaboration of an `array_type_definition` creates the array type and its first subtype, and consists of the elaboration of any `discrete_subtype_definition` (see [S0055], page 114)s and the `component_definition` (see [S0056], page 114).

22/2

The elaboration of a `discrete_subtype_definition` that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the `subtype_indication` (see [S0027], page 56) or the evaluation of the range. The elaboration of a `discrete_subtype_definition` that contains one or more per-object expressions is defined in Section 4.8 [3.8], page 130. The elaboration of a `component_definition` (see [S0056], page 114) in an `array_type_definition` (see [S0051], page 114) consists of the elaboration of the `subtype_indication` (see [S0027], page 56) or `access_definition`. The elaboration of any `discrete_subtype_definition` (see [S0055], page 114)s and the elaboration of the `component_definition` (see [S0056], page 114) are performed in an arbitrary order.

NOTES

23

43 All components of an array have the same subtype. In particular, for an array of components that are one-dimensional arrays, this means that all components have the same bounds and hence the same length.

24

44 Each elaboration of an `array_type_definition` creates a distinct array type. A consequence of this is that each object whose `object_declaration` contains an `array_type_definition` is of its own unique type.

Examples

25

<Examples of type declarations with unconstrained array definitions: >

26

```
type Vector      is array(Integer range <>) of Real;
type Matrix      is array(Integer range <>, Integer range <>) of Real;
type Bit_Vector is array(Integer range <>) of Boolean;
type Roman       is array(Positive range <>) of Roman_Digit; --< see Section 4.5.2
[3.5.2], page 93>
```

27

<Examples of type declarations with constrained array definitions: >

28

```
type Table       is array(1 .. 10) of Integer;
type Schedule    is array(Day) of Boolean;
type Line        is array(1 .. Max_Line_Size) of Character;
```

29

<Examples of object declarations with array type definitions: >

30/2

```
Grid           : array(1 .. 80, 1 .. 100) of Boolean;
Mix            : array(Color range Red .. Green) of Boolean;
Msg_Table      : constant array(Error_Code) of access constant String :=
    (Too_Big => new String("Result too big"), Too_Small => ...);
Page          : array(Positive range <>) of Line := --< an array of arrays>
    (1 | 50 => Line'(1 | Line'Last => '+', others => '-'), --< see Section 5.3.3
    [4.3.3], page 196>
    2 .. 49 => Line'(1 | Line'Last => '|', others => ' '));
--< Page is constrained by its initial value to (1..50)>
```

4.6.1 3.6.1 Index Constraints and Discrete Ranges

1

An `index_constraint` determines the range of possible values for every index of an array subtype, and thereby the corresponding array bounds.

Syntax

2

```
index_constraint ::= (discrete_range {, discrete_range})
```

3

```
discrete_range ::= <discrete_>subtype_indication | range
Name Resolution Rules
```

4

The type of a `discrete_range` is the type of the subtype defined by the `subtype_indication`,

or the type of the range. For an `index_constraint`, each `discrete_range` shall resolve to be of the type of the corresponding index.

Legality Rules

5

An `index_constraint` shall appear only in a `subtype_indication` whose `subtype_mark` denotes either an unconstrained array subtype, or an unconstrained access subtype whose designated subtype is an unconstrained array subtype; in either case, the `index_constraint` shall provide a `discrete_range` for each index of the array type.

Static Semantics

6

A `discrete_range` defines a range whose bounds are given by the range, or by the range of the subtype defined by the `subtype_indication`.

Dynamic Semantics

7

An `index_constraint` is `<compatible>` with an unconstrained array subtype if and only if the index range defined by each `discrete_range` is compatible (see Section 4.5 [3.5], page 76) with the corresponding index subtype. If any of the `discrete_ranges` defines a null range, any array thus constrained is a `<null array>`, having no components. An array value `<satisfies>` an `index_constraint` if at each index position the array value and the `index_constraint` have the same index bounds.

8

The elaboration of an `index_constraint` consists of the evaluation of the `discrete_range(s)`, in an arbitrary order. The evaluation of a `discrete_range` consists of the elaboration of the `subtype_indication` or the evaluation of the range.

NOTES

9

45 The elaboration of a `subtype_indication` consisting of a `subtype_mark` followed by an `index_constraint` checks the compatibility of the `index_constraint` with the `subtype_mark` (see Section 4.2.2 [3.2.2], page 55).

10

46 Even if an array value does not satisfy the index constraint of an array subtype, `Constraint_Error` is not raised on conversion to the array subtype, so long as the length of each dimension of the array value and the array subtype match. See Section 5.6 [4.6], page 219.

Examples

11

`<Examples of array declarations including an index constraint: >`

12

```
Board      : Matrix(1 .. 8, 1 .. 8); --< see Section 4.6 [3.6],  
page 114>
```

```
Rectangle : Matrix(1 .. 20, 1 .. 30);
Inverse   : Matrix(1 .. N, 1 .. N);  --< N need not be static >
```

13

```
Filter    : Bit_Vector(0 .. 31);
```

14

<Example of array declaration with a constrained array subtype: >

15

```
My_Schedule : Schedule;  --< all arrays of type Schedule have the same bounds>
```

16

<Example of record type with a component that is an array: >

17

```
type Var_Line(Length : Natural) is
  record
    Image : String(1 .. Length);
  end record;
```

18

```
Null_Line : Var_Line(0);  --< Null_Line.Image is a null array>
```

4.6.2 3.6.2 Operations of Array Types

Legality Rules

1

The argument N used in the attribute designators for the N -th dimension of an array shall be a static expression of some integer type. The value of N shall be positive (nonzero) and no greater than the dimensionality of the array.

Static Semantics

2/1

The following attributes are defined for a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

3

A 'First

A 'First denotes the lower bound of the first index range; its type is the corresponding index type.

4

A 'First(N)

A 'First(N) denotes the lower bound

of the N-th index range; its type is the corresponding index type.

5

A'Last

A'Last denotes the upper bound of the first index range; its type is the corresponding index type.

6

A'Last(N)

A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type.

7

A'Range

A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once.

8

A'Range(N)

A'Range(N) is equivalent to the range A'First(N) .. A'Last(N), except that the prefix A is only evaluated once.

9

A'Length

A'Length denotes the number of values of the first index range (zero for a null range); its type is <universal_integer>.

10

A'Length(N)

A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is <universal_integer>.

Implementation Advice

11

An implementation should normally represent multidimensional arrays in row-major order, consistent with the notation used for multidimensional array aggregates (see Section 5.3.3 [4.3.3], page 196). However, if a pragma Convention(Fortran, ...) applies to a multidimensional array type, then column-major order should be used instead (see Section 16.5 [B.5], page 945, "Section 16.5 [B.5], page 945, Interfacing with Fortran").

NOTES

12

47 The attribute_references A'First and A'First(1) denote the same value. A similar relation exists for the attribute_references A'Last, A'Range, and A'Length. The following relation is satisfied (except for a null array) by the above attributes if the index type is an integer type:

13

$$A'Length(N) = A'Last(N) - A'First(N) + 1$$

14

48 An array type is limited if its component type is limited (see Section 8.5 [7.5], page 292).

15

49 The predefined operations of an array type include the membership tests, qualification, and explicit conversion. If the array type is not limited, they also include assignment and the predefined equality operators. For a one-dimensional array type, they include the predefined concatenation operators (if nonlimited) and, if the component type is discrete, the predefined relational operators; if the component type is boolean, the predefined logical operators are also included.

16/2

50 A component of an array can be named with an indexed_component. A value of an array type can be specified with an array-aggregate. For a one-dimensional array type, a slice

of the array can be named; also, string literals are defined if the component type is a character type.

Examples

17

<Examples (using arrays declared in the examples of subclause Section 4.6.1 [3.6.1], page 117):>

18

```
-- Filter'First      = 0   Filter'Last      = 31   Filter'Length = 32█
-- Rectangle'Last(1) = 20  Rectangle'Last(2) = 30
```

4.6.3 3.6.3 String Types

Static Semantics

1

A one-dimensional array type whose component type is a character type is called a <string> type.

2/2

There are three predefined string types, `String`, `Wide_String`, and `Wide_Wide_String`, each indexed by values of the predefined subtype `Positive`; these are declared in the visible part of package `Standard`:

3

```
subtype Positive is Integer range 1 .. Integer'Last;
```

4/2

```
type String is array(Positive range <>) of Character;
type Wide_String is array(Positive range <>) of Wide_Character;
type Wide_Wide_String is array(Positive range <>) of Wide_Wide_Character;█
```

NOTES

5

51 String literals (see Section 3.6 [2.6], page 42, and Section 5.2 [4.2], page 189) are defined for all string types. The concatenation operator `&` is predefined for string types, as for all nonlimited one-dimensional array types. The ordering operators `<`, `<=`, `>`, and `>=` are predefined for string types, as for all one-dimensional discrete array types; these ordering operators correspond to lexicographic order (see Section 5.5.2 [4.5.2], page 206).

Examples

6

<Examples of string objects:>

7

```
Stars      : String(1 .. 120) := (1 .. 120 => '*' );
Question   : constant String := "How many characters?";
  --< Question'First = 1, Question'Last = 20>
  --< Question'Length = 20 (the number of characters)>
```

8

```
Ask_Twice  : String := Question & Question;  --< constrained to (1..40)>
Ninety_Six : constant Roman := "XCVI";  --< see Section 4.5.2 [3.5.2],
page 93 and Section 4.6 [3.6], page 114>
```

4.7 3.7 Discriminants

1/2

A composite type (other than an array or interface type) can have discriminants, which parameterize the type. A `known_discriminant_part` specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An `unknown_discriminant_part` in the declaration of a view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a view are indefinite subtypes.

Syntax

2/2

```
discriminant_part ::= unknown_discriminant_part | known_discriminant_part
```

3

```
unknown_discriminant_part ::= (<>)
```

4

```
known_discriminant_part ::=
  (discriminant_specification {; discriminant_specification})
```

5/2

```
discriminant_specification ::=
  defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
  | defining_identifier_list : access_definition [:= default_expression]
```

6

```
default_expression ::= expression
Name Resolution Rules
```

7

The expected type for the `default_expression` of a `discriminant_specification` is that of the corresponding discriminant.

Legality Rules

8/2

A `discriminant_part` is only permitted in a declaration for a composite type that is not an array or interface type (this includes generic formal types). A type declared with a `known_discriminant_part` is called a <discriminated> type, as is a type that inherits (known) discriminants.

9/2

The subtype of a discriminant may be defined by an optional `null_exclusion` and a `subtype_mark`, in which case the `subtype_mark` shall denote a discrete or access subtype, or it may be defined by an `access_definition`. A discriminant that is defined by an `access_definition` is called an <access discriminant> and is of an anonymous access type.

9.1/2

`Default_expressions` shall be provided either for all or for none of the discriminants of a `known_discriminant_part` (see [S0061], page 123). No `default_expression` (see [S0063], page 123)s are permitted in a `known_discriminant_part` (see [S0061], page 123) in a declaration of a tagged type or a generic formal type.

10/2

A `discriminant_specification` for an access discriminant may have a `default_expression` only in the declaration for a task or protected type, or for a type that is a descendant of an explicitly limited record type. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit.

11/2

<This paragraph was deleted.>

12

For a type defined by a `derived_type_definition`, if a `known_discriminant_part` is provided in its declaration, then:

13

- The parent subtype shall be constrained;

14

- If the parent type is not a tagged type, then each discriminant of the derived type shall be used in the constraint defining the parent subtype;

15

- If a discriminant is used in the constraint defining the parent subtype, the subtype of the discriminant shall be statically compatible (see Section 5.9.1 [4.9.1], page 238) with the subtype of the corresponding parent discriminant.

16

The type of the `default_expression`, if any, for an access discriminant shall be convertible to the anonymous access type of the discriminant (see Section 5.6 [4.6], page 219).

17

A `discriminant_specification` declares a discriminant; the `subtype_mark` denotes its subtype unless it is an access discriminant, in which case the discriminant's subtype is the anonymous `access-to-variable` subtype defined by the `access_definition`.

18

For a type defined by a `derived_type_definition`, each discriminant of the parent type is either inherited, constrained to equal some new discriminant of the derived type, or constrained to the value of an expression. When inherited or constrained to equal some new discriminant, the parent discriminant and the discriminant of the derived type are said to `<correspond>`. Two discriminants also correspond if there is some common discriminant to which they both correspond. A discriminant corresponds to itself as well. If a discriminant of a parent type is constrained to a specific value by a `derived_type_definition`, then that discriminant is said to be `<specified>` by that `derived_type_definition`.

19

A constraint that appears within the definition of a discriminated type `<depends on a discriminant>` of the type if it names the discriminant as a bound or discriminant value. A `component_definition` depends on a discriminant if its constraint depends on the discriminant, or on a discriminant that corresponds to it.

20

A component `<depends on a discriminant>` if:

21

- Its `component_definition` depends on the discriminant; or

22

- It is declared in a `variant_part` that is governed by the discriminant; or

23

- It is a component inherited as part of a `derived_type_definition`, and the constraint of the `<parent_>subtype_indication` depends on the discriminant; or

24

- It is a subcomponent of a component that depends on the discriminant.

25

Each value of a discriminated type includes a value for each component of the type that does not depend on a discriminant; this includes the discriminants themselves. The values of discriminants determine which other component values are present in the value of the discriminated type.

26

A type declared with a `known_discriminant_part` is said to have `<known discriminants>`; its first subtype is unconstrained. A type declared with an `unknown_discriminant_part` is said to have `<unknown discriminants>`. A type declared without a `discriminant_part` has no discriminants, unless it is a derived type; if derived, such a type has the same sort

of discriminants (known, unknown, or none) as its parent (or ancestor) type. A tagged class-wide type also has unknown discriminants. Any subtype of a type with unknown discriminants is an unconstrained and indefinite subtype (see Section 4.2 [3.2], page 50, and Section 4.3 [3.3], page 58).

Dynamic Semantics

27/2

For an access discriminant, its `access_definition` is elaborated when the value of the access discriminant is defined: by evaluation of its `default_expression`, by elaboration of a `discriminant_constraint`, or by an assignment that initializes the enclosing object.

NOTES

28

52 If a discriminated type has `default_expressions` for its discriminants, then unconstrained variables of the type are permitted, and the values of the discriminants can be changed by an assignment to such a variable. If defaults are not provided for the discriminants, then all variables of the type are constrained, either by explicit constraint or by their initial value; the values of the discriminants of such a variable cannot be changed after initialization.

29

53 The `default_expression` for a discriminant of a type is evaluated when an object of an unconstrained subtype of the type is created.

30

54 Assignment to a discriminant of an object (after its initialization) is not allowed, since the name of a discriminant is a constant; neither `assignment_statements` nor assignments inherent in passing as an in out or out parameter are allowed. Note however that the value of a discriminant can be changed by assigning to the enclosing object, presuming it is an unconstrained variable.

31

55 A discriminant that is of a named access type is not called an access discriminant; that term is used only for discriminants defined by an `access_definition`.

Examples

32

<Examples of discriminated types:>

33

```
type Buffer(Size : Buffer_Size := 100) is      --< see Section 4.5.4
[3.5.4], page 95>
  record
```

```

        Pos    : Buffer_Size := 0;
        Value  : String(1 .. Size);
    end record;

```

34

```

type Matrix_Rec(Rows, Columns : Integer) is
    record
        Mat : Matrix(1 .. Rows, 1 .. Columns);           --< see Section 4.6
    [3.6], page 114>
    end record;

```

35

```

type Square(Side : Integer) is new
    Matrix_Rec(Rows => Side, Columns => Side);

```

36

```

type Double_Square(Number : Integer) is
    record
        Left   : Square(Number);
        Right  : Square(Number);
    end record;

```

37/2

```

task type Worker(Prio : System.Priority; Buf : access Buffer) is
    --< discriminants used to parameterize the task type (see Section 10.1
    [9.1], page 329)>
    pragma Priority(Prio); --< see Section 18.1 [D.1], page 975>
    entry Fill;
    entry Drain;
end Worker;

```

4.7.1 3.7.1 Discriminant Constraints

1

A `discriminant_constraint` specifies the values of the discriminants for a given discriminated type.

Syntax

2

```

discriminant_constraint ::=
    (discriminant_association {, discriminant_association})

```

3

```

discriminant_association ::=
    [<discriminant_>selector_name { | <discriminant_>selector_name } =>] expression

```

4

A `discriminant_association` is said to be `<named>` if it has one or more `<discriminant_selector_names>`; it is otherwise said to be `<positional>`. In a `discriminant_constraint`, any positional associations shall precede any named associations.

Name Resolution Rules

5

Each `selector_name` of a named `discriminant_association` (see [S0065], page 127) shall resolve to denote a discriminant of the subtype being constrained; the discriminants so named are the `<associated discriminants>` of the named association. For a positional association, the `<associated discriminant>` is the one whose `discriminant_specification` (see [S0062], page 123) occurred in the corresponding position in the `known_discriminant_part` (see [S0061], page 123) that defined the discriminants of the subtype being constrained.

6

The expected type for the expression in a `discriminant_association` is that of the associated discriminant(s).

Legality Rules

7/2

A `discriminant_constraint` is only allowed in a `subtype_indication` whose `subtype_mark` denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of an access subtype, a `discriminant_constraint` (see [S0064], page 127) is illegal if the designated type has a partial view that is constrained or, for a general access subtype, has `default_expressions` for its discriminants. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), these rules apply also in the private part of an instance of a generic unit. In a generic body, this rule is checked presuming all formal access types of the generic might be general access types, and all untagged discriminated formal types of the generic might have `default_expressions` for their discriminants.

8

A named `discriminant_association` with more than one `selector_name` is allowed only if the named discriminants are all of the same type. A `discriminant_constraint` shall provide exactly one value for each discriminant of the subtype being constrained.

9

The expression associated with an access discriminant shall be of a type convertible to the anonymous access type.

Dynamic Semantics

10

A `discriminant_constraint` is `<compatible>` with an unconstrained discriminated subtype if each discriminant value belongs to the subtype of the corresponding discriminant.

11

A composite value `<satisfies>` a discriminant constraint if and only if each discriminant of the composite value has the value imposed by the discriminant constraint.

12

For the elaboration of a `discriminant_constraint`, the expressions in the discrimi-

nant_associations are evaluated in an arbitrary order and converted to the type of the associated discriminant (which might raise `Constraint_Error` — see Section 5.6 [4.6], page 219); the expression of a named association is evaluated (and converted) once for each associated discriminant. The result of each evaluation and conversion is the value imposed by the constraint for the associated discriminant.

NOTES

13

56 The rules of the language ensure that a discriminant of an object always has a value, either from explicit or implicit initialization.

Examples

14

<Examples (using types declared above in clause Section 4.7 [3.7], page 123):>

15

```
Large    : Buffer(200);  --<  constrained, always 200 characters>
                                --<  (explicit discriminant value)>
Message  : Buffer;      --<  unconstrained, initially 100 characters>■
                                --<  (default discriminant value)>
Basis    : Square(5);  --<  constrained, always 5 by 5>
Illegal  : Square;     --<  illegal, a Square has to be constrained>■
```

4.7.2 3.7.2 Operations of Discriminated Types

1

If a discriminated type has `default_expressions` for its discriminants, then unconstrained variables of the type are permitted, and the discriminants of such a variable can be changed by assignment to the variable. For a formal parameter of such a type, an attribute is provided to determine whether the corresponding actual parameter is constrained or unconstrained.

Static Semantics

2

For a prefix `A` that is of a discriminated type (after any implicit dereference), the following attribute is defined:

3

`A'Constrained`

Yields the value
True if `A` denotes a
constant, a value, or a
constrained variable,
and False otherwise.

Erroneous Execution

4

The execution of a construct is erroneous if the construct has a constituent that is a name denoting a subcomponent that depends on discriminants, and the value of any of these

discriminants is changed by this execution between evaluating the name and the last use (within this execution) of the subcomponent denoted by the name.

4.8 3.8 Record Types

1

A record object is a composite object consisting of named components. The value of a record object is a composite value consisting of the values of the components.

Syntax

2

```
record_type_definition ::= [[abstract] tagged] [limited] record_definition
```

3

```
record_definition ::=  
  record  
    component_list  
  end record  
| null record
```

4

```
component_list ::=  
  component_item {component_item}  
| {component_item} variant_part  
| null;
```

5/1

```
component_item ::= component_declaration | aspect_clause
```

6

```
component_declaration ::=  
  defining_identifier_list : component_definition [:= default_expression];  
Name Resolution Rules
```

7

The expected type for the `default_expression`, if any, in a `component_declaration` is the type of the component.

Legality Rules

8/2

<This paragraph was deleted.>

9/2

Each `component_declaration` declares a component of the record type. Besides components declared by `component_declarations`, the components of a record type include any components declared by `discriminant_specifications` of the record type declaration. The identifiers of all components of a record type shall be distinct.

10

Within a `type_declaration`, a name that denotes a component, protected subprogram, or entry of the type is allowed only in the following cases:

11

- A name that denotes any component, protected subprogram, or entry is allowed within a representation item that occurs within the declaration of the composite type.

12

- A name that denotes a noninherited discriminant is allowed within the declaration of the type, but not within the `discriminant_part`. If the discriminant is used to define the constraint of a component, the bounds of an entry family, or the constraint of the parent subtype in a `derived_type_definition` then its name shall appear alone as a `direct_name` (not as part of a larger expression or expanded name). A discriminant shall not be used to define the constraint of a scalar component.

13

If the name of the current instance of a type (see Section 9.6 [8.6], page 324) is used to define the constraint of a component, then it shall appear as a `direct_name` that is the prefix of an `attribute_reference` whose result is of an access type, and the `attribute_reference` shall appear alone.

Static Semantics

13.1/2

If a `record_type_definition` includes the reserved word `limited`, the type is called an <explicitly limited record> type.

14

The `component_definition` of a `component_declaration` defines the (nominal) subtype of the component. If the reserved word `aliased` appears in the `component_definition`, then the component is aliased (see Section 4.10 [3.10], page 156).

15

If the `component_list` of a record type is defined by the reserved word `null` and there are no discriminants, then the record type has no components and all records of the type are <null records>. A `record_definition` of `null record` is equivalent to `record null; end record`.

Dynamic Semantics

16

The elaboration of a `record_type_definition` creates the record type and its first subtype, and consists of the elaboration of the `record_definition`. The elaboration of a `record_definition` consists of the elaboration of its `component_list`, if any.

17

The elaboration of a `component_list` consists of the elaboration of the `component_items` and `variant_part`, if any, in the order in which they appear. The elaboration of a `component_declaration` consists of the elaboration of the `component_definition`.

18/2

Within the definition of a composite type, if a `component_definition` or `discrete_subtype_definition` (see Section 10.5.2 [9.5.2], page 347) includes a name that denotes

a discriminant of the type, or that is an `attribute_reference` whose prefix denotes the current instance of the type, the expression containing the name is called a `<per-object expression>`, and the constraint or range being defined is called a `<per-object constraint>`. For the elaboration of a `component_definition` of a `component_declaration` or the `discrete_subtype_definition` (see [S0055], page 114) of an `entry_declaration` (see [S0200], page 347) for an entry family (see Section 10.5.2 [9.5.2], page 347), if the component subtype is defined by an `access_definition` or if the constraint or range of the `subtype_indication` or `discrete_subtype_definition` (see [S0055], page 114) is not a `per-object constraint`, then the `access_definition`, `subtype_indication`, or `discrete_subtype_definition` (see [S0055], page 114) is elaborated. On the other hand, if the constraint or range is a `per-object constraint`, then the elaboration consists of the evaluation of any included expression that is not part of a `per-object expression`. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant.

18.1/1

When a `per-object constraint` is elaborated (as part of creating an object), each `per-object expression` of the constraint is evaluated. For other expressions, the values determined during the elaboration of the `component_definition` (see [S0056], page 114) or `entry_declaration` (see [S0200], page 347) are used. Any checks associated with the enclosing `subtype_indication` or `discrete_subtype_definition` are performed, including the subtype compatibility check (see Section 4.2.2 [3.2.2], page 55), and the associated subtype is created.

NOTES

19

57 A `component_declaration` with several identifiers is equivalent to a sequence of single `component_declarations`, as explained in Section 4.3.1 [3.3.1], page 61.

20

58 The `default_expression` of a record component is only evaluated upon the creation of a `default-initialized` object of the record type (presuming the object has the component, if it is in a `variant_part` — see Section 4.3.1 [3.3.1], page 61).

21

59 The subtype defined by a `component_definition` (see Section 4.6 [3.6], page 114) has to be a definite subtype.

22

60 If a record type does not have a `variant_part`, then the same components are present in all values of the type.

23

61 A record type is limited if it has the reserved word `limited` in its definition, or if any of its components are limited (see Section 8.5 [7.5], page 292).

24

62 The predefined operations of a record type include membership tests, qualification, and explicit conversion. If the record type is nonlimited, they also include assignment and the predefined equality operators.

25/2

63 A component of a record can be named with a `selected_component`. A value of a record can be specified with a `record_aggregate`.

Examples

26

<Examples of record type declarations: >

27

```
type Date is
  record
    Day   : Integer range 1 .. 31;
    Month : Month_Name;
    Year  : Integer range 0 .. 4000;
  end record;
```

28

```
type Complex is
  record
    Re : Real := 0.0;
    Im : Real := 0.0;
  end record;
```

29

<Examples of record variables: >

30

```
Tomorrow, Yesterday : Date;
A, B, C : Complex;
```

31

--< both components of A, B, and C are implicitly initialized to zero >■

4.8.1 3.8.1 Variant Parts and Discrete Choices

1

A record type with a `variant_part` specifies alternative lists of components. Each variant defines the components for the value or values of the discriminant covered by its `discrete_choice_list`.

Syntax

2

```
variant_part ::=
  case <discriminant_>direct_name is
    variant
    {variant}
  end case;
```

3

```
variant ::=
  when discrete_choice_list =>
    component_list
```

4

```
discrete_choice_list ::= discrete_choice { | discrete_choice }
```

5

```
discrete_choice ::= expression | discrete_range | others
```

Name Resolution Rules

6

The `<discriminant_>direct_name` shall resolve to denote a discriminant (called the `<discriminant of the variant_part>`) specified in the `known_discriminant_part` of the `full_type_declaration` that contains the `variant_part`. The expected type for each `discrete_choice` in a variant is the type of the discriminant of the `variant_part`.

Legality Rules

7

The discriminant of the `variant_part` shall be of a discrete type.

8

The expressions and `discrete_ranges` given as `discrete_choices` in a `variant_part` shall be static. The `discrete_choice others` shall appear alone in a `discrete_choice_list`, and such a `discrete_choice_list`, if it appears, shall be the last one in the enclosing construct.

9

A `discrete_choice` is defined to `<cover a value>` in the following cases:

10

- A `discrete_choice` that is an expression covers a value if the value equals the value of the expression converted to the expected type.

11

- A `discrete_choice` that is a `discrete_range` covers all values (possibly none) that belong to the range.

12

- The `discrete_choice others` covers all values of its expected type that are not covered by previous `discrete_choice_lists` of the same construct.

13

A `discrete_choice_list` covers a value if one of its `discrete_choices` covers the value.

14

The possible values of the discriminant of a `variant_part` shall be covered as follows:

15

- If the discriminant is of a static constrained scalar subtype, then each non-`others` `discrete_choice` (see [S0074], page 134) shall cover only values in that subtype, and each value of that subtype shall be covered by some `discrete_choice` (see [S0074], page 134) (either explicitly or by `others`);

16

- If the type of the discriminant is a descendant of a generic formal scalar type then the `variant_part` shall have an `others` `discrete_choice`;

17

- Otherwise, each value of the base range of the type of the discriminant shall be covered (either explicitly or by `others`).

18

Two distinct `discrete_choices` of a `variant_part` shall not cover the same value.

Static Semantics

19

If the `component_list` of a `variant` is specified by `null`, the `variant` has no components.

20

The discriminant of a `variant_part` is said to <govern> the `variant_part` and its variants. In addition, the discriminant of a derived type governs a `variant_part` and its variants if it corresponds (see Section 4.7 [3.7], page 123) to the discriminant of the `variant_part`.

Dynamic Semantics

21

A record value contains the values of the components of a particular `variant` only if the value of the discriminant governing the `variant` is covered by the `discrete_choice_list` of the `variant`. This rule applies in turn to any further `variant` that is, itself, included in the `component_list` of the given `variant`.

22

The elaboration of a variant_part consists of the elaboration of the component_list of each variant in the order in which they appear.

Examples

23

<Example of record type with a variant part: >

24

```
type Device is (Printer, Disk, Drum);
type State is (Open, Closed);
```

25

```
type Peripheral(Unit : Device := Disk) is
  record
    Status : State;
    case Unit is
      when Printer =>
        Line_Count : Integer range 1 .. Page_Size;
      when others =>
        Cylinder : Cylinder_Index;
        Track : Track_Number;
    end case;
  end record;
```

26

<Examples of record subtypes:>

27

```
subtype Drum_Unit is Peripheral(Drum);
subtype Disk_Unit is Peripheral(Disk);
```

28

<Examples of constrained record variables:>

29

```
Writer : Peripheral(Unit => Printer);
Archive : Disk_Unit;
```

4.9 3.9 Tagged Types and Type Extensions

1

Tagged types and type extensions support object-oriented programming, based on inheritance with extension and run-time polymorphism via <dispatching operations>.

Static Semantics

2/2

A record type or private type that has the reserved word tagged in its declaration is called

a <tagged> type. In addition, an interface type is a tagged type, as is a task or protected type derived from an interface (see Section 4.9.4 [3.9.4], page 152). When deriving from a tagged type, as for any derived type, additional primitive subprograms may be defined, and inherited primitive subprograms may be overridden. The derived type is called an <extension> of its ancestor types, or simply a <type extension>.

2.1/2

Every type extension is also a tagged type, and is a <record extension> or a <private extension> of some other tagged type, or a non-interface synchronized tagged type (see Section 4.9.4 [3.9.4], page 152). A record extension is defined by a `derived_type_definition` with a `record_extension_part` (see Section 4.9.1 [3.9.1], page 143), which may include the definition of additional components. A private extension, which is a partial view of a record extension or of a synchronized tagged type, can be declared in the visible part of a package (see Section 8.3 [7.3], page 283) or in a generic formal part (see Section 13.5.1 [12.5.1], page 462).

3

An object of a tagged type has an associated (run-time) <tag> that identifies the specific tagged type used to create the object originally. The tag of an operand of a class-wide tagged type `<T>'Class` controls which subprogram body is to be executed when a primitive subprogram of type `<T>` is applied to the operand (see Section 4.9.2 [3.9.2], page 145); using a tag to control which body to execute is called <dispatching>.

4/2

The tag of a specific tagged type identifies the `full_type_declaration` of the type, and for a type extension, is sufficient to uniquely identify the type among all descendants of the same ancestor. If a declaration for a tagged type occurs within a `generic_package_declaration`, then the corresponding type declarations in distinct instances of the generic package are associated with distinct tags. For a tagged type that is local to a generic package body and with all of its ancestors (if any) also local to the generic body, the language does not specify whether repeated instantiations of the generic body result in distinct tags.

5

The following language-defined library package exists:

6/2

```
package Ada.Tags is
  pragma Preelaborate(Tags);
  type
    Tag is private;
  pragma Preelaborable_Initialization(Tag);
```

6.1/2

```
No_Tag : constant Tag;
```

7/2

```

    function
Expanded_Name(T : Tag) return String;
    function
Wide_Expanded_Name(T : Tag) return Wide_String;
    function
Wide_Wide_Expanded_Name(T : Tag) return Wide_Wide_String;
    function
External_Tag(T : Tag) return String;
    function
Internal_Tag(External : String) return Tag;

```

7.1/2

```

    function
Descendant_Tag(External : String; Ancestor : Tag) return Tag;
    function
Is_Descendant_At_Same_Level(Descendant, Ancestor : Tag)
    return Boolean;

```

7.2/2

```

    function
Parent_Tag (T : Tag) return Tag;

```

7.3/2

```

    type
Tag_Array is array (Positive range <>) of Tag;

```

7.4/2

```

    function
Interface_Ancestor_Tags (T : Tag) return Tag_Array;

```

8

```

Tag_Error : exception;

```

9

```

private
... -- <not specified by the language>
end Ada.Tags;

```

9.1/2

No_Tag is the default initial value of type Tag.

10/2

The function Wide_Wide_Expanded_Name returns the full expanded name of the first sub-type of the specific type identified by the tag, in upper case, starting with a root library

unit. The result is implementation defined if the type is declared within an unnamed block_statement.

10.1/2

The function `Expanded_Name` (respectively, `Wide_Expanded_Name`) returns the same sequence of graphic characters as that defined for `Wide_Wide_Expanded_Name`, if all the graphic characters are defined in `Character` (respectively, `Wide_Character`); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by `Wide_Wide_Expanded_Name` for the same value of the argument.

11

The function `External_Tag` returns a string to be used in an external representation for the given tag. The call `External_Tag(S'Tag)` is equivalent to the attribute_reference `S'External_Tag` (see Section 14.3 [13.3], page 486).

11.1/2

The string returned by the functions `Expanded_Name`, `Wide_Expanded_Name`, `Wide_Wide_Expanded_Name`, and `External_Tag` has lower bound 1.

12/2

The function `Internal_Tag` returns a tag that corresponds to the given external tag, or raises `Tag_Error` if the given string is not the external tag for any specific type of the partition. `Tag_Error` is also raised if the specific type identified is a library-level type whose tag has not yet been created (see Section 14.14 [13.14], page 550).

12.1/2

The function `Descendant_Tag` returns the (internal) tag for the type that corresponds to the given external tag and is both a descendant of the type identified by the `Ancestor` tag and has the same accessibility level as the identified ancestor. `Tag_Error` is raised if `External` is not the external tag for such a type. `Tag_Error` is also raised if the specific type identified is a library-level type whose tag has not yet been created.

12.2/2

The function `Is_Descendant_At_Same_Level` returns `True` if the `Descendant` tag identifies a type that is both a descendant of the type identified by `Ancestor` and at the same accessibility level. If not, it returns `False`.

12.3/2

The function `Parent_Tag` returns the tag of the parent type of the type whose tag is `T`. If the type does not have a parent type (that is, it was not declared by a `derived_type_declaration`), then `No_Tag` is returned.

12.4/2

The function `Interface_Ancestor_Tags` returns an array containing the tag of each interface ancestor type of the type whose tag is `T`, other than `T` itself. The lower bound of the returned array is 1, and the order of the returned tags is unspecified. Each tag appears in the result exactly once. If the type whose tag is `T` has no interface ancestors, a null array is returned.

13

For every subtype `S` of a tagged type `<T>` (specific or class-wide), the following attributes are defined:

14

`S'Class`

S'Class denotes a subtype of the class-wide type (called <T>'Class in this International Standard) for the class rooted at <T> (or if S already denotes a class-wide subtype, then S'Class is the same as S).

15

S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type <T> belong to S.

16

S'Tag

S'Tag denotes the tag of the type <T> (or if <T> is class-wide, the tag of the root type of the corresponding class). The value of this attribute is of type Tag.

17

Given a prefix X that is of a class-wide tagged type (after any implicit dereference), the following attribute is defined:

18

X'Tag

X'Tag denotes the tag of X. The value of this attribute is of type Tag.

18.1/2

The following language-defined generic function exists:

18.2/2

```

generic
  type T (<>) is abstract tagged limited private;
  type Parameters (<>) is limited private;
  with function Constructor (Params : not null access Parameters)
    return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
  (The_Tag : Tag;
   Params  : not null access Parameters) return T'Class;
pragma Preelaborate(Generic_Dispatching_Constructor);
pragma Convention(Intrinsic, Generic_Dispatching_Constructor);

```

18.3/2

Tags.Generic_Dispatching_Constructor provides a mechanism to create an object of an appropriate type from just a tag value. The function Constructor is expected to create the object given a reference to an object of type Parameters.

Dynamic Semantics

19

The tag associated with an object of a tagged type is determined as follows:

20

- The tag of a stand-alone object, a component, or an aggregate of a specific tagged type <T> identifies <T>.

21

- The tag of an object created by an allocator for an access type with a specific designated tagged type <T>, identifies <T>.

22

- The tag of an object of a class-wide tagged type is that of its initialization expression.

23

- The tag of the result returned by a function whose result type is a specific tagged type <T> identifies <T>.

24/2

- The tag of the result returned by a function with a class-wide result type is that of the return object.

25

The tag is preserved by type conversion and by parameter passing. The tag of a value is the tag of the associated object (see Section 7.2 [6.2], page 260).

25.1/2

Tag_Error is raised by a call of Descendant_Tag, Expanded_Name, External_Tag,

Interface_Ancestors_Tag, Is_Descendant_At_Same_Level, or Parent_Tag if any tag passed is No_Tag.

25.2/2

An instance of Tags.Generic_Dispatching_Constructor raises Tag_Error if The_Tag does not represent a concrete descendant of T or if the innermost master (see Section 8.6.1 [7.6.1], page 299) of this descendant is not also a master of the instance. Otherwise, it dispatches to the primitive function denoted by the formal Constructor for the type identified by The_Tag, passing Params, and returns the result. Any exception raised by the function is propagated.

Erroneous Execution

25.3/2

If an internal tag provided to an instance of Tags.Generic_Dispatching_Constructor or to any subprogram declared in package Tags identifies either a type that is not library-level and whose tag has not been created (see Section 14.14 [13.14], page 550), or a type that does not exist in the partition at the time of the call, then execution is erroneous.

Implementation Permissions

26/2

The implementation of Internal_Tag and Descendant_Tag may raise Tag_Error if no specific type corresponding to the string External passed as a parameter exists in the partition at the time the function is called, or if there is no such type whose innermost master is a master of the point of the function call.

Implementation Advice

26.1/2

Internal_Tag should return the tag of a type whose innermost master is the master of the point of the function call.

NOTES

27

64 A type declared with the reserved word tagged should normally be declared in a package_specification, so that new primitive subprograms can be declared for it.

28

65 Once an object has been created, its tag never changes.

29

66 Class-wide types are defined to have unknown discriminants (see Section 4.7 [3.7], page 123). This means that objects of a class-wide type have to be explicitly initialized (whether created by an object_declaration or an allocator), and that aggregates have to be explicitly qualified with a specific type when their expected type is class-wide.

30/2

<This paragraph was deleted.>

30.1/2

67 The capability provided by `Tags.Generic.Dispatching_Constructor` is sometimes known as a <factory>. ■

Examples

31

<Examples of tagged record types:>

32

```
type Point is tagged
  record
    X, Y : Real := 0.0;
  end record;
```

33

```
type Expression is tagged null record;
  --< Components will be added by each extension>
```

4.9.1 3.9.1 Type Extensions

1/2

Every type extension is a tagged type, and is a <record extension> or a <private extension> of some other tagged type, or a non-interface synchronized tagged type..

Syntax

2

```
record_extension_part ::= with record_definition
  Legality Rules
```

3/2

The parent type of a record extension shall not be a class-wide type nor shall it be a synchronized tagged type (see Section 4.9.4 [3.9.4], page 152). If the parent type or any progenitor is nonlimited, then each of the components of the `record_extension_part` shall be nonlimited. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), these rules apply also in the private part of an instance of a generic unit.

4/2

Within the body of a generic unit, or the body of any of its descendant library units, a tagged type shall not be declared as a descendant of a formal type declared within the formal part of the generic unit.

Static Semantics

4.1/2

A record extension is a <null extension> if its declaration has no `known_discriminant_part` and its `record_extension_part` includes no `component_declarations`.

Dynamic Semantics

5

The elaboration of a `record_extension_part` consists of the elaboration of the `record_definition`.

NOTES

6

68 The term "type extension" refers to a type as a whole. The term "extension part" refers to the piece of text that defines the additional components (if any) the type extension has relative to its specified ancestor type.

7/2

69 When an extension is declared immediately within a body, primitive subprograms are inherited and are overridable, but new primitive subprograms cannot be added.

8

70 A name that denotes a component (including a discriminant) of the parent type is not allowed within the `record_extension_part`. Similarly, a name that denotes a component defined within the `record_extension_part` is not allowed within the `record_extension_part`. It is permissible to use a name that denotes a discriminant of the record extension, providing there is a new `known_discriminant_part` in the enclosing type declaration. (The full rule is given in Section 4.8 [3.8], page 130.)

9

71 Each visible component of a record extension has to have a unique name, whether the component is (visibly) inherited from the parent type or declared in the `record_extension_part` (see Section 9.3 [8.3], page 308).

Examples

10

<Examples of record extensions (of types defined above in Section 4.9 [3.9], page 136):>

11

```
type Painted_Point is new Point with
  record
    Paint : Color := White;
  end record;
  --< Components X and Y are inherited>
```

12

```
Origin : constant Painted_Point := (X | Y => 0.0, Paint => Black);
```


13

```
type Literal is new Expression with
  record
    Value : Real;
  end record;
  --< a leaf in an Expression tree>
```

14

```
type Expr_Ptr is access all Expression'Class;
  --< see Section 4.10 [3.10], page 156>
```

15

```
type Binary_Operation is new Expression with
  record
    Left, Right : Expr_Ptr;
  end record;
  --< an internal node in an Expression tree>
```

16

```
type Addition is new Binary_Operation with null record;
type Subtraction is new Binary_Operation with null record;
  --< No additional components needed for these extensions>
```

17

```
Tree : Expr_Ptr :=
  new Addition'(
    Left => new Literal'(Value => 5.0),
    Right => new Subtraction'(
      Left => new Literal'(Value => 13.0),
      Right => new Literal'(Value => 7.0)));
  --< A tree representation of "5.0 + (13.0-7.0)">
```

4.9.2 3.9.2 Dispatching Operations of Tagged Types

1/2

The primitive subprograms of a tagged type, the subprograms declared by `formal_abstract_subprogram_declaration` (see [S0277], page 471)s, and the stream attributes of a specific tagged type that are available (see Section 14.13.2 [13.13.2], page 540) at the end of the declaration list where the type is declared are called `<dispatching operations>`. A dispatching operation can be called using a statically determined `<controlling>` tag, in which case the body to be executed is determined at compile time. Alternatively, the controlling tag can be dynamically determined, in which case the call `<dispatches>` to a body that is determined at run time; such a call is termed a `<dispatching call>`. As explained below, the properties of the operands and the context of a particular call on a dispatching operation determine how the controlling tag is determined, and hence whether or not the call is a dispatching call. Run-time polymorphism is achieved when a dispatching operation is called by a dispatching call.

2/2

A <call on a dispatching operation> is a call whose name or prefix denotes the declaration of a dispatching operation. A <controlling operand> in a call on a dispatching operation of a tagged type <T> is one whose corresponding formal parameter is of type <T> or is of an anonymous access type with designated type <T>; the corresponding formal parameter is called a <controlling formal parameter>. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type <T>, then the call has a <controlling result> — the context of the call can control the dispatching. Similarly, if the call is to a function with access result type designating <T>, then the call has a <controlling access result>, and the context can similarly control dispatching.

3

A name or expression of a tagged type is either <statically> tagged, <dynamically> tagged, or <tag indeterminate>, according to whether, when used as a controlling operand, the tag that controls dispatching is determined statically by the operand's (specific) type, dynamically by its tag at run time, or from context. A qualified_expression or parenthesized expression is statically, dynamically, or indeterminately tagged according to its operand. For other kinds of names and expressions, this is determined as follows:

4/2

- The name or expression is <statically tagged> if it is of a specific tagged type and, if it is a call with a controlling result or controlling access result, it has at least one statically tagged controlling operand;

5/2

- The name or expression is <dynamically tagged> if it is of a class-wide type, or it is a call with a controlling result or controlling access result and at least one dynamically tagged controlling operand;

6/2

- The name or expression is <tag indeterminate> if it is a call with a controlling result or controlling access result, all of whose controlling operands (if any) are tag indeterminate.

7/1

A type_conversion is statically or dynamically tagged according to whether the type determined by the subtype_mark is specific or class-wide, respectively. For an object that is designated by an expression whose expected type is an anonymous access-to-specific tagged type, the object is dynamically tagged if the expression, ignoring enclosing parentheses, is of the form X'Access, where X is of a class-wide type, or is of the form new T'(...), where T denotes a class-wide subtype. Otherwise, the object is statically or dynamically tagged according to whether the designated type of the type of the expression is specific or class-wide, respectively.

8

A call on a dispatching operation shall not have both dynamically tagged and statically tagged controlling operands.

9/1

If the expected type for an expression or name is some specific tagged type, then the expression or name shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. Similarly, if the expected type for an expression is an anonymous access-to-specific tagged type, then the object designated by the expression shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation.

10/2

In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see Section 7.1 [6.1], page 255), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. The convention of an inherited dispatching operation is the convention of the corresponding primitive operation of the parent or progenitor type. The default convention of a dispatching operation that overrides an inherited primitive operation is the convention of the inherited operation; if the operation overrides multiple inherited operations, then they shall all have the same convention. An explicitly declared dispatching operation shall not be of convention Intrinsic.

11/2

The `default_expression` for a controlling formal parameter of a dispatching operation shall be tag indeterminate.

11.1/2

If a dispatching operation is defined by a `subprogram_renaming_declaration` or the instantiation of a generic subprogram, any access parameter of the renamed subprogram or the generic subprogram that corresponds to a controlling access parameter of the dispatching operation, shall have a subtype that excludes null.

12

A given subprogram shall not be a dispatching operation of two or more distinct tagged types.

13

The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see Section 14.14 [13.14], page 550). For example, new dispatching operations cannot be added after objects or values of the type exist, nor after deriving a record extension from it, nor after a body.

Dynamic Semantics

14

For the execution of a call on a dispatching operation of a type `<T>`, the `<controlling tag value>` determines which subprogram body is executed. The controlling tag value is defined as follows:

15

- If one or more controlling operands are statically tagged, then the controlling tag value is <statically determined> to be the tag of <T>.

16

- If one or more controlling operands are dynamically tagged, then the controlling tag value is not statically determined, but is rather determined by the tags of the controlling operands. If there is more than one dynamically tagged controlling operand, a check is made that they all have the same tag. If this check fails, `Constraint_Error` is raised unless the call is a `function_call` whose name denotes the declaration of an equality operator (predefined or user defined) that returns `Boolean`, in which case the result of the call is defined to indicate inequality, and no `subprogram_body` is executed. This check is performed prior to evaluating any tag-indeterminate controlling operands.

17/2

- If all of the controlling operands (if any) are tag-indeterminate, then:

18/2

- If the call has a controlling result or controlling access result and is itself, or designates, a (possibly parenthesized or qualified) controlling operand of an enclosing call on a dispatching operation of a descendant of type <T>, then its controlling tag value is determined by the controlling tag value of this enclosing call;

18.1/2

- If the call has a controlling result or controlling access result and (possibly parenthesized, qualified, or dereferenced) is the expression of an `assignment_statement` whose target is of a class-wide type, then its controlling tag value is determined by the target;

19

- Otherwise, the controlling tag value is statically determined to be the tag of type <T>.

20/2

For the execution of a call on a dispatching operation, the action performed is determined by the properties of the corresponding dispatching operation of the specific type identified by

the controlling tag value. If the corresponding operation is explicitly declared for this type, even if the declaration occurs in a private part, then the action comprises an invocation of the explicit body for the operation. If the corresponding operation is implicitly declared for this type:

20.1/2

- if the operation is implemented by an entry or protected subprogram (see Section 10.1 [9.1], page 329, and Section 10.4 [9.4], page 337), then the action comprises a call on this entry or protected subprogram, with the target object being given by the first actual parameter of the call, and the actual parameters of the entry or protected subprogram being given by the remaining actual parameters of the call, if any;

20.2/2

- otherwise, the action is the same as the action for the corresponding operation of the parent type.

NOTES

21

72 The body to be executed for a call on a dispatching operation is determined by the tag; it does not matter whether that tag is determined statically or dynamically, and it does not matter whether the subprogram's declaration is visible at the place of the call.

22/2

73 This subclause covers calls on dispatching subprograms of a tagged type. Rules for tagged type membership tests are described in Section 5.5.2 [4.5.2], page 206. Controlling tag determination for an assignment_statement is described in Section 6.2 [5.2], page 242.

23

74 A dispatching call can dispatch to a body whose declaration is not visible at the place of the call.

24

75 A call through an access-to-subprogram value is never a dispatching call, even if the access value designates a dispatching operation. Similarly a call whose prefix denotes a subprogram_renaming_declaration cannot be a dispatching call unless the renaming itself is the declaration of a primitive subprogram.

4.9.3 3.9.3 Abstract Types and Subprograms

1/2

An <abstract type> is a tagged type intended for use as an ancestor of other types, but

which is not allowed to have objects of its own. An <abstract subprogram> is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body.

Syntax

1.1/2

```
abstract_subprogram_declaration ::=  
    [overriding_indicator]  
    subprogram_specification is abstract;
```

Static Semantics

1.2/2

Interface types (see Section 4.9.4 [3.9.4], page 152) are abstract types. In addition, a tagged type that has the reserved word `abstract` in its declaration is an abstract type. The class-wide type (see Section 4.4.1 [3.4.1], page 72) rooted at an abstract type is not itself an abstract type.

Legality Rules

2/2

Only a tagged type shall have the reserved word `abstract` in its declaration.

3/2

A subprogram declared by an `abstract_subprogram_declaration` (see [S0076], page 150) or a `formal_abstract_subprogram_declaration` (see [S0277], page 471) (see Section 13.6 [12.6], page 470) is an <abstract subprogram>. If it is a primitive subprogram of a tagged type, then the tagged type shall be abstract.

4/2

If a type has an implicitly declared primitive subprogram that is inherited or is the pre-defined equality operator, and the corresponding primitive subprogram of the parent or ancestor type is abstract or is a function with a controlling access result, or if a type other than a null extension inherits a function with a controlling result, then:

5/2

- If the type is abstract or untagged, the implicitly declared subprogram is <abstract>.

6/2

- Otherwise, the subprogram shall be overridden with a nonabstract subprogram or, in the case of a private extension inheriting a function with a controlling result, have a full type that is a null extension; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part. Such a subprogram is said to <require overriding>. However, if the type is a generic formal type, the subprogram need not be overridden for the formal type itself; a nonabstract version will necessarily be provided by the actual type.

7

A call on an abstract subprogram shall be a dispatching call; nondispatching calls to an abstract subprogram are not allowed.

8

The type of an aggregate, or of an object created by an `object_declaration` or an allocator, or a generic formal object of mode `in`, shall not be abstract. The type of the target of an assignment operation (see Section 6.2 [5.2], page 242) shall not be abstract. The type of a component shall not be abstract. If the result type of a function is abstract, then the function shall be abstract.

9

If a partial view is not abstract, the corresponding full view shall not be abstract. If a generic formal type is abstract, then for each primitive subprogram of the formal that is not abstract, the corresponding primitive subprogram of the actual shall not be abstract.

10

For an abstract type declared in a visible part, an abstract primitive subprogram shall not be declared in the private part, unless it is overriding an abstract subprogram implicitly declared in the visible part. For a tagged type declared in a visible part, a primitive function with a controlling result shall not be declared in the private part, unless it is overriding a function implicitly declared in the visible part.

11/2

A generic actual subprogram shall not be an abstract subprogram unless the generic formal subprogram is declared by a `formal_abstract_subprogram_declaration`. The prefix of an `attribute_reference` for the `Access`, `Unchecked_Access`, or `Address` attributes shall not denote an abstract subprogram.

Dynamic Semantics

11.1/2

The elaboration of an `abstract_subprogram_declaration` has no effect.

NOTES

12

76 Abstractness is not inherited; to declare an abstract type, the reserved word `abstract` has to be used in the declaration of the type extension.

13

77 A class-wide type is never abstract. Even if a class is rooted at an abstract type, the class-wide type for the class is not abstract, and an object of the class-wide type can be created; the tag of such an object will identify some nonabstract type in the class.

Examples

14

<Example of an abstract type representing a set of natural numbers:>

15

```
package Sets is
  subtype Element_Type is Natural;
  type Set is abstract tagged null record;
```

```

function Empty return Set is abstract;
function Union(Left, Right : Set) return Set is abstract;
function Intersection(Left, Right : Set) return Set is abstract;
function Unit_Set(Element : Element_Type) return Set is abstract;
procedure Take(Element : out Element_Type;
               From : in out Set) is abstract;
end Sets;

```

NOTES

16

78 <Notes on the example:> Given the above abstract type, one could then derive various (nonabstract) extensions of the type, representing alternative implementations of a set. One might use a bit vector, but impose an upper bound on the largest element representable, while another might use a hash table, trading off space for flexibility.

4.9.4 3.9.4 Interface Types

1/2

An interface type is an abstract tagged type that provides a restricted form of multiple inheritance. A tagged type, task type, or protected type may have one or more interface types as ancestors.

Syntax

2/2

```

interface_type_definition ::=
    [limited | task | protected | synchronized] interface [and interface_list]

```

3/2

```

interface_list ::= <interface_>subtype_mark {and <interface_>subtype_mark}

```

Static Semantics

4/2

An interface type (also called an <interface>) is a specific abstract tagged type that is defined by an interface_type_definition.

5/2

An interface with the reserved word limited, task, protected, or synchronized in its definition is termed, respectively, a <limited interface>, a <task interface>, a <protected interface>, or a <synchronized interface>. In addition, all task and protected interfaces are synchronized interfaces, and all synchronized interfaces are limited interfaces.

6/2

A task or protected type derived from an interface is a tagged type. Such a tagged type is called a <synchronized> tagged type, as are synchronized interfaces and private extensions whose declaration includes the reserved word synchronized.

7/2

A task interface is an abstract task type. A protected interface is an abstract protected type.

8/2

An interface type has no components.

9/2

An `<interface_>subtype_mark` in an `interface_list` names a `<progenitor subtype>`; its type is the `<progenitor type>`. An interface type inherits user-defined primitive subprograms from each progenitor type in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see Section 4.4 [3.4], page 66).

Legality Rules

10/2

All user-defined primitive subprograms of an interface type shall be abstract subprograms or null procedures.

11/2

The type of a subtype named in an `interface_list` shall be an interface type.

12/2

A type derived from a nonlimited interface shall be nonlimited.

13/2

An interface derived from a task interface shall include the reserved word `task` in its definition; any other type derived from a task interface shall be a private extension or a task type declared by a task declaration (see Section 10.1 [9.1], page 329).

14/2

An interface derived from a protected interface shall include the reserved word `protected` in its definition; any other type derived from a protected interface shall be a private extension or a protected type declared by a protected declaration (see Section 10.4 [9.4], page 337).

15/2

An interface derived from a synchronized interface shall include one of the reserved words `task`, `protected`, or `synchronized` in its definition; any other type derived from a synchronized interface shall be a private extension, a task type declared by a task declaration, or a protected type declared by a protected declaration.

16/2

No type shall be derived from both a task interface and a protected interface.

17/2

In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), these rules apply also in the private part of an instance of a generic unit.

Dynamic Semantics

18/2

The elaboration of an `interface_type_definition` has no effect.

NOTES

19/2

79 Nonlimited interface types have predefined nonabstract equality operators. These may be overridden with user-defined abstract

equality operators. Such operators will then require an explicit overriding for any nonabstract descendant of the interface.

Examples

20/2

<Example of a limited interface and a synchronized interface extending it:>

21/2

```
type Queue is limited interface;
procedure Append(Q : in out Queue; Person : in Person_Name) is abstract;
procedure Remove_First(Q : in out Queue;
    Person : out Person_Name) is abstract;
function Cur_Count(Q : in Queue) return Natural is abstract;
function Max_Count(Q : in Queue) return Natural is abstract;
-- <See Section 4.10.1 [3.10.1], page 160 for Person_Name.>
```

22/2

```
Queue_Error : exception;
--< Append raises Queue_Error if Count(Q) = Max_Count(Q)>
--< Remove_First raises Queue_Error if Count(Q) = 0>
```

23/2

```
type Synchronized_Queue is synchronized interface and Queue; --< see Section 10.1
[9.11], page 391>
procedure Append_Wait(Q : in out Synchronized_Queue;
    Person : in Person_Name) is abstract;
procedure Remove_First_Wait(Q : in out Synchronized_Queue;
    Person : out Person_Name) is abstract;
```

24/2

...

25/2

```
procedure Transfer(From : in out Queue'Class;
    To : in out Queue'Class;
    Number : in Natural := 1) is
    Person : Person_Name;
begin
    for I in 1..Number loop
        Remove_First(From, Person);
        Append(To, Person);
    end loop;
end Transfer;
```

26/2

This defines a Queue interface defining a queue of people. (A similar design could be created to define any kind of queue simply by replacing Person_Name by an appropriate type.)

The Queue interface has four dispatching operations, Append, Remove_First, Cur_Count, and Max_Count. The body of a class-wide operation, Transfer is also shown. Every non-abstract extension of Queue must provide implementations for at least its four dispatching operations, as they are abstract. Any object of a type derived from Queue may be passed to Transfer as either the From or the To operand. The two operands need not be of the same type in any given call.

27/2

The Synchronized_Queue interface inherits the four dispatching operations from Queue and adds two additional dispatching operations, which wait if necessary rather than raising the Queue_Error exception. This synchronized interface may only be implemented by a task or protected type, and as such ensures safe concurrent access.

28/2

<Example use of the interface:>

29/2

```
type Fast_Food_Queue is new Queue with record ...;
procedure Append(Q : in out Fast_Food_Queue; Person : in Person_Name);
procedure Remove_First(Q : in out Fast_Food_Queue; Person : in Person_Name);
function Cur_Count(Q : in Fast_Food_Queue) return Natural;
function Max_Count(Q : in Fast_Food_Queue) return Natural;
```

30/2

...

31/2

```
Cashier, Counter : Fast_Food_Queue;
```

32/2

```
...
-- <Add George (see Section 4.10.1 [3.10.1], page 160) to the cashier's queue:>
Append (Cashier, George);
-- <After payment, move George to the sandwich counter queue:>
Transfer (Cashier, Counter);
...
```

33/2

An interface such as Queue can be used directly as the parent of a new type (as shown here), or can be used as a progenitor when a type is derived. In either case, the primitive operations of the interface are inherited. For Queue, the implementation of the four inherited routines must be provided. Inside the call of Transfer, calls will dispatch to the implementations of Append and Remove_First for type Fast_Food_Queue.

34/2

<Example of a task interface:>

35/2

```

type Serial_Device is task interface; --< see Section 10.1 [9.1],
page 329>
procedure Read (Dev : in Serial_Device; C : out Character) is abstract;
procedure Write(Dev : in Serial_Device; C : in Character) is abstract;

```

36/2

The `Serial_Device` interface has two dispatching operations which are intended to be implemented by task entries (see 9.1).

4.10 3.10 Access Types

1

A value of an access type (an <access value>) provides indirect access to the object or subprogram it <designates>. Depending on its type, an access value can designate either subprograms, objects created by allocators (see Section 5.8 [4.8], page 230), or more generally <aliased> objects of an appropriate type.

Syntax

2/2

```

access_type_definition ::=
    [null_exclusion] access_to_object_definition
    | [null_exclusion] access_to_subprogram_definition

```

3

```

access_to_object_definition ::=
    access [general_access_modifier] subtype_indication

```

4

```

general_access_modifier ::= all | constant

```

5

```

access_to_subprogram_definition ::=
    access [protected] procedure parameter_profile
    | access [protected] function parameter_and_result_profile

```

5.1/2

```

null_exclusion ::= not null

```

6/2

```

access_definition ::=
    [null_exclusion] access [constant] subtype_mark
    | [null_exclusion] access [protected] procedure parameter_profile
    | [null_exclusion] access [protected] function parameter_and_result_profile

```

Static Semantics

7/1

There are two kinds of access types, `<access-to-object>` types, whose values designate objects, and `<access-to-subprogram>` types, whose values designate subprograms. Associated with an `access-to-object` type is a `<storage pool>`; several access types may share the same storage pool. All descendants of an access type share the same storage pool. A storage pool is an area of storage used to hold dynamically allocated objects (called `<pool elements>`) created by allocators; storage pools are described further in Section 14.11 [13.11], page 526, "Section 14.11 [13.11], page 526, Storage Management".

8

`Access-to-object` types are further subdivided into `<pool-specific>` access types, whose values can designate only the elements of their associated storage pool, and `<general>` access types, whose values can designate the elements of any storage pool, as well as aliased objects created by declarations rather than allocators, and aliased subcomponents of other objects.

9/2

A view of an object is defined to be `<aliased>` if it is defined by an `object_declaration` (see [S0032], page 61) or `component_definition` (see [S0056], page 114) with the reserved word `aliased`, or by a renaming of an aliased view. In addition, the dereference of an `access-to-object` value denotes an aliased view, as does a view conversion (see Section 5.6 [4.6], page 219) of an aliased view. The current instance of a limited tagged type, a protected type, a task type, or a type that has the reserved word `limited` in its full definition is also defined to be aliased. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. Aliased views are the ones that can be designated by an access value.

10

An `access_to_object_definition` defines an `access-to-object` type and its first subtype; the `subtype_indication` (see [S0027], page 56) defines the `<designated subtype>` of the access type. If a `general_access_modifier` (see [S0081], page 156) appears, then the access type is a general access type. If the modifier is the reserved word `constant`, then the type is an `<access-to-constant type>`; a designated object cannot be updated through a value of such a type. If the modifier is the reserved word `all`, then the type is an `<access-to-variable type>`; a designated object can be both read and updated through a value of such a type. If no `general_access_modifier` (see [S0081], page 156) appears in the `access_to_object_definition` (see [S0080], page 156), the access type is a `pool-specific access-to-variable` type.

11

An `access_to_subprogram_definition` defines an `access-to-subprogram` type and its first subtype; the `parameter_profile` or `parameter_and_result_profile` defines the `<designated profile>` of the access type. There is a `<calling convention>` associated with the designated profile; only subprograms with this calling convention can be designated by values of the access type. By default, the calling convention is `"<protected>"` if the reserved word `protected` appears, and `"Ada"` otherwise. See Chapter 16 [Annex B], page 894, for how to override this default.

12/2

An `access_definition` defines an anonymous general access type or an anonymous `access-to-subprogram` type. For a general access type, the `subtype_mark` denotes its `<designated subtype>`; if the `general_access_modifier` (see [S0081], page 156) `constant`

appears, the type is an access-to-constant type; otherwise it is an access-to-variable type. For an access-to-subprogram type, the `parameter_profile` (see [S0157], page 256) or `parameter_and_result_profile` (see [S0158], page 256) denotes its <designated profile>.

13/2

For each access type, there is a null access value designating no entity at all, which can be obtained by (implicitly) converting the literal null to the access type. The null value of an access type is the default initial value of the type. Non-null values of an access-to-object type are obtained by evaluating an allocator, which returns an access value designating a newly created object (see Section 4.10.2 [3.10.2], page 164), or in the case of a general access-to-object type, evaluating an `attribute_reference` for the `Access` or `Unchecked_Access` attribute of an aliased view of an object. Non-null values of an access-to-subprogram type are obtained by evaluating an `attribute_reference` for the `Access` attribute of a non-intrinsic subprogram..

13.1/2

A `null_exclusion` in a construct specifies that the null value does not belong to the access subtype defined by the construct, that is, the access subtype <excludes null>. In addition, the anonymous access subtype defined by the `access_definition` for a controlling access parameter (see Section 4.9.2 [3.9.2], page 145) excludes null. Finally, for a subtype indication without a `null_exclusion`, the subtype denoted by the `subtype_indication` excludes null if and only if the subtype denoted by the `subtype_mark` in the `subtype_indication` excludes null.

14/1

All subtypes of an access-to-subprogram type are constrained. The first subtype of a type defined by an `access_definition` or an `access_to_object_definition` is unconstrained if the designated subtype is an unconstrained array or discriminated subtype; otherwise it is constrained.

Legality Rules

14.1/2

If a `subtype_indication` (see [S0027], page 56), `discriminant_specification` (see [S0062], page 123), `parameter_specification` (see [S0160], page 256), `parameter_and_result_profile` (see [S0158], page 256), `object_renaming_declaration` (see [S0183], page 317), or `formal_object_declaration` (see [S0261], page 458) has a `null_exclusion` (see [S0083], page 156), the `subtype_mark` (see [S0028], page 56) in that construct shall denote an access subtype that does not exclude null.

Dynamic Semantics

15/2

A `composite_constraint` is <compatible> with an unconstrained access subtype if it is compatible with the designated subtype. A `null_exclusion` is compatible with any access subtype that does not exclude null. An access value <satisfies> a `composite_constraint` of an access subtype if it equals the null value of its type or if it designates an object whose value satisfies the constraint. An access value satisfies an exclusion of the null value if it does not equal the null value of its type.

16

The elaboration of an `access_type_definition` creates the access type and its first subtype. For an access-to-object type, this elaboration includes the elaboration of the `subtype_indication`, which creates the designated subtype.

17/2

The elaboration of an `access_definition` creates an anonymous access type.

NOTES

18

80 Access values are called "pointers" or "references" in some other languages.

19

81 Each `access-to-object` type has an associated storage pool; several access types can share the same pool. An object can be created in the storage pool of an access type by an allocator (see Section 5.8 [4.8], page 230) for the access type. A storage pool (roughly) corresponds to what some other languages call a "heap." See Section 14.11 [13.11], page 526, for a discussion of pools.

20

82 Only `index_constraints` and `discriminant_constraints` can be applied to access types (see Section 4.6.1 [3.6.1], page 117, and Section 4.7.1 [3.7.1], page 127).

Examples

21

<Examples of `access-to-object` types:>

22/2

```
type Peripheral_Ref is not null access Peripheral; --< see Section 4.8.1 [3.8.1], page 134>
type Binop_Ptr is access all Binary_Operation'Class;
--< general access-to-class-wide, see Section 4.8.1 [3.9.1], page 143>
```

23

<Example of an access subtype:>

24

```
subtype Drum_Ref is Peripheral_Ref(Drum); --< see Section 4.8.1 [3.8.1], page 134>
```

25

<Example of an `access-to-subprogram` type:>

26

```
type Message_Procedure is access procedure (M : in String := "Error!");
procedure Default_Message_Procedure(M : in String);
Give_Message : Message_Procedure := Default_Message_Procedure'Access;
```

```

...
procedure Other_Procedure(M : in String);
...
Give_Message := Other_Procedure'Access;
...
Give_Message("File not found."); --< call with parameter (.all is optional)>
Give_Message.all;                --< call with no parameters>

```

4.10.1 3.10.1 Incomplete Type Declarations

1

There are no particular limitations on the designated type of an access type. In particular, the type of a component of the designated type can be another access type, or even the same access type. This permits mutually dependent and recursive access types. An `incomplete_type_declaration` can be used to introduce a type to be used as a designated type, while deferring its full definition to a subsequent `full_type_declaration`.

Syntax

2/2

```
incomplete_type_declaration ::= type_defining_identifier [discriminant_part] [is tagged];
```

Static Semantics

2.1/2

An `incomplete_type_declaration` declares an `<incomplete view>` of a type and its first subtype; the first subtype is unconstrained if a `discriminant_part` appears. If the `incomplete_type_declaration` (see [S0085], page 160) includes the reserved word `tagged`, it declares a `<tagged incomplete view>`. An incomplete view of a type is a limited view of the type (see Section 8.5 [7.5], page 292).

2.2/2

Given an access type `<A>` whose designated type `<T>` is an incomplete view, a dereference of a value of type `<A>` also has this incomplete view except when:

2.3/2

- it occurs within the immediate scope of the completion of `<T>`, or

2.4/2

- it occurs within the scope of a `nonlimited_with_clause` that mentions a library package in whose visible part the completion of `<T>` is declared.

2.5/2

In these cases, the dereference has the full view of `<T>`.

2.6/2

Similarly, if a `subtype_mark` denotes a `subtype_declaration` defining a subtype of an incomplete view `<T>`, the `subtype_mark` denotes an incomplete view except under the same two circumstances given above, in which case it denotes the full view of `<T>`.

Legality Rules

3

An `incomplete_type_declaration` requires a completion, which shall be a `full_type_declaration` (see [S0024], page 53). If the `incomplete_type_declaration` (see [S0085], page 160) occurs immediately within either the visible part of a `package_specification` (see [S0174], page 279) or a `declarative_part` (see [S0086], page 175), then the `full_type_declaration` (see [S0024], page 53) shall occur later and immediately within this visible part or `declarative_part` (see [S0086], page 175). If the `incomplete_type_declaration` (see [S0085], page 160) occurs immediately within the private part of a given `package_specification` (see [S0174], page 279), then the `full_type_declaration` (see [S0024], page 53) shall occur later and immediately within either the private part itself, or the `declarative_part` (see [S0086], page 175) of the corresponding `package_body` (see [S0175], page 281).

4/2

If an `incomplete_type_declaration` (see [S0085], page 160) includes the reserved word `tagged`, then a `full_type_declaration` (see [S0024], page 53) that completes it shall declare a `tagged_type`. If an `incomplete_type_declaration` (see [S0085], page 160) has a `known_discriminant_part` (see [S0061], page 123), then a `full_type_declaration` (see [S0024], page 53) that completes it shall have a fully conforming (explicit) `known_discriminant_part` (see [S0061], page 123) (see Section 7.3.1 [6.3.1], page 263). If an `incomplete_type_declaration` (see [S0085], page 160) has no `discriminant_part` (or an `unknown_discriminant_part` (see [S0060], page 123)), then a corresponding `full_type_declaration` (see [S0024], page 53) is nevertheless allowed to have discriminants, either explicitly, or inherited via derivation.

5/2

A name that denotes an incomplete view of a type may be used as follows:

6

- as the `subtype_mark` in the `subtype_indication` of an `access_to_object_definition` (see [S0080], page 156); the only form of constraint allowed in this `subtype_indication` is a `discriminant_constraint`;

7/2

- as the `subtype_mark` in the `subtype_indication` of a `subtype_declaration`; the `subtype_indication` (see [S0027], page 56) shall not have a `null_exclusion` (see [S0083], page 156) or a constraint;

8/2

- as the `subtype_mark` in an `access_definition`.

8.1/2

If such a name denotes a tagged incomplete view, it may also be used:

8.2/2

- as the `subtype_mark` defining the subtype of a parameter in a `formal_part`;

9/2

- as the prefix of an `attribute_reference` whose `attribute_designator` (see [S0101], page 187) is `Class`; such an `attribute_reference` (see [S0100], page 187) is restricted to the uses allowed here; it denotes a tagged incomplete view.

9.1/2

If such a name occurs within the declaration list containing the completion of the incomplete view, it may also be used:

9.2/2

- as the `subtype_mark` defining the subtype of a parameter or result of an `access_to_-subprogram_definition` (see [S0082], page 156).

9.3/2

If any of the above uses occurs as part of the declaration of a primitive subprogram of the incomplete view, and the declaration occurs immediately within the private part of a package, then the completion of the incomplete view shall also occur immediately within the private part; it shall not be deferred to the package body.

9.4/2

No other uses of a name that denotes an incomplete view of a type are allowed.

10/2

A prefix that denotes an object shall not be of an incomplete view.

Static Semantics

11/2

<This paragraph was deleted.>

Dynamic Semantics

12

The elaboration of an `incomplete_type_declaration` has no effect.

NOTES

13

83 Within a `declarative_part`, an `incomplete_type_declaration` and a corresponding `full_type_declaration` cannot be separated by an intervening body. This is because a type has to be completely defined before it is frozen, and a body freezes all types declared prior to it in the same `declarative_part` (see Section 14.14 [13.14], page 550).

Examples

14

<Example of a recursive type:>

15

```
type Cell; --< incomplete type declaration>
type Link is access Cell;
```

16

```
type Cell is
```

```

record
    Value : Integer;
    Succ  : Link;
    Pred  : Link;
end record;

```

17

```

Head  : Link := new Cell'(0, null, null);
Next  : Link := Head.Succ;

```

18

<Examples of mutually dependent access types:>

19/2

```

type Person(<>);    --< incomplete type declaration>
type Car is tagged; --< incomplete type declaration>

```

20/2

```

type Person_Name is access Person;
type Car_Name     is access all Car'Class;

```

21/2

```

type Car is tagged
record
    Number : Integer;
    Owner  : Person_Name;
end record;

```

22

```

type Person(Sex : Gender) is
record
    Name      : String(1 .. 20);
    Birth     : Date;
    Age       : Integer range 0 .. 130;
    Vehicle   : Car_Name;
    case Sex is
        when M => Wife           : Person_Name(Sex => F);
        when F => Husband        : Person_Name(Sex => M);
    end case;
end record;

```

23

```

My_Car, Your_Car, Next_Car : Car_Name := new Car;  --< see Section 5.8
[4.8], page 230>
George : Person_Name := new Person(M);

```

...
George.Vehicle := Your_Car;

4.10.2 3.10.2 Operations of Access Types

1

The attribute `Access` is used to create access values designating aliased objects and non-intrinsic subprograms. The "accessibility" rules prevent dangling references (in the absence of uses of certain unchecked features -- see Section 13).

Name Resolution Rules

2/2

For an `attribute_reference` with `attribute_designator` `Access` (or `Unchecked_Access` -- see Section 14.10 [13.10], page 525), the expected type shall be a single access type `<A>` such that:

2.1/2

- `<A>` is an access-to-object type with designated type `<D>` and the type of the prefix is `<D>'Class` or is covered by `<D>`, or

2.2/2

- `<A>` is an access-to-subprogram type whose designated profile is type conformant with that of the prefix.

2.3/2

The prefix of such an `attribute_reference` is never interpreted as an `implicit_dereference` or a `parameterless_function_call` (see Section 5.1.4 [4.1.4], page 187). The designated type or profile of the expected type of the `attribute_reference` is the expected type or profile for the prefix.

Static Semantics

3/2

The accessibility rules, which prevent dangling references, are written in terms of `<accessibility levels>`, which reflect the run-time nesting of `<masters>`. As explained in Section 8.6.1 [7.6.1], page 299, a master is the execution of a certain construct, such as a `subprogram_body`. An accessibility level is `<deeper than>` another if it is more deeply nested at run time. For example, an object declared local to a called subprogram has a deeper accessibility level than an object declared local to the calling subprogram. The accessibility rules for access types require that the accessibility level of an object designated by an access value be no deeper than that of the access type. This ensures that the object will live at least as long as the access type, which in turn ensures that the access value cannot later designate an object that no longer exists. The `Unchecked_Access` attribute may be used to circumvent the accessibility rules.

4

A given accessibility level is said to be `<statically deeper>` than another if the given level is known at compile time (as defined below) to be deeper than the other for all possible executions. In most cases, accessibility is enforced at compile time by Legality Rules. Run-time accessibility checks are also used, since the Legality Rules do not cover certain cases involving access parameters and generic packages.

5

Each master, and each entity and view created by it, has an accessibility level:

6

- The accessibility level of a given master is deeper than that of each dynamically enclosing master, and deeper than that of each master upon which the task executing the given master directly depends (see Section 10.3 [9.3], page 335).

7/2

- An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. A parameter of a master has the same accessibility level as the master.

8

- The accessibility level of a view of an object or subprogram defined by a renaming_declaration is the same as that of the renamed view.

9/2

- The accessibility level of a view conversion, qualified_expression, or parenthesized expression, is the same as that of the operand.

10/2

- The accessibility level of an aggregate or the result of a function call (or equivalent use of an operator) that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an aggregate or the result of a function call is that of the innermost master that evaluates the aggregate or function call.

10.1/2

- Within a return statement, the accessibility level of the return object is that of the execution of the return statement. If the return statement completes normally by returning from the function, then prior to leaving the function, the accessibility level of the return object changes to be a level determined by the point of call, as does the level of any coextensions (see below) of the return object.

11

- The accessibility level of a derived access type is the same as that of its ultimate ancestor.

11.1/2

- The accessibility level of the anonymous access type defined by an access_definition of an object_renaming_declaration is the same as that of the renamed view.

12/2

- The accessibility level of the anonymous access type of an access discriminant in the `subtype_indication` or `qualified_expression` of an allocator, or in the `expression` or `return_subtype_indication` (see [S0171], page 272) of a return statement is determined as follows:

12.1/2

- If the value of the access discriminant is determined by a `discriminant_association` in a `subtype_indication`, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null);

12.2/2

- If the value of the access discriminant is determined by a `record_component_association` in an aggregate, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null);

12.3/2

- In other cases, where the value of the access discriminant is determined by an object with an unconstrained nominal subtype, the accessibility level of the object.

12.4/2

- The accessibility level of the anonymous access type of an access discriminant in any other context is that of the enclosing object.

13/2

- The accessibility level of the anonymous access type of an access parameter specifying an `access-to-object` type is the same as that of the view designated by the `actual`.

13.1/2

- The accessibility level of the anonymous access type of an access parameter specifying an `access-to-subprogram` type is deeper than that of any master; all such anonymous access types have this same level.

14/2

- The accessibility level of an object created by an allocator is the same as that of the access type, except for an allocator of an anonymous access type that defines the value of an access parameter or an access discriminant. For an allocator defining the value of an access parameter, the accessibility level is that of the innermost master of the call. For one defining an access discriminant, the accessibility level is determined as follows:

14.1/2

- for an allocator used to define the constraint in a `subtype_declaration`, the level of the `subtype_declaration`;

14.2/2

- for an allocator used to define the constraint in a `component_definition`, the level of the enclosing type;

14.3/2

- for an allocator used to define the discriminant of an object, the level of the object.

14.4/2

In this last case, the allocated object is said to be a `<coextension>` of the object whose discriminant designates it, as well as of any object of which the discriminated object is itself a coextension or subcomponent. All coextensions of an object are finalized when the object is finalized (see Section 8.6.1 [7.6.1], page 299).

15

- The accessibility level of a view of an object or subprogram denoted by a dereference of an access value is the same as that of the access type.

16

- The accessibility level of a component, protected subprogram, or entry of (a view of) a composite object is the same as that of (the view of) the composite object.

16.1/2

In the above rules, the operand of a view conversion, parenthesized expression or `qualified_expression` is considered to be used in a context if the view conversion, parenthesized expression or `qualified_expression` itself is used in that context.

17

One accessibility level is defined to be <statically deeper> than another in the following cases:

18

- For a master that is statically nested within another master, the accessibility level of the inner master is statically deeper than that of the outer master.

18.1/2

- The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is statically deeper than that of any master; all such anonymous access types have this same level.

19/2

- The statically deeper relationship does not apply to the accessibility level of the anonymous type of an access parameter specifying an access-to-object type; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other.

20

- For determining whether one level is statically deeper than another when within a generic package body, the generic package is presumed to be instantiated at the same level as where it was declared; run-time checks are needed in the case of more deeply nested instantiations.

21

- For determining whether one level is statically deeper than another when within the declarative region of a type_declaration, the current instance of the type is presumed to be an object created at a deeper level than that of the type.

22

The accessibility level of all library units is called the <library level>; a library-level declaration or entity is one whose accessibility level is the library level.

23

The following attribute is defined for a prefix X that denotes an aliased view of an object:

24/1

X'Access

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined

by the expected type. The expected type shall be a general access type. X shall denote an aliased view of an object, including possibly the current instance (see Section 9.6 [8.6], page 324) of a limited type within its definition, or a formal parameter or generic formal object of a tagged type. The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X'Access is the general access type <A> with designated type <D>:

25

- If <A> is an access-to-variable type, then the view shall be a variable; on the other hand, if <A> is an access-to-constant type, the view may be either a constant or a variable. ■

26/2

- The view shall not be a subcomponent that depends on

discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is constrained by its initial value.

27/2

- If $\langle A \rangle$ is a named access type and $\langle D \rangle$ is a tagged type, then the type of the view shall be covered by $\langle D \rangle$; if $\langle A \rangle$ is anonymous and $\langle D \rangle$ is tagged, then the type of the view shall be either $\langle D \rangle$ 'Class or a type covered by $\langle D \rangle$; if $\langle D \rangle$ is untagged, then the type of the view shall be $\langle D \rangle$, and either:

27.1/2

- the designated subtype of $\langle A \rangle$ shall statically match

27.2/2

the
nom-
i-
nal
sub-
type
of
the
view;
or

- <D>
shall
be
dis-
crim-
i-
nated
in
its
full
view
and
un-
con-
strained
in
any
par-
tial
view,
and
the
des-
ig-
nated
sub-
type
of
<A>
shall
be
un-
con-
strained.

- The accessibility level of the view shall not be statically deeper than that of the access type $\langle A \rangle$. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit.

A check is made that the accessibility level of X is not deeper than that of the access type $\langle A \rangle$. If this check fails, `Program_Error` is raised.

If the nominal subtype of X does not statically match the designated subtype of $\langle A \rangle$, a view conversion of X to the designated subtype is evaluated (which might raise `Constraint_Error` — see Section 5.6 [4.6], page 219) and the value of X 'Access designates that view.

31

The following attribute is defined for a prefix P that denotes a subprogram:

32/2

P'Access

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (<S>), as determined by the expected type. The accessibility level of P shall not be statically deeper than that of <S>. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit. The profile of P shall be subtype-conformant with the designated profile of <S>, and shall not be Intrinsic. If the subprogram denoted by P is declared within a generic unit, and the expression P'Access occurs within the body of that generic unit or within the body of a generic unit declared within the declarative region of the generic unit, then the ultimate ancestor of <S> shall be either

a non-formal type declared within the generic unit or an anonymous access type of an access parameter.

NOTES

33

84 The `Unchecked_Access` attribute yields the same result as the `Access` attribute for objects, but has fewer restrictions (see Section 14.10 [13.10], page 525). There are other predefined operations that yield access values: an allocator can be used to create an object, and return an access value that designates it (see Section 5.8 [4.8], page 230); evaluating the literal `null` yields a null access value that designates no entity at all (see Section 5.2 [4.2], page 189).

34/2

85 The predefined operations of an access type also include the assignment operation, qualification, and membership tests. Explicit conversion is allowed between general access types with matching designated subtypes; explicit conversion is allowed between access-to-subprogram types with subtype conformant profiles (see Section 5.6 [4.6], page 219). Named access types have predefined equality operators; anonymous access types do not, but they can use the predefined equality operators for `<universal_access>` (see Section 5.5.2 [4.5.2], page 206).

35

86 The object or subprogram designated by an access value can be named with a dereference, either an `explicit_dereference` (see [S0094], page 179) or an `implicit_dereference`. See Section 5.1 [4.1], page 179.

36

87 A call through the dereference of an access-to-subprogram value is never a dispatching call.

37/2

88 The `Access` attribute for subprograms and parameters of an anonymous access-to-subprogram type may together be used to implement "downward closures" -- that is, to pass a more-nested subprogram as a parameter to a less-nested subprogram, as might be appropriate for an iterator abstraction or numerical

integration. Downward closures can also be implemented using generic formal subprograms (see Section 13.6 [12.6], page 470). Note that `Unchecked_Access` is not allowed for subprograms.

38

89 Note that using an `access-to-class-wide` tagged type with a dispatching operation is a potentially more structured alternative to using an `access-to-subprogram` type.

39

90 An implementation may consider two `access-to-subprogram` values to be unequal, even though they designate the same subprogram. This might be because one points directly to the subprogram, while the other points to a special prologue that performs an `Elaboration_Check` and then jumps to the subprogram. See Section 5.5.2 [4.5.2], page 206.

Examples

40

<Example of use of the `Access` attribute:>

41

```
Martha : Person_Name := new Person(F);           --< see Section 4.10.1
[3.10.1], page 160>
Cars   : array (1..2) of aliased Car;
...
Martha.Vehicle := Cars(1)'Access;
George.Vehicle := Cars(2)'Access;
```

4.11 3.11 Declarative Parts

1

A `declarative_part` contains `declarative_items` (possibly none).

Syntax

2

```
declarative_part ::= {declarative_item}
```

3

```
declarative_item ::=
  basic_declarative_item | body
```

4/1

```
basic_declarative_item ::=
  basic_declaration | aspect_clause | use_clause
```

5

body ::= proper_body | body_stub

6

proper_body ::=
subprogram_body | package_body | task_body | protected_body ■
Static Semantics

6.1/2

The list of declarative_items of a declarative_part is called the <declaration list> of the declarative_part.

Dynamic Semantics

7

The elaboration of a declarative_part consists of the elaboration of the declarative_items, if any, in the order in which they are given in the declarative_part.

8

An elaborable construct is in the <elaborated> state after the normal completion of its elaboration. Prior to that, it is <not yet elaborated>.

9

For a construct that attempts to use a body, a check (Elaboration_Check) is performed, as follows:

10/1

- For a call to a (non-protected) subprogram that has an explicit body, a check is made that the body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

11

- For a call to a protected operation of a protected type (that has a body -- no check is performed if a pragma Import applies to the protected type), a check is made that the protected_body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order.

12

- For the activation of a task, a check is made by the activator that the task_body is already elaborated. If two or more tasks are being activated together (see Section 10.2 [9.2], page 333), as the result of the elaboration of a declarative_part or the initialization for the object created by an allocator, this check is done for all of them before activating any of them.

13

- For the instantiation of a generic unit that has a body, a check is made that this body is already elaborated. This check and the evaluation of any explicit_generic_actual_parameters of the instantiation are done in an arbitrary order.

14

The exception `Program.Error` is raised if any of these checks fails.

4.11.1 3.11.1 Completions of Declarations

1/1

Declarations sometimes come in two parts. A declaration that requires a second part is said to `<require completion>`. The second part is called the `<completion>` of the declaration (and of the entity declared), and is either another declaration, a body, or a pragma. A `<body>` is a body, an `entry_body`, or a `renaming-as-body` (see Section 9.5.4 [8.5.4], page 319).

Name Resolution Rules

2

A construct that can be a completion is interpreted as the completion of a prior declaration only if:

3

- The declaration and the completion occur immediately within the same declarative region;

4

- The defining name or `defining_program_unit_name` in the completion is the same as in the declaration, or in the case of a pragma, the pragma applies to the declaration;

5

- If the declaration is overloadable, then the completion either has a `type-conformant` profile, or is a pragma.

Legality Rules

6

An implicit declaration shall not have a completion. For any explicit declaration that is specified to `<require completion>`, there shall be a corresponding explicit completion.

7

At most one completion is allowed for a given declaration. Additional requirements on completions appear where each kind of completion is defined.

8

A type is `<completely defined>` at a place that is after its full type definition (if it has one) and after all of its subcomponent types are completely defined. A type shall be completely defined before it is frozen (see Section 14.14 [13.14], page 550, and Section 8.3 [7.3], page 283).

NOTES

9

91 Completions are in principle allowed for any kind of explicit declaration. However, for some kinds of declaration, the only allowed

completion is a pragma Import, and implementations are not required to support pragma Import for every kind of entity.

10

92 There are rules that prevent premature uses of declarations that have a corresponding completion. The `Elaboration_Checks` of Section 4.11 [3.11], page 175, prevent such uses at run time for subprograms, protected operations, tasks, and generic units. The rules of Section 14.14 [13.14], page 550, "Section 14.14 [13.14], page 550, Freezing Rules" prevent, at compile time, premature uses of other entities such as private types and deferred constants.

5 4 Names and Expressions

1

The rules applicable to the different forms of name and expression, and to their evaluation, are given in this section.

5.1 4.1 Names

1

Names can denote declared entities, whether declared explicitly or implicitly (see Section 4.1 [3.1], page 49). Names can also denote objects or subprograms designated by access values; the results of type_conversions or function_calls; subcomponents and slices of objects and values; protected subprograms, single entries, entry families, and entries in families of entries. Finally, names can denote attributes of any of the foregoing.

Syntax

2

```
name ::=
    direct_name | explicit_dereference
    | indexed_component | slice
    | selected_component | attribute_reference
    | type_conversion | function_call
    | character_literal
```

3

```
direct_name ::= identifier | operator_symbol
```

4

```
prefix ::= name | implicit_dereference
```

5

```
explicit_dereference ::= name.all
```

6

```
implicit_dereference ::= name
```

7/2

Certain forms of name (indexed_components, selected_components, slices, and attribute_references) include a prefix that is either itself a name that denotes some related entity, or an implicit_dereference of an access value that designates some related entity.

Name Resolution Rules

8

The name in a <dereference> (either an implicit_dereference or an explicit_dereference) is expected to be of any access type.

Static Semantics

9

If the type of the name in a dereference is some access-to-object type <T>, then the dereference denotes a view of an object, the <nominal subtype> of the view being the designated subtype of <T>.

10

If the type of the name in a dereference is some access-to-subprogram type <S>, then the dereference denotes a view of a subprogram, the <profile> of the view being the designated profile of <S>.

Dynamic Semantics

11/2

The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is a `direct_name` or a `character_literal`.

12

The evaluation of a name that has a prefix includes the evaluation of the prefix. The evaluation of a prefix consists of the evaluation of the name or the `implicit_dereference`. The prefix denotes the entity denoted by the name or the `implicit_dereference`.

13

The evaluation of a dereference consists of the evaluation of the name and the determination of the object or subprogram that is designated by the value of the name. A check is made that the value of the name is not the null access value. `Constraint_Error` is raised if this check fails. The dereference denotes the object or subprogram designated by the value of the name.

Examples

14

<Examples of direct names:>

15

```
Pi      <-- the direct name of a number>      (see Section 4.3.2 [3.3.2],
page 65)
Limit   <-- the direct name of a constant>    (see Section 4.3.1
[3.3.1], page 61)
Count   <-- the direct name of a scalar variable> (see Section 4.3.1
[3.3.1], page 61)
Board   <-- the direct name of an array variable> (see Section 4.6.1
[3.6.1], page 117)
Matrix  <-- the direct name of a type>        (see Section 4.6 [3.6],
page 114)
Random  <-- the direct name of a function>    (see Section 7.1 [6.1],
page 255)
Error   <-- the direct name of an exception>  (see Section 12.1
[11.1], page 419)
```

16

<Examples of dereferences:>

17

```

Next_Car.all    --< explicit dereference denoting the object designated by>
                --< the access variable Next_Car (see Section 4.10.1
[3.10.1], page 160)>
Next_Car.Owner --< selected component with implicit dereference;>
                --< same as Next_Car.all.Owner>

```

5.1.1 4.1.1 Indexed Components

1

An indexed_component denotes either a component of an array or an entry in a family of entries.

Syntax

2

```

indexed_component ::= prefix(expression {, expression})
                    Name Resolution Rules

```

3

The prefix of an indexed_component with a given number of expressions shall resolve to denote an array (after any implicit dereference) with the corresponding number of index positions, or shall resolve to denote an entry family of a task or protected object (in which case there shall be only one expression).

4

The expected type for each expression is the corresponding index type.

Static Semantics

5

When the prefix denotes an array, the indexed_component denotes the component of the array with the specified index value(s). The nominal subtype of the indexed_component is the component subtype of the array type.

6

When the prefix denotes an entry family, the indexed_component denotes the individual entry of the entry family with the specified index value.

Dynamic Semantics

7

For the evaluation of an indexed_component, the prefix and the expressions are evaluated in an arbitrary order. The value of each expression is converted to the corresponding index type. A check is made that each index value belongs to the corresponding index range of the array or entry family denoted by the prefix. Constraint_Error is raised if this check fails.

Examples

8

<Examples of indexed components:>

9

```

My_Schedule(Sat) --< a component of a one-dimensional array (see Section
[3.6.1], page 117)>

```

Page(10)	--<	a component of a one-dimensional array	(see Section
[3.6], page 114)>			
Board(M, J + 1)	--<	a component of a two-dimensional array	(see Section
[3.6.1], page 117)>			
Page(10)(20)	--<	a component of a component	(see Section 4.6
[3.6], page 114)>			
Request(Medium)	--<	an entry in a family of entries	(see Section 10.1
[9.1], page 329)>			
Next_Frame(L)(M, N)	--<	a component of a function call	(see Section 7.1
[6.1], page 255)>			

NOTES

10

1 <Notes on the examples:> Distinct notations are used for components of multidimensional arrays (such as Board) and arrays of arrays (such as Page). The components of an array of arrays are arrays and can therefore be indexed. Thus Page(10)(20) denotes the 20th component of Page(10). In the last example Next_Frame(L) is a function call returning an access value that designates a two-dimensional array.

5.1.2 4.1.2 Slices

1

A slice denotes a one-dimensional array formed by a sequence of consecutive components of a one-dimensional array. A slice of a variable is a variable; a slice of a constant is a constant; a slice of a value is a value.

Syntax

2

slice ::= prefix(discrete_range)

Name Resolution Rules

3

The prefix of a slice shall resolve to denote a one-dimensional array (after any implicit dereference).

4

The expected type for the discrete_range of a slice is the index type of the array type.

Static Semantics

5

A slice denotes a one-dimensional array formed by the sequence of consecutive components of the array denoted by the prefix, corresponding to the range of values of the index given by the discrete_range.

6

The type of the slice is that of the prefix. Its bounds are those defined by the discrete_range.

Dynamic Semantics

7

For the evaluation of a slice, the prefix and the discrete_range are evaluated in an arbitrary order. If the slice is not a <null slice> (a slice where the discrete_range is a null range), then a check is made that the bounds of the discrete_range belong to the index range of the array denoted by the prefix. Constraint_Error is raised if this check fails.

NOTES

8

2 A slice is not permitted as the prefix of an Access attribute_reference, even if the components or the array as a whole are aliased. See Section 4.10.2 [3.10.2], page 164.

9

3 For a one-dimensional array A, the slice A(N .. N) denotes an array that has only one component; its type is the type of A. On the other hand, A(N) denotes a component of the array A and has the corresponding component type.

Examples

10

<Examples of slices:>

11

```

    Stars(1 .. 15)          --< a slice of 15 characters      (see Section 4.6.3 [3.6.3], page 122)>
    Page(10 .. 10 + Size) --< a slice of 1 + Size components (see Section 4.6 [3.6], page 114)>
    Page(L)(A .. B)       --< a slice of the array Page(L)   (see Section 4.6 [3.6], page 114)>
    Stars(1 .. 0)         --< a null slice                    (see Section 4.6.3 [3.6.3], page 122)>
    My_Schedule(Weekday) --< bounds given by subtype       (see Section 4.6.1 [3.6.1], page 117 and Section 4.5.1 [3.5.1], page 92)>
    Stars(5 .. 15)(K)     --< same as Stars(K)              (see Section 4.6.3 [3.6.3], page 122)>
                          --< provided that K is in 5 .. 15>

```

5.1.3 4.1.3 Selected Components

1

Selected_components are used to denote components (including discriminants), entries, entry families, and protected subprograms; they are also used as expanded names as described below.

Syntax

2

selected_component ::= prefix . selector_name

3

selector_name ::= identifier | character_literal | operator_symbol

Name Resolution Rules

4

A selected_component is called an <expanded name> if, according to the visibility rules, at least one possible interpretation of its prefix denotes a package or an enclosing named construct (directly, not through a subprogram_renaming_declaration or generic_renaming_declaration).

5

A selected_component that is not an expanded name shall resolve to denote one of the following:

6

- A component (including a discriminant):

7

The prefix shall resolve to denote an object or value of some non-array composite type (after any implicit dereference). The selector_name shall resolve to denote a discriminant_specification of the type, or, unless the type is a protected type, a component_declaration of the type. The selected_component denotes the corresponding component of the object or value.

8

- A single entry, an entry family, or a protected subprogram:

9

The prefix shall resolve to denote an object or value of some task or protected type (after any implicit dereference). The selector_name shall resolve to denote an entry_declaration or subprogram_declaration occurring (implicitly or explicitly) within the visible part of that type. The selected_component denotes the corresponding entry, entry family, or protected subprogram.

9.1/2

- A view of a subprogram whose first formal parameter is of a tagged type or is an access parameter whose designated type is tagged:

9.2/2

The prefix (after any implicit dereference) shall resolve to denote an object or value of a specific tagged type <T> or class-wide type <T>'Class. The selector_name shall resolve to denote a view of a

subprogram declared immediately within the declarative region in which an ancestor of the type `<T>` is declared. The first formal parameter of the subprogram shall be of type `<T>`, or a class-wide type that covers `<T>`, or an access parameter designating one of these types. The designator of the subprogram shall not be the same as that of a component of the tagged type visible at the point of the `selected_component`. The `selected_component` denotes a view of this subprogram that omits the first formal parameter. This view is called a `<prefixed view>` of the subprogram, and the prefix of the `selected_component` (after any implicit dereference) is called the `<prefix>` of the prefixed view.

10

An expanded name shall resolve to denote a declaration that occurs immediately within a named declarative region, as follows:

11

- The prefix shall resolve to denote either a package (including the current instance of a generic package, or a rename of a package), or an enclosing named construct.

12

- The `selector_name` shall resolve to denote a declaration that occurs immediately within the declarative region of the package or enclosing construct (the declaration shall be visible at the place of the expanded name — see Section 9.3 [8.3], page 308). The expanded name denotes that declaration.

13

- If the prefix does not denote a package, then it shall be a `direct_name` or an expanded name, and it shall resolve to denote a program unit (other than a package), the current instance of a type, a `block_statement`, a `loop_statement`, or an `accept_statement` (see [S0201], page 347) (in the case of an `accept_statement` (see [S0201], page 347) or `entry-body` (see [S0203], page 348), no family index is allowed); the expanded name shall occur within the declarative region of this construct. Further, if this construct is a callable construct and the prefix denotes more than one such enclosing callable construct, then the expanded name is ambiguous, independently of the `selector_name`.

Legality Rules

13.1/2

For a subprogram whose first parameter is an access parameter, the prefix of any prefixed view shall denote an aliased view of an object.

13.2/2

For a subprogram whose first parameter is of mode `in out` or `out`, or of an anonymous access-to-variable type, the prefix of any prefixed view shall denote a variable.

Dynamic Semantics

14

The evaluation of a `selected_component` includes the evaluation of the prefix.

15

For a selected_component that denotes a component of a variant, a check is made that the values of the discriminants are such that the value or object denoted by the prefix has this component. The exception Constraint_Error is raised if this check fails.

Examples

16

<Examples of selected components:>

17/2

```
Tomorrow.Month    --< a record component      (see Section 4.8 [3.8],
page 130)>
Next_Car.Owner    --< a record component      (see Section 4.10.1
[3.10.1], page 160)>
Next_Car.Owner.Age --< a record component      (see Section 4.10.1
[3.10.1], page 160)>
--< the previous two lines involve implicit dereferences>
Writer.Unit       --< a record component (a discriminant) (see Section 4.8
[3.8.1], page 134)>
Min_Cell(H).Value --< a record component of the result (see Section 7.1
[6.1], page 255)>
--< of the function call Min_Cell(H)>
Cashier.Append    --< a prefixed view of a procedure (see Section 4.9.4
[3.9.4], page 152)>
Control.Seize     --< an entry of a protected object (see Section 10.4
[9.4], page 337)>
Pool(K).Write     --< an entry of the task Pool(K) (see Section 10.4
[9.4], page 337)>
```

18

<Examples of expanded names:>

19

```
Key_Manager."<"    --< an operator of the visible part of a package (see
[7.3.1], page 287)>
Dot_Product.Sum   --< a variable declared in a function body (see Section
[6.1], page 255)>
Buffer.Pool       --< a variable declared in a protected unit (see Section
[9.11], page 391)>
Buffer.Read       --< an entry of a protected unit (see Section 10.11
[9.11], page 391)>
Swap.Temp         --< a variable declared in a block statement (see Section
[5.6], page 251)>
Standard.Boolean  --< the name of a predefined type (see Section 15.1
[A.1], page 556)>
```

5.1.4 4.1.4 Attributes

1

An <attribute> is a characteristic of an entity that can be queried via an `attribute_reference` (see [S0100], page 187) or a `range_attribute_reference` (see [S0102], page 187).

Syntax

2

`attribute_reference ::= prefix'attribute_designator`

3

`attribute_designator ::=`
 `identifier[(<static_>expression)]`
 | `Access` | `Delta` | `Digits`

4

`range_attribute_reference ::= prefix'range_attribute_designator`

5

`range_attribute_designator ::= Range[(<static_>expression)]`
Name Resolution Rules

6

In an `attribute_reference`, if the `attribute_designator` is for an attribute defined for (at least some) objects of an access type, then the prefix is never interpreted as an `implicit_dereference`; otherwise (and for all `range_attribute_references`), if the type of the name within the prefix is of an access type, the prefix is interpreted as an `implicit_dereference`. Similarly, if the `attribute_designator` is for an attribute defined for (at least some) functions, then the prefix is never interpreted as a `parameterless_function_call`; otherwise (and for all `range_attribute_references`), if the prefix consists of a name that denotes a function, it is interpreted as a `parameterless_function_call`.

7

The expression, if any, in an `attribute_designator` or `range_attribute_designator` is expected to be of any integer type.

Legality Rules

8

The expression, if any, in an `attribute_designator` or `range_attribute_designator` shall be static.

Static Semantics

9

An `attribute_reference` denotes a value, an object, a subprogram, or some other kind of program entity.

10

A `range_attribute_reference` `X'Range(N)` is equivalent to the range `X'First(N) .. X'Last(N)`, except that the prefix is only evaluated once. Similarly, `X'Range` is equivalent to `X'First .. X'Last`, except that the prefix is only evaluated once.

Dynamic Semantics

11

The evaluation of an `attribute_reference` (or `range_attribute_reference`) consists of the evaluation of the prefix.

Implementation Permissions

12/1

An implementation may provide implementation–defined attributes; the identifier for an implementation–defined attribute shall differ from those of the language–defined attributes unless supplied for compatibility with a previous edition of this International Standard.

NOTES

13

4 Attributes are defined throughout this International Standard, and are summarized in Chapter 24 [Annex K], page 1179.

14/2

5 In general, the name in a prefix of an `attribute_reference` (or a `range_attribute_reference`) has to be resolved without using any context. However, in the case of the `Access` attribute, the expected type for the `attribute_reference` has to be a single access type, and the resolution of the name can use the fact that the type of the object or the profile of the callable entity denoted by the prefix has to match the designated type or be type conformant with the designated profile of the access type.

Examples

15

<Examples of attributes:>

16

```
Color'First      --< minimum value of the enumeration type Color    (see Section
[3.5.1], page 92)>
Rainbow'Base'First --< same as Color'First      (see Section 4.5.1 [3.5.1],
page 92)>
Real'Digits      --< precision of the type Real      (see Section 4.5.7
[3.5.7], page 103)>
Board'Last(2)    --< upper bound of the second dimension of Board    (see Section
[3.6.1], page 117)>
Board'Range(1)   --< index range of the first dimension of Board    (see Section
[3.6.1], page 117)>
Pool(K)'Terminated --< True if task Pool(K) is terminated    (see Section 10.1
[9.1], page 329)>
Date'Size        --< number of bits for records of type Date    (see Section 4.
[3.8], page 130)>
Message'Address  --< address of the record variable Message    (see Section 4.7
[3.7.1], page 127)>
```

5.2 4.2 Literals

1

A <literal> represents a value literally, that is, by means of notation suited to its kind. A literal is either a numeric_literal, a character_literal, the literal null, or a string_literal.

Name Resolution Rules

2/2

<This paragraph was deleted.>

3

For a name that consists of a character_literal, either its expected type shall be a single character type, in which case it is interpreted as a parameterless function_call that yields the corresponding value of the character type, or its expected profile shall correspond to a parameterless function with a character result type, in which case it is interpreted as the name of the corresponding parameterless function declared as part of the character type's definition (see Section 4.5.1 [3.5.1], page 92). In either case, the character_literal denotes the enumeration_literal_specification.

4

The expected type for a primary that is a string_literal shall be a single string type.

Legality Rules

5

A character_literal that is a name shall correspond to a defining_character_literal of the expected type, or of the result type of the expected profile.

6

For each character of a string_literal with a given expected string type, there shall be a corresponding defining_character_literal of the component type of the expected string type.

7/2

<This paragraph was deleted.>

Static Semantics

8/2

An integer literal is of type <universal_integer>. A real literal is of type <universal_real>. The literal null is of type <universal_access>.

Dynamic Semantics

9

The evaluation of a numeric literal, or the literal null, yields the represented value.

10

The evaluation of a string_literal that is a primary yields an array value containing the value of each character of the sequence of characters of the string_literal, as defined in Section 3.6 [2.6], page 42. The bounds of this array value are determined according to the rules for positional_array_aggregates (see Section 5.3.3 [4.3.3], page 196), except that for a null string literal, the upper bound is the predecessor of the lower bound.

11

For the evaluation of a string_literal of type <T>, a check is made that the value of each character of the string_literal belongs to the component subtype of <T>. For the evaluation of a null string literal, a check is made that its lower bound is greater than the lower bound

of the base range of the index type. The exception `Constraint_Error` is raised if either of these checks fails.

NOTES

12

6 Enumeration literals that are identifiers rather than `character_literals` follow the normal rules for identifiers when used in a name (see Section 5.1 [4.1], page 179, and Section 5.1.3 [4.1.3], page 183). `Character_literals` used as `selector_names` follow the normal rules for expanded names (see Section 5.1.3 [4.1.3], page 183).

Examples

13

<Examples of literals:>

14

```
3.14159_26536  --< a real literal>
1_345         --< an integer literal>
'A'          --< a character literal>
"Some Text"   --< a string literal >
```

5.3 4.3 Aggregates

1

An <aggregate> combines component values into a composite value of an array type, record type, or record extension.

Syntax

2

```
aggregate ::= record_aggregate | extension_aggregate | array_aggregate
```

Name Resolution Rules

3/2

The expected type for an aggregate shall be a single array type, record type, or record extension.

Legality Rules

4

An aggregate shall not be of a class-wide type.

Dynamic Semantics

5

For the evaluation of an aggregate, an anonymous object is created and values for the components or ancestor part are obtained (as described in the subsequent subclause for each kind of the aggregate) and assigned into the corresponding components or ancestor part of the anonymous object. Obtaining the values and the assignments occur in an arbitrary order. The value of the aggregate is the value of this object.

6

If an aggregate is of a tagged type, a check is made that its value belongs to the first subtype of the type. `Constraint_Error` is raised if this check fails.

5.3.1 4.3.1 Record Aggregates

1

In a `record_aggregate`, a value is specified for each component of the record or record extension value, using either a named or a positional association.

Syntax

2

```
record_aggregate ::= (record_component_association_list)
```

3

```
record_component_association_list ::=  
    record_component_association {, record_component_association}  
    | null record
```

4/2

```
record_component_association ::=  
    [component_choice_list =>] expression  
    | component_choice_list => <>
```

5

```
component_choice_list ::=  
    <component_selector_name { | <component_selector_name }  
    | others
```

6

A `record_component_association` (see [S0107], page 191) is a `<named component association>` if it has a `component_choice_list`; otherwise, it is a `<positional component association>`. Any positional component associations shall precede any named component associations. If there is a named association with a `component_choice_list` of others, it shall come last.

7

In the `record_component_association_list` (see [S0106], page 191) for a `record_aggregate` (see [S0105], page 191), if there is only one association, it shall be a named association.

Name Resolution Rules

8/2

The expected type for a `record_aggregate` shall be a single record type or record extension.

9

For the `record_component_association_list` (see [S0106], page 191) of a `record_aggregate` (see [S0105], page 191), all components of the composite value defined by the aggregate are <needed>; for the association list of an `extension_aggregate`, only those components not determined by the ancestor expression or subtype are needed (see Section 5.3.2 [4.3.2], page 194). Each `selector_name` (see [S0099], page 184) in a `record_component_association` (see [S0107], page 191) shall denote a needed component (including possibly a discriminant).

10

The expected type for the expression of a `record_component_association` (see [S0107], page 191) is the type of the <associated> component(s); the associated component(s) are as follows:

11

- For a positional association, the component (including possibly a discriminant) in the corresponding relative position (in the declarative region of the type), counting only the needed components;

12

- For a named association with one or more <component_>`selector_names`, the named component(s);

13

- For a named association with the reserved word `others`, all needed components that are not associated with some previous association.

Legality Rules

14

If the type of a `record_aggregate` is a `record extension`, then it shall be a descendant of a `record type`, through one or more `record extensions` (and no `private extensions`).

15

If there are no components needed in a given `record_component_association_list` (see [S0106], page 191), then the reserved words `null record` shall appear rather than a list of `record_component_association` (see [S0107], page 191)s.

16/2

Each `record_component_association` other than an `others choice` with a <> shall have at least one associated component, and each needed component shall be associated with exactly one `record_component_association` (see [S0107], page 191). If a `record_component_association` (see [S0107], page 191) with an expression has two or more associated components, all of them shall be of the same type.

17

If the components of a `variant_part` are needed, then the value of a discriminant that governs the `variant_part` shall be given by a static expression.

17.1/2

A `record_component_association` for a discriminant without a `default_expression` shall have an expression rather than <>.

18

The evaluation of a `record_aggregate` consists of the evaluation of the `record_component_association_list` (see [S0106], page 191).

19

For the evaluation of a `record_component_association_list` (see [S0106], page 191), any per-object constraints (see Section 4.8 [3.8], page 130) for components specified in the association list are elaborated and any expressions are evaluated and converted to the subtype of the associated component. Any constraint elaborations and expression evaluations (and conversions) occur in an arbitrary order, except that the expression for a discriminant is evaluated (and converted) prior to the elaboration of any per-object constraint that depends on it, which in turn occurs prior to the evaluation and conversion of the expression for the component with the per-object constraint.

19.1/2

For a `record_component_association` with an expression, the expression defines the value for the associated component(s). For a `record_component_association` with `<>`, if the `component_declaration` has a `default_expression`, that `default_expression` defines the value for the associated component(s); otherwise, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see Section 4.3.1 [3.3.1], page 61).

20

The expression of a `record_component_association` is evaluated (and converted) once for each associated component.

NOTES

21

7 For a `record_aggregate` with positional associations, expressions specifying discriminant values appear first since the `known_discriminant_part` is given first in the declaration of the type; they have to be in the same order as in the `known_discriminant_part`.

Examples

22

<Example of a record aggregate with positional associations:>

23

(4, July, 1776)
[3.8], page 130 >

--< see Section 4.8 ■

24

<Examples of record aggregates with named associations:>

25

(Day => 4, Month => July, Year => 1776)
(Month => July, Day => 4, Year => 1776)

26

```
(Disk, Closed, Track => 5, Cylinder => 12)      --< see Section 4.8.1 [3.8.1], page 134>
(Unit => Disk, Status => Closed, Cylinder => 9, Track => 1)
```

27/2

<Examples of component associations with several choices:>

28

```
(Value => 0, Succ|Pred => new Cell'(0, null, null))  --< see Section 4.10.1 [3.10.1], page 160>
```

29

```
--< The allocator is evaluated twice: Succ and Pred designate different cells>
```

29.1/2

```
(Value => 0, Succ|Pred => <>)  --< see Section 4.10.1 [3.10.1], page 160>
```

29.2/2

```
--< Succ and Pred will be set to null>
```

30

<Examples of record aggregates for tagged types (see Section 4.9 [3.9], page 136, and Section 4.9.1 [3.9.1], page 143):>

31

```
Expression'(null record)
Literal'(Value => 0.0)
Painted_Point'(0.0, Pi/2.0, Paint => Red)
```

5.3.2 4.3.2 Extension Aggregates

1

An `extension_aggregate` specifies a value for a type that is a record extension by specifying a value or subtype for an ancestor of the type, followed by associations for any components not determined by the `ancestor_part`.

Syntax

2

```
extension_aggregate ::=
  (ancestor_part with record_component_association_list)
```

3

```
ancestor_part ::= expression | subtype_mark
```

Name Resolution Rules

4/2

The expected type for an `extension_aggregate` shall be a single type that is a record extension. If the `ancestor_part` is an expression, it is expected to be of any tagged type.

Legality Rules

5/2

If the `ancestor_part` is a `subtype_mark`, it shall denote a specific tagged subtype. If the `ancestor_part` is an expression, it shall not be dynamically tagged. The type of the `extension_aggregate` shall be derived from the type of the `ancestor_part`, through one or more record extensions (and no private extensions).

Static Semantics

6

For the `record_component_association_list` (see [S0106], page 191) of an `extension_aggregate` (see [S0109], page 194), the only components `<needed>` are those of the composite value defined by the aggregate that are not inherited from the type of the `ancestor_part` (see [S0110], page 194), plus any inherited discriminants if the `ancestor_part` (see [S0110], page 194) is a `subtype_mark` (see [S0028], page 56) that denotes an unconstrained subtype.

Dynamic Semantics

7

For the evaluation of an `extension_aggregate`, the `record_component_association_list` (see [S0106], page 191) is evaluated. If the `ancestor_part` is an expression, it is also evaluated; if the `ancestor_part` is a `subtype_mark`, the components of the value of the aggregate not given by the `record_component_association_list` (see [S0106], page 191) are initialized by default as for an object of the ancestor type. Any implicit initializations or evaluations are performed in an arbitrary order, except that the expression for a discriminant is evaluated prior to any other evaluation or initialization that depends on it.

8

If the type of the `ancestor_part` has discriminants that are not inherited by the type of the `extension_aggregate`, then, unless the `ancestor_part` is a `subtype_mark` that denotes an unconstrained subtype, a check is made that each discriminant of the ancestor has the value specified for a corresponding discriminant, either in the `record_component_association_list` (see [S0106], page 191), or in the `derived_type_definition` for some ancestor of the type of the `extension_aggregate`. `Constraint_Error` is raised if this check fails.

NOTES

9

8 If all components of the value of the `extension_aggregate` are determined by the `ancestor_part`, then the `record_component_association_list` (see [S0106], page 191) is required to be simply null record.

10

9 If the `ancestor_part` is a `subtype_mark`, then its type can be abstract. If its type is controlled, then as the last step of evaluating the aggregate, the `Initialize` procedure of the ancestor type is

called, unless the Initialize procedure is abstract (see Section 8.6 [7.6], page 295).

Examples

11

<Examples of extension aggregates (for types defined in Section 4.9.1 [3.9.1], page 143):>

12

```
Painted_Point'(Point with Red)
(Point'(P) with Paint => Black)
```

13

```
(Expression with Left => 1.2, Right => 3.4)
Addition'(Binop with null record)
  --< presuming Binop is of type Binary_Operation>
```

5.3.3 4.3.3 Array Aggregates

1

In an array_aggregate, a value is specified for each component of an array, either positionally or by its index. For a positional_array_aggregate, the components are given in increasing-index order, with a final others, if any, representing any remaining components. For a named_array_aggregate, the components are identified by the values covered by the discrete_choices.

Syntax

2

```
array_aggregate ::=
  positional_array_aggregate | named_array_aggregate
```

3/2

```
positional_array_aggregate ::=
  (expression, expression {, expression})
  | (expression {, expression}, others => expression)
  | (expression {, expression}, others => <>)
```

4

```
named_array_aggregate ::=
  (array_component_association {, array_component_association})
```

5/2

```
array_component_association ::=
  discrete_choice_list => expression
  | discrete_choice_list => <>
```

6

An `<n-dimensional>` `array_aggregate` is one that is written as `n` levels of nested `array_aggregates` (or at the bottom level, equivalent `string_literals`). For the multidimensional case (`n` \geq 2) the `array_aggregates` (or equivalent `string_literals`) at the `n-1` lower levels are called `<subaggregate>`s of the enclosing `n-dimensional array_aggregate`. The expressions of the bottom level subaggregates (or of the `array_aggregate` itself if one-dimensional) are called the `<array component expressions>` of the enclosing `n-dimensional array_aggregate`.

Name Resolution Rules

7/2

The expected type for an `array_aggregate` (that is not a subaggregate) shall be a single array type. The component type of this array type is the expected type for each array component expression of the `array_aggregate`.

8

The expected type for each `discrete_choice` in any `discrete_choice_list` of a `named_array_aggregate` is the type of the `<corresponding index>`; the corresponding index for an `array_aggregate` that is not a subaggregate is the first index of its type; for an `(n-m)-dimensional subaggregate` within an `array_aggregate` of an `n-dimensional` type, the corresponding index is the index in position `m+1`.

Legality Rules

9

An `array_aggregate` of an `n-dimensional array type` shall be written as an `n-dimensional array_aggregate`.

10

An others choice is allowed for an `array_aggregate` only if an `<applicable index constraint>` applies to the `array_aggregate`. An applicable index constraint is a constraint provided by certain contexts where an `array_aggregate` is permitted that can be used to determine the bounds of the array value specified by the aggregate. Each of the following contexts (and none other) defines an applicable index constraint:

11/2

- For an `explicit_actual_parameter`, an `explicit_generic_actual_parameter`, the expression of a return statement, the initialization expression in an `object_declaration` (see [S0032], page 61), or a `default_expression` (see [S0063], page 123) (for a parameter or a component), when the nominal subtype of the corresponding formal parameter, generic formal parameter, function return object, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

12

- For the expression of an `assignment_statement` where the name denotes an array variable, the applicable index constraint is the constraint of the array variable;

13

- For the operand of a `qualified_expression` whose `subtype_mark` denotes a constrained array subtype, the applicable index constraint is the constraint of the subtype;

14

- For a component expression in an aggregate, if the component's nominal subtype is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

15

- For a parenthesized expression, the applicable index constraint is that, if any, defined for the expression.

16

The applicable index constraint <applies> to an array_aggregate that appears in such a context, as well as to any subaggregates thereof. In the case of an explicit_actual_parameter (or default_expression) for a call on a generic formal subprogram, no applicable index constraint is defined.

17

The discrete_choice_list of an array_component_association is allowed to have a discrete_choice that is a nonstatic expression or that is a discrete_range that defines a nonstatic or null range, only if it is the single discrete_choice of its discrete_choice_list, and there is only one array_component_association in the array_aggregate.

18

In a named_array_aggregate with more than one discrete_choice, no two discrete_choices are allowed to cover the same value (see Section 4.8.1 [3.8.1], page 134); if there is no others choice, the discrete_choices taken together shall exactly cover a contiguous sequence of values of the corresponding index type.

19

A bottom level subaggregate of a multidimensional array_aggregate of a given array type is allowed to be a string_literal only if the component type of the array type is a character type; each character of such a string_literal shall correspond to a defining_character_literal of the component type.

Static Semantics

20

A subaggregate that is a string_literal is equivalent to one that is a positional_array_aggregate of the same length, with each expression being the character_literal for the corresponding character of the string_literal.

Dynamic Semantics

21

The evaluation of an array_aggregate of a given array type proceeds in two steps:

22

1. Any discrete_choices of this aggregate and of its subaggregates are evaluated in an arbitrary order, and converted to the corresponding index type;

23

2. The array component expressions of the aggregate are evaluated in an arbitrary order and their values are converted to the component subtype of the array type; an array component expression is evaluated once for each associated component.

23.1/2

Each expression in an `array_component_association` defines the value for the associated component(s). For an `array_component_association` with `<>`, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see Section 4.3.1 [3.3.1], page 61).

24

The bounds of the index range of an `array_aggregate` (including a subaggregate) are determined as follows:

25

- For an `array_aggregate` with an `others choice`, the bounds are those of the corresponding index range from the applicable index constraint;

26

- For a `positional_array_aggregate` (or equivalent `string_literal`) without an `others choice`, the lower bound is that of the corresponding index range in the applicable index constraint, if defined, or that of the corresponding index subtype, if not; in either case, the upper bound is determined from the lower bound and the number of expressions (or the length of the `string_literal`);

27

- For a `named_array_aggregate` without an `others choice`, the bounds are determined by the smallest and largest index values covered by any `discrete_choice_list`.

28

For an `array_aggregate`, a check is made that the index range defined by its bounds is compatible with the corresponding index subtype.

29

For an `array_aggregate` with an `others choice`, a check is made that no expression is specified for an index value outside the bounds determined by the applicable index constraint.

30

For a multidimensional `array_aggregate`, a check is made that all subaggregates that correspond to the same index have the same bounds.

31

The exception `Constraint_Error` is raised if any of the above checks fail.

NOTES

32/2

10 In an `array_aggregate`, positional notation may only be used with two or more expressions; a single expression in parentheses is interpreted as a parenthesized expression. A `named_array_aggregate`,

such as (1 => X), may be used to specify an array with a single component.

Examples

33

<Examples of array aggregates with positional associations:>

34

```
(7, 9, 5, 1, 3, 2, 4, 8, 6, 0)
Table'(5, 8, 4, 1, others => 0)  --< see Section 4.6 [3.6], page 114 >■
```

35

<Examples of array aggregates with named associations:>

36

```
(1 .. 5 => (1 .. 8 => 0.0))      --< two-dimensional>
(1 .. N => new Cell)           --< N new cells, in particular for N = 0>■
```

37

```
Table'(2 | 4 | 10 => 1, others => 0)
Schedule'(Mon .. Fri => True, others => False)  --< see Section 4.6
[3.6], page 114>■
Schedule'(Wed | Sun => False, others => True)
Vector'(1 => 2.5)                          --< single-component vector>■
```

38

<Examples of two-dimensional array aggregates:>

39

```
--< Three aggregates for the same value of subtype Matrix(1..2,1..3) (see Section
[3.6], page 114):>
```

40

```
((1.1, 1.2, 1.3), (2.1, 2.2, 2.3))
(1 => (1.1, 1.2, 1.3), 2 => (2.1, 2.2, 2.3))
(1 => (1 => 1.1, 2 => 1.2, 3 => 1.3), 2 => (1 => 2.1, 2 => 2.2, 3 => 2.3))■
```

41

<Examples of aggregates as initial values:>

42

```
A : Table := (7, 9, 5, 1, 3, 2, 4, 8, 6, 0);      --< A(1)=7, A(10)=0>■
B : Table := (2 | 4 | 10 => 1, others => 0);      --< B(1)=0, B(10)=1>■
C : constant Matrix := (1 .. 5 => (1 .. 8 => 0.0)); --< C'Last(1)=5, C'Last(2)=8>■
```

43

```
D : Bit_Vector(M .. N) := (M .. N => True);      --< see Section 4.6
[3.6], page 114>■
```



```

E : Bit_Vector(M .. N) := (others => True);
F : String(1 .. 1) := (1 => 'F'); --< a one component aggregate: same as "F">

```

44/2

<Example of an array aggregate with defaulted others choice and with an applicable index constraint provided by an enclosing record aggregate:>

45/2

```

Buffer'(Size => 50, Pos => 1, Value => String('x', others => <>)) --< see Section
[3.7], page 123>

```

5.4 4.4 Expressions

1

An <expression> is a formula that defines the computation or retrieval of a value. In this International Standard, the term "expression" refers to a construct of the syntactic category expression or of any of the other five syntactic categories defined below.

Syntax

2

```

expression ::=
    relation {and relation} | relation {and then relation}
    | relation {or relation} | relation {or else relation}
    | relation {xor relation}

```

3

```

relation ::=
    simple_expression [relational_operator simple_expression]
    | simple_expression [not] in range
    | simple_expression [not] in subtype_mark

```

4

```

simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}

```

5

```

term ::= factor {multiplying_operator factor}

```

6

```

factor ::= primary [** primary] | abs primary | not primary

```

7

```

primary ::=
    numeric_literal | null | string_literal | aggregate
    | name | qualified_expression | allocator | (expression)

```

Name Resolution Rules

8

A name used as a primary shall resolve to denote an object or a value.

Static Semantics

9

Each expression has a type; it specifies the computation or retrieval of a value of that type.

Dynamic Semantics

10

The value of a primary that is a name denoting an object is the value of the object.

Implementation Permissions

11

For the evaluation of a primary that is a name denoting an object of an unconstrained numeric subtype, if the value of the object is outside the base range of its type, the implementation may either raise `Constraint_Error` or return the value of the object.

Examples

12

<Examples of primaries:>

13

```
4.0          --< real literal>
Pi           --< named number>
(1 .. 10 => 0) --< array aggregate>
Sum          --< variable>
Integer'Last --< attribute>
Sine(X)      --< function call>
Color'(Blue) --< qualified expression>
Real(M*N)    --< conversion>
(Line_Count + 10) --< parenthesized expression >
```

14

<Examples of expressions:>

15/2

```
Volume          --< primary>
not Destroyed   --< factor>
2*Line_Count    --< term>
-4.0            --< simple expression>
-4.0 + A        --< simple expression>
B**2 - 4.0*A*C  --< simple expression>
R*Sin([Unicode 952])*Cos([Unicode 966]) --< simple expression>█
Password(1 .. 3) = "Bwv" --< relation>
Count in Small_Int --< relation>
Count not in Small_Int --< relation>
Index = 0 or Item_Hit --< expression>
(Cold and Sunny) or Warm --< expression (parentheses are required)>█
A**(B**C)       --< expression (parentheses are required)>█
```

5.5 4.5 Operators and Expression Evaluation

1

The language defines the following six categories of operators (given in order of increasing precedence). The corresponding operator_symbols, and only those, can be used as designators in declarations of functions for user-defined operators. See Section 7.6 [6.6], page 276, "Section 7.6 [6.6], page 276, Overloading of Operators".

Syntax

2

logical_operator ::= and | or | xor

3

relational_operator ::= = | /= | < | <= | > | >=

4

binary_adding_operator ::= + | - | &

5

unary_adding_operator ::= + | -

6

multiplying_operator ::= * | / | mod | rem

7

highest_precedence_operator ::= ** | abs | not

Static Semantics

8

For a sequence of operators of the same precedence level, the operators are associated with their operands in textual order from left to right. Parentheses can be used to impose specific associations.

9

For each form of type definition, certain of the above operators are <predefined>; that is, they are implicitly declared immediately after the type definition. For each such implicit operator declaration, the parameters are called Left and Right for <binary> operators; the single parameter is called Right for <unary> operators. An expression of the form X op Y, where op is a binary operator, is equivalent to a function_call of the form "op"(X, Y). An expression of the form op Y, where op is a unary operator, is equivalent to a function_call of the form "op"(Y). The predefined operators and their effects are described in subclauses Section 5.5.1 [4.5.1], page 204, through Section 5.5.6 [4.5.6], page 217.

Dynamic Semantics

10

The predefined operations on integer types either yield the mathematically correct result or raise the exception Constraint_Error. For implementations that support the Numerics

Annex, the predefined operations on real types yield results whose accuracy is defined in Chapter 21 [Annex G], page 1083, or raise the exception `Constraint_Error`.

Implementation Requirements

11

The implementation of a predefined operator that delivers a result of an integer or fixed point type may raise `Constraint_Error` only if the result is outside the base range of the result type.

12

The implementation of a predefined operator that delivers a result of a floating point type may raise `Constraint_Error` only if the result is outside the safe range of the result type.

Implementation Permissions

13

For a sequence of predefined operators of the same precedence level (and in the absence of parentheses imposing a specific association), an implementation may impose any association of the operators with operands so long as the result produced is an allowed result for the left-to-right association, but ignoring the potential for failure of language-defined checks in either the left-to-right or chosen order of association.

NOTES

14

11 The two operands of an expression of the form `X op Y`, where `op` is a binary operator, are evaluated in an arbitrary order, as for any `function_call` (see Section 7.4 [6.4], page 266).

Examples

15

<Examples of precedence:>

16

```
not Sunny or Warm    --< same as (not Sunny) or Warm>
X > 4.0 and Y > 0.0  --< same as (X > 4.0) and (Y > 0.0)>
```

17

```
-4.0*A**2            --< same as -(4.0 * (A**2))>
abs(1 + A) + B       --< same as (abs (1 + A)) + B>
Y**(-3)              --< parentheses are necessary>
A / B * C            --< same as (A/B)*C>
A + (B + C)          --< evaluate B + C before adding it to A >
```

5.5.1 4.5.1 Logical Operators and Short-circuit Control Forms

Name Resolution Rules

1

An expression consisting of two relations connected by `and` then `or` or `else` (a <short-circuit control form>) shall resolve to be of some boolean type; the expected type for both relations is that same boolean type.

Static Semantics

2

The following logical operators are predefined for every boolean type $\langle T \rangle$, for every modular type $\langle T \rangle$, and for every one-dimensional array type $\langle T \rangle$ whose component type is a boolean type:

3

```
function "and"(Left, Right :  $\langle T \rangle$ ) return  $\langle T \rangle$ 
function "or" (Left, Right :  $\langle T \rangle$ ) return  $\langle T \rangle$ 
function "xor"(Left, Right :  $\langle T \rangle$ ) return  $\langle T \rangle$ 
```

4

For boolean types, the predefined logical operators and, or, and xor perform the conventional operations of conjunction, inclusive disjunction, and exclusive disjunction, respectively.

5

For modular types, the predefined logical operators are defined on a bit-by-bit basis, using the binary representation of the value of the operands to yield a binary representation for the result, where zero represents False and one represents True. If this result is outside the base range of the type, a final subtraction by the modulus is performed to bring the result into the base range of the type.

6

The logical operators on arrays are performed on a component-by-component basis on matching components (as for equality — see Section 5.5.2 [4.5.2], page 206), using the predefined logical operator for the component type. The bounds of the resulting array are those of the left operand.

Dynamic Semantics

7

The short-circuit control forms and then and or else deliver the same result as the corresponding predefined and and or operators for boolean types, except that the left operand is always evaluated first, and the right operand is not evaluated if the value of the left operand determines the result.

8

For the logical operators on arrays, a check is made that for each component of the left operand there is a matching component of the right operand, and vice versa. Also, a check is made that each component of the result belongs to the component subtype. The exception `Constraint_Error` is raised if either of the above checks fails.

NOTES

9

12 The conventional meaning of the logical operators is given by the following truth table:

10

A	B	(A and B)	(A or B)	(A xor B)
---	---	-----------	----------	-----------

```

True  True  True  True  False
True  False False True  True
False True  False True  True
False False False False False

```

Examples

11

<Examples of logical operators:>

12

```

Sunny or Warm
Filter(1 .. 10) and Filter(15 .. 24)  --<  see Section 4.6.1 [3.6.1],
page 117 >

```

13

<Examples of short-circuit control forms:>

14

```

Next_Car.Owner /= null and then Next_Car.Owner.Age > 25  --<  see Section 4.10.
[3.10.1], page 160>
N = 0 or else A(N) = Hit_Value

```

5.5.2 4.5.2 Relational Operators and Membership Tests

1

The <equality operators> = (equals) and /= (not equals) are predefined for nonlimited types. The other relational operators are the <ordering operators> < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). The ordering operators are predefined for scalar types, and for <discrete array types>, that is, one-dimensional array types whose components are of a discrete type.

2

A <membership test>, using in or not in, determines whether or not a value belongs to a given subtype or range, or has a tag that identifies a type that is covered by a given type. Membership tests are allowed for all types.

Name Resolution Rules

3/2

The <tested type> of a membership test is the type of the range or the type determined by the subtype_mark. If the tested type is tagged, then the simple_expression shall resolve to be of a type that is convertible (see Section 5.6 [4.6], page 219) to the tested type; if untagged, the expected type for the simple_expression is the tested type.

Legality Rules

4

For a membership test, if the simple_expression is of a tagged class-wide type, then the tested type shall be (visibly) tagged.

Static Semantics

5

The result type of a membership test is the predefined type Boolean.

6

The equality operators are predefined for every specific type `<T>` that is not limited, and not an anonymous access type, with the following specifications:

7

```
function "=" (Left, Right : <T>) return Boolean
function "/="(Left, Right : <T>) return Boolean
```

7.1/2

The following additional equality operators for the `<universal_access>` type are declared in package `Standard` for use with anonymous access types:

7.2/2

```
function "=" (Left, Right : <universal_access>) return Boolean
function "/="(Left, Right : <universal_access>) return Boolean
```

8

The ordering operators are predefined for every specific scalar type `<T>`, and for every discrete array type `<T>`, with the following specifications:

9

```
function "<" (Left, Right : <T>) return Boolean
function "<=" (Left, Right : <T>) return Boolean
function ">" (Left, Right : <T>) return Boolean
function ">=" (Left, Right : <T>) return Boolean
```

Name Resolution Rules

9.1/2

At least one of the operands of an equality operator for `<universal_access>` shall be of a specific anonymous access type. Unless the predefined equality operator is identified using an expanded name with prefix denoting the package `Standard`, neither operand shall be of an access-to-object type whose designated type is `<D>` or `<D>'Class`, where `<D>` has a user-defined primitive equality operator such that:

9.2/2

- its result type is `Boolean`;

9.3/2

- it is declared immediately within the same declaration list as `<D>`; and

9.4/2

- at least one of its operands is an access parameter with designated type `<D>`.

Legality Rules

9.5/2

At least one of the operands of the equality operators for `<universal_access>` shall be of

type <universal_access>, or both shall be of access-to-object types, or both shall be of access-to-subprogram types. Further:

9.6/2

- When both are of access-to-object types, the designated types shall be the same or one shall cover the other, and if the designated types are elementary or array types, then the designated subtypes shall statically match;

9.7/2

- When both are of access-to-subprogram types, the designated profiles shall be subtype conformant.

Dynamic Semantics

10

For discrete types, the predefined relational operators are defined in terms of corresponding mathematical operations on the position numbers of the values of the operands.

11

For real types, the predefined relational operators are defined in terms of the corresponding mathematical operations on the values of the operands, subject to the accuracy of the type.

12

Two access-to-object values are equal if they designate the same object, or if both are equal to the null value of the access type.

13

Two access-to-subprogram values are equal if they are the result of the same evaluation of an Access attribute_reference, or if both are equal to the null value of the access type. Two access-to-subprogram values are unequal if they designate different subprograms. It is unspecified whether two access values that designate the same subprogram but are the result of distinct evaluations of Access attribute_references are equal or unequal.

14

For a type extension, predefined equality is defined in terms of the primitive (possibly user-defined) equals operator of the parent type and of any tagged components of the extension part, and predefined equality for any other components not inherited from the parent type.

15

For a private type, if its full type is tagged, predefined equality is defined in terms of the primitive equals operator of the full type; if the full type is untagged, predefined equality for the private type is that of its full type.

16

For other composite types, the predefined equality operators (and certain other predefined operations on composite types — see Section 5.5.1 [4.5.1], page 204, and Section 5.6 [4.6], page 219) are defined in terms of the corresponding operation on <matching components>, defined as follows:

17

- For two composite objects or values of the same non-array type, matching components are those that correspond to the same `component_declaration` or `discriminant_specification`;

18

- For two one-dimensional arrays of the same type, matching components are those (if any) whose index values match in the following sense: the lower bounds of the index ranges are defined to match, and the successors of matching indices are defined to match;

19

- For two multidimensional arrays of the same type, matching components are those whose index values match in successive index positions.

20

The analogous definitions apply if the types of the two objects or values are convertible, rather than being the same.

21

Given the above definition of matching components, the result of the predefined equals operator for composite types (other than for those composite types covered earlier) is defined as follows:

22

- If there are no components, the result is defined to be `True`;

23

- If there are unmatched components, the result is defined to be `False`;

24

- Otherwise, the result is defined in terms of the primitive equals operator for any matching tagged components, and the predefined equals for any matching untagged components.

24.1/1

For any composite type, the order in which `"=`" is called for components is unspecified. Furthermore, if the result can be determined before calling `"=`" on some components, it is unspecified whether `"=`" is called on those components.

25

The predefined `"/=`" operator gives the complementary result to the predefined `"=`" operator.

26

For a discrete array type, the predefined ordering operators correspond to `<lexicographic order>` using the predefined order relation of the component type: A null array is lexicographically less than any array having at least one component. In the case of nonnull arrays,

the left operand is lexicographically less than the right operand if the first component of the left operand is less than that of the right; otherwise the left operand is lexicographically less than the right operand only if their first components are equal and the tail of the left operand is lexicographically less than that of the right (the <tail> consists of the remaining components beyond the first and can be null).

27

For the evaluation of a membership test, the `simple_expression` and the range (if any) are evaluated in an arbitrary order.

28

A membership test using `in` yields the result `True` if:

29

- The tested type is scalar, and the value of the `simple_expression` belongs to the given range, or the range of the named subtype; or

30/2

- The tested type is not scalar, and the value of the `simple_expression` satisfies any constraints of the named subtype, and:

30.1/2

- if the type of the `simple_expression` is class-wide, the value has a tag that identifies a type covered by the tested type;

30.2/2

- if the tested type is an access type and the named subtype excludes null, the value of the `simple_expression` is not null.

31

Otherwise the test yields the result `False`.

32

A membership test using `not in` gives the complementary result to the corresponding membership test using `in`.

Implementation Requirements

32.1/1

For all nonlimited types declared in language-defined packages, the `"="` and `"!="` operators of the type shall behave as if they were the predefined equality operators for the purposes of the equality of composite types and generic formal types.

NOTES

33/2

<This paragraph was deleted.>

34

13 If a composite type has components that depend on discriminants, two values of this type have matching components if and only if their discriminants are equal. Two nonnull arrays have matching components if and only if the length of each dimension is the same for both.

Examples

35

<Examples of expressions involving relational operators and membership tests:>

36

```
X /= Y
```

37

```
" " < "A" and "A" < "Aa"      --< True>
"Aa" < "B" and "A" < "A  "    --< True>
```

38

```
My_Car = null                --< true if My_Car has been set to null (see Section
[3.10.1], page 160)>
My_Car = Your_Car           --< true if we both share the same car>
My_Car.all = Your_Car.all   --< true if the two cars are identical>
```

39

```
N not in 1 .. 10             --< range membership test>
Today in Mon .. Fri         --< range membership test>
Today in Weekday            --< subtype membership test (see Section 4.5.1
[3.5.1], page 92)>
Archive in Disk_Unit        --< subtype membership test (see Section 4.8.1
[3.8.1], page 134)>
Tree.all in Addition'Class  --< class membership test (see Section 4.9.1
[3.9.1], page 143)>
```

5.5.3 4.5.3 Binary Adding Operators

Static Semantics

1

The binary adding operators + (addition) and - (subtraction) are predefined for every specific numeric type <T> with their conventional meaning. They have the following specifications:

2

```
function "+"(Left, Right : <T>) return <T>
```

```
function "-"(Left, Right : <T>) return <T>
```

3

The concatenation operators & are predefined for every nonlimited, one-dimensional array type <T> with component type <C>. They have the following specifications:

4

```
function "&"(Left : <T>; Right : <T>) return <T>
function "&"(Left : <T>; Right : <C>) return <T>
function "&"(Left : <C>; Right : <T>) return <T>
function "&"(Left : <C>; Right : <C>) return <T>
```

Dynamic Semantics

5

For the evaluation of a concatenation with result type <T>, if both operands are of type <T>, the result of the concatenation is a one-dimensional array whose length is the sum of the lengths of its operands, and whose components comprise the components of the left operand followed by the components of the right operand. If the left operand is a null array, the result of the concatenation is the right operand. Otherwise, the lower bound of the result is determined as follows:

6

- If the ultimate ancestor of the array type was defined by a `constrained_array_definition`, then the lower bound of the result is that of the index subtype;

7

- If the ultimate ancestor of the array type was defined by an `unconstrained_array_definition`, then the lower bound of the result is that of the left operand.

8

The upper bound is determined by the lower bound and the length. A check is made that the upper bound of the result of the concatenation belongs to the range of the index subtype, unless the result is a null array. `Constraint_Error` is raised if this check fails.

9

If either operand is of the component type <C>, the result of the concatenation is given by the above rules, using in place of such an operand an array having this operand as its only component (converted to the component subtype) and having the lower bound of the index subtype of the array type as its lower bound.

10

The result of a concatenation is defined in terms of an assignment to an anonymous object, as for any function call (see Section 7.5 [6.5], page 272).

NOTES

11

14 As for all predefined operators on modular types, the binary adding operators + and − on modular types include a final reduction modulo the modulus if the result is outside the base range of the type.

Examples

12

<Examples of expressions involving binary adding operators:>

13

```
Z + 0.1      --< Z has to be of a real type >
```

14

```
"A" & "BCD"  --< concatenation of two string literals>
'A' & "BCD"  --< concatenation of a character literal and a string literal>
'A' & 'A'    --< concatenation of two character literals >
```

5.5.4 4.5.4 Unary Adding Operators

Static Semantics

1

The unary adding operators + (identity) and − (negation) are predefined for every specific numeric type <T> with their conventional meaning. They have the following specifications:

2

```
function "+"(Right : <T>) return <T>
function "-"(Right : <T>) return <T>
```

NOTES

3

15 For modular integer types, the unary adding operator −, when given a nonzero operand, returns the result of subtracting the value of the operand from the modulus; for a zero operand, the result is zero.

5.5.5 4.5.5 Multiplying Operators

Static Semantics

1

The multiplying operators * (multiplication), / (division), mod (modulus), and rem (remainder) are predefined for every specific integer type <T>:

2

```
function "*" (Left, Right : <T>) return <T>
function "/" (Left, Right : <T>) return <T>
function "mod"(Left, Right : <T>) return <T>
function "rem"(Left, Right : <T>) return <T>
```

3

Signed integer multiplication has its conventional meaning.

4

Signed integer division and remainder are defined by the relation:

5

$$A = (A/B)*B + (A \text{ rem } B)$$

6

where $(A \text{ rem } B)$ has the sign of A and an absolute value less than the absolute value of B . Signed integer division satisfies the identity:

7

$$(-A)/B = -(A/B) = A/(-B)$$

8

The signed integer modulus operator is defined such that the result of $A \text{ mod } B$ has the sign of B and an absolute value less than the absolute value of B ; in addition, for some signed integer value N , this result satisfies the relation:

9

$$A = B*N + (A \text{ mod } B)$$

10

The multiplying operators on modular types are defined in terms of the corresponding signed integer operators, followed by a reduction modulo the modulus if the result is outside the base range of the type (which is only possible for the "*" operator).

11

Multiplication and division operators are predefined for every specific floating point type $\langle T \rangle$:

12

```
function "*" (Left, Right : <T>) return <T>
```

```
function "/" (Left, Right : <T>) return <T>
```

13

The following multiplication and division operators, with an operand of the predefined type Integer, are predefined for every specific fixed point type $\langle T \rangle$:

14

```
function "*" (Left : <T>; Right : Integer) return <T>
```

```
function "*" (Left : Integer; Right : <T>) return <T>
```

```
function "/" (Left : <T>; Right : Integer) return <T>
```

15

All of the above multiplying operators are usable with an operand of an appropriate universal numeric type. The following additional multiplying operators for $\langle \text{root_real} \rangle$ are

predefined, and are usable when both operands are of an appropriate universal or root numeric type, and the result is allowed to be of type `<root_real>`, as in a `number_declaration`:

16

```
function "*" (Left, Right : <root_real>) return <root_real>
function "/" (Left, Right : <root_real>) return <root_real>
```

17

```
function "*" (Left : <root_real>; Right : <root_integer>) return <root_real>
function "*" (Left : <root_integer>; Right : <root_real>) return <root_real>
function "/" (Left : <root_real>; Right : <root_integer>) return <root_real>
```

18

Multiplication and division between any two fixed point types are provided by the following two predefined operators:

19

```
function "*" (Left, Right : <universal_fixed>) return <universal_fixed>
function "/" (Left, Right : <universal_fixed>) return <universal_fixed>
```

Name Resolution Rules

19.1/2

The above two fixed–fixed multiplying operators shall not be used in a context where the expected type for the result is itself `<universal_fixed>` — the context has to identify some other numeric type to which the result is to be converted, either explicitly or implicitly. Unless the predefined universal operator is identified using an expanded name with prefix denoting the package `Standard`, an explicit conversion is required on the result when using the above fixed–fixed multiplication operator if either operand is of a type having a user–defined primitive multiplication operator such that:

19.2/2

- it is declared immediately within the same declaration list as the type; and

19.3/2

- both of its formal parameters are of a fixed–point type.

19.4/2

A corresponding requirement applies to the universal fixed–fixed division operator.

Legality Rules

20/2

<This paragraph was deleted.>

Dynamic Semantics

21

The multiplication and division operators for real types have their conventional meaning. For floating point types, the accuracy of the result is determined by the precision of the

result type. For decimal fixed point types, the result is truncated toward zero if the mathematical result is between two multiples of the <small> of the specific result type (possibly determined by context); for ordinary fixed point types, if the mathematical result is between two multiples of the <small>, it is unspecified which of the two is the result.

22

The exception `Constraint_Error` is raised by integer division, `rem`, and `mod` if the right operand is zero. Similarly, for a real type `<T>` with `<T>Machine_Overflows` `True`, division by zero raises `Constraint_Error`.

NOTES

23

16 For positive `A` and `B`, `A/B` is the quotient and `A rem B` is the remainder when `A` is divided by `B`. The following relations are satisfied by the `rem` operator:

24

$$\begin{aligned} A \text{ rem } (-B) &= A \text{ rem } B \\ (-A) \text{ rem } B &= -(A \text{ rem } B) \end{aligned}$$

25

17 For any signed integer `K`, the following identity holds:

26

$$A \text{ mod } B = (A + K*B) \text{ mod } B$$

27

The relations between signed integer division, remainder, and modulus are illustrated by the following table:

28

29

A	B	A/B	A rem B	A mod B	A	B	A/B	A rem B	A mo
10	5	2	0	0	-10	5	-2	0	
11	5	2	1	1	-11	5	-2	-1	
12	5	2	2	2	-12	5	-2	-2	
13	5	2	3	3	-13	5	-2	-3	
14	5	2	4	4	-14	5	-2	-4	

30

A	B	A/B	A rem B	A mod B	A	B	A/B	A rem B	A mo
10	-5	-2	0	0	-10	-5	2	0	

11	-5	-2	1	-4	-11	-5	2	-1
12	-5	-2	2	-3	-12	-5	2	-2
13	-5	-2	3	-2	-13	-5	2	-3
14	-5	-2	4	-1	-14	-5	2	-4

Examples

31

<Examples of expressions involving multiplying operators:>

32

```
I : Integer := 1;
J : Integer := 2;
K : Integer := 3;
```

33

```
X : Real := 1.0;           --< see Section 4.5.7 [3.5.7],
page 103>
Y : Real := 2.0;
```

34

```
F : Fraction := 0.25;     --< see Section 4.5.9 [3.5.9],
page 106>
G : Fraction := 0.5;
```

35

<Expression>	<Value>	<Result Type>
I*J	2	<same as I and J, that is, Integer>
K/J	1	<same as K and J, that is, Integer>
K mod J	1	<same as K and J, that is, Integer>
X/Y	0.5	<same as X and Y, that is, Real>
F/2	0.125	<same as F, that is, Fraction>
3*F	0.75	<same as F, that is, Fraction>
0.75*G	0.375	<universal_fixed, implicitly convertible> <to any fixed point type>
Fraction(F*G)	0.125	<Fraction, as stated by the conversion>
Real(J)*Y	4.0	<Real, the type of both operands after> <conversion of J>

5.5.6 4.5.6 Highest Precedence Operators

Static Semantics

1

The highest precedence unary operator `abs` (absolute value) is predefined for every specific numeric type `<T>`, with the following specification:

2

```
function "abs"(Right : <T>) return <T>
```

3

The highest precedence unary operator not (logical negation) is predefined for every boolean type <T>, every modular type <T>, and for every one-dimensional array type <T> whose components are of a boolean type, with the following specification:

4

```
function "not"(Right : <T>) return <T>
```

5

The result of the operator not for a modular type is defined as the difference between the high bound of the base range of the type and the value of the operand. For a binary modulus, this corresponds to a bit-wise complement of the binary representation of the value of the operand.

6

The operator not that applies to a one-dimensional array of boolean components yields a one-dimensional boolean array with the same bounds; each component of the result is obtained by logical negation of the corresponding component of the operand (that is, the component that has the same index value). A check is made that each component of the result belongs to the component subtype; the exception `Constraint_Error` is raised if this check fails.

7

The highest precedence <exponentiation> operator `**` is predefined for every specific integer type <T> with the following specification:

8

```
function "**"(Left : <T>; Right : Natural) return <T>
```

9

Exponentiation is also predefined for every specific floating point type as well as `<root_real>`, with the following specification (where <T> is `<root_real>` or the floating point type):

10

```
function "**"(Left : <T>; Right : Integer'Base) return <T>
```

11

The right operand of an exponentiation is the <exponent>. The expression `X**N` with the value of the exponent `N` positive is equivalent to the expression `X*X*...X` (with `N-1` multiplications) except that the multiplications are associated in an arbitrary order. With `N` equal to zero, the result is one. With the value of `N` negative (only defined for a floating point operand), the result is the reciprocal of the result using the absolute value of `N` as the exponent.

Implementation Permissions

12

The implementation of exponentiation for the case of a negative exponent is allowed to

raise `Constraint_Error` if the intermediate result of the repeated multiplications is outside the safe range of the type, even though the final result (after taking the reciprocal) would not be. (The best machine approximation to the final result in this case would generally be 0.0.)

NOTES

13

18 As implied by the specification given above for exponentiation of an integer type, a check is made that the exponent is not negative. `Constraint_Error` is raised if this check fails.

5.6 4.6 Type Conversions

1

Explicit type conversions, both value conversions and view conversions, are allowed between closely related types as defined below. This clause also defines rules for value and view conversions to a particular subtype of a type, both explicit ones and those implicit in other constructs.

Syntax

2

```
type_conversion ::=
  subtype_mark(expression)
  | subtype_mark(name)
```

3

The <target subtype> of a `type_conversion` is the subtype denoted by the `subtype_mark`. The <operand> of a `type_conversion` is the expression or name within the parentheses; its type is the <operand type>.

4

One type is <convertible> to a second type if a `type_conversion` with the first type as operand type and the second type as target type is legal according to the rules of this clause. Two types are convertible if each is convertible to the other.

5/2

A `type_conversion` whose operand is the name of an object is called a <view conversion> if both its target type and operand type are tagged, or if it appears in a call as an actual parameter of mode out or in out; other `type_conversions` are called <value conversions>.

Name Resolution Rules

6

The operand of a `type_conversion` is expected to be of any type.

7

The operand of a view conversion is interpreted only as a name; the operand of a value conversion is interpreted as an expression.

Legality Rules

8/2

In a view conversion for an untagged type, the target type shall be convertible (back) to the operand type.

<Paragraphs 9 through 20 were reorganized and moved below.>

21/2

If there is a type that is an ancestor of both the target type and the operand type, or both types are class-wide types, then at least one of the following rules shall apply:

21.1/2

- The target type shall be untagged; or

22

- The operand type shall be covered by or descended from the target type; or

23/2

- The operand type shall be a class-wide type that covers the target type; or

23.1/2

- The operand and target types shall both be class-wide types and the specific type associated with at least one of them shall be an interface type.

24/2

If there is no type that is the ancestor of both the target type and the operand type, and they are not both class-wide types, one of the following rules shall apply:

24.1/2

- If the target type is a numeric type, then the operand type shall be a numeric type.

24.2/2

- If the target type is an array type, then the operand type shall be an array type.
Further:

24.3/2

- The types shall have the same dimensionality;

24.4/2

- Corresponding index types shall be convertible;

24.5/2

- The component subtypes shall statically match;

24.6/2

- If the component types are anonymous access types, then the accessibility level of the operand type shall not be statically deeper than that of the target type;

24.7/2

- Neither the target type nor the operand type shall be limited;

24.8/2

- If the target type of a view conversion has aliased components, then so shall the operand type; and

24.9/2

- The operand type of a view conversion shall not have a tagged, private, or volatile subcomponent.

24.10/2

- If the target type is <universal_access>, then the operand type shall be an access type.

24.11/2

- If the target type is a general access-to-object type, then the operand type shall be <universal_access> or an access-to-object type. Further, if the operand type is not <universal_access>:

24.12/2

- If the target type is an access-to-variable type, then the operand type shall be an access-to-variable type;

24.13/2

- If the target designated type is tagged, then the operand designated type shall be convertible to the target designated type;

24.14/2

- If the target designated type is not tagged, then the designated types shall be the same, and either:

24.15/2

- the designated subtypes shall statically match; or

24.16/2

- the designated type shall be discriminated in its full view and unconstrained in any partial view, and one of the designated subtypes shall be unconstrained;

24.17/2

- The accessibility level of the operand type shall not be statically deeper than that of the target type. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit.

24.18/2

- If the target type is a pool-specific access-to-object type, then the operand type shall be <universal_access>.

24.19/2

- If the target type is an access-to-subprogram type, then the operand type shall be <universal_access> or an access-to-subprogram type. Further, if the operand type is not <universal_access>:

24.20/2

- The designated profiles shall be subtype-conformant.

24.21/2

- The accessibility level of the operand type shall not be statically deeper than that of the target type. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit. If the operand type is declared within a generic body, the target type shall be declared within the generic body.

Static Semantics

25

A `type_conversion` that is a value conversion denotes the value that is the result of converting the value of the operand to the target subtype.

26

A `type_conversion` that is a view conversion denotes a view of the object denoted by the operand. This view is a variable of the target type if the operand denotes a variable; otherwise it is a constant of the target type.

27

The nominal subtype of a `type_conversion` is its target subtype.

Dynamic Semantics

28

For the evaluation of a `type_conversion` that is a value conversion, the operand is evaluated, and then the value of the operand is <converted> to a <corresponding> value of the target type, if any. If there is no value of the target type that corresponds to the operand value, `Constraint_Error` is raised; this can only happen on conversion to a modular type, and only when the operand value is outside the base range of the modular type. Additional rules follow:

29

- Numeric Type Conversion

30

- If the target and the operand types are both integer types, then the result is the value of the target type that corresponds

to the same mathematical integer as the operand.

31

- If the target type is a decimal fixed point type, then the result is truncated (toward 0) if the value of the operand is not a multiple of the <small> of the target type.

32

- If the target type is some other real type, then the result is within the accuracy of the target type (see Section 21.2 [G.2], page 1103, "Section 21.2 [G.2], page 1103, Numeric Performance Requirements", for implementations that support the Numerics Annex).

33

- If the target type is an integer type and the operand type is real, the result is rounded to the nearest integer (away from zero if exactly halfway between two integers).

34

- Enumeration Type Conversion

35

- The result is the value of the target type with the same position number as that of the operand value.

36

- Array Type Conversion

37

- If the target subtype is a constrained array subtype, then a check is made that the length of each dimension of the value of the operand equals the length

of the corresponding dimension of the target subtype. The bounds of the result are those of the target subtype.

38

- If the target subtype is an unconstrained array subtype, then the bounds of the result are obtained by converting each bound of the value of the operand to the corresponding index type of the target type. For each nonnull index range, a check is made that the bounds of the range belong to the corresponding index subtype.

39

- In either array case, the value of each component of the result is that of the matching component of the operand value (see Section 5.5.2 [4.5.2], page 206).

39.1/2

- If the component types of the array types are anonymous access types, then a check is made that the accessibility level of the operand type is not deeper than that of the target type.

40

- Composite (Non-Array) Type Conversion

41

- The value of each nondiscriminant component of the result is that of the matching component of the operand value.

42

- The tag of the result is that of the operand. If the operand type is class-wide, a check is made that the tag of the operand identifies a (specific) type that is covered by or descended from the target type.

43

- For each discriminant of the target type that corresponds to a discriminant of the operand type, its value is that of the corresponding discriminant of the operand value; if it corresponds to more than one discriminant of the operand type, a check is made that all these discriminants are equal in the operand value.

44

- For each discriminant of the target type that corresponds to a discriminant that is specified by the `derived_type_definition` for some ancestor of the operand type (or if class-wide, some ancestor of the specific type identified by the tag of the operand), its value in the result is that specified by the `derived_type_definition`.

45

- For each discriminant of the operand type that corresponds to a discriminant that is specified by the `derived_type_definition` for some ancestor of the target type, a check is made that in the operand value it equals the value specified for it.

46

- For each discriminant of the result, a check is made that its value belongs to its subtype.

47

- Access Type Conversion

48

- For an access-to-object type, a check is made that the accessibility level of the operand type is not deeper than that of the target type.

49/2

- If the operand value is null, the result of the conversion is the null value of the target type.

50

- If the operand value is not null, then the result designates the same object (or sub-program) as is designated by the operand value, but viewed as being of the target designated subtype (or profile); any checks associated with evaluating a conversion to the target designated subtype are performed.

51/2

After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null.

52

For the evaluation of a view conversion, the operand name is evaluated, and a new view of the object denoted by the operand is created, whose type is the target type; if the target type is composite, checks are performed as above for a value conversion.

53

The properties of this new view are as follows:

54/1

- If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each nondiscriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the target subtype is indefinite, or if the operand type is a descendant of the target type and has discriminants that were not inherited from the target type;

55

- If the target type is tagged, then an assignment to the view assigns to the corresponding part of the object denoted by the operand; otherwise, an assignment to the view assigns to the object, after converting the assigned value to the subtype of the object (which might raise `Constraint_Error`);

56

- Reading the value of the view yields the result of converting the value of the operand object to the target subtype (which might raise `Constraint_Error`), except if the object

is of an access type and the view conversion is passed as an out parameter; in this latter case, the value of the operand object is used to initialize the formal parameter without checking against any constraint of the target subtype (see Section 7.4.1 [6.4.1], page 270).

57

If an `Accessibility_Check` fails, `Program_Error` is raised. Any other check associated with a conversion raises `Constraint_Error` if it fails.

58

Conversion to a type is the same as conversion to an unconstrained subtype of the type.

NOTES

59

19 In addition to explicit `type_conversions`, type conversions are performed implicitly in situations where the expected type and the actual type of a construct differ, as is permitted by the type resolution rules (see Section 9.6 [8.6], page 324). For example, an integer literal is of the type `<universal_integer>`, and is implicitly converted when assigned to a target of some specific integer type. Similarly, an actual parameter of a specific tagged type is implicitly converted when the corresponding formal parameter is of a class-wide type.

60

Even when the expected and actual types are the same, implicit subtype conversions are performed to adjust the array bounds (if any) of an operand to match the desired target subtype, or to raise `Constraint_Error` if the (possibly adjusted) value does not satisfy the constraints of the target subtype.

61/2

20 A ramification of the overload resolution rules is that the operand of an (explicit) `type_conversion` cannot be an allocator, an aggregate, a `string_literal`, a `character_literal`, or an `attribute_reference` for an `Access` or `Unchecked_Access` attribute. Similarly, such an expression enclosed by parentheses is not allowed. A `qualified_expression` (see Section 5.7 [4.7], page 229) can be used instead of such a `type_conversion`.

62

21 The constraint of the target subtype has no effect for a `type_conversion` of an elementary type passed as an out parameter. Hence, it is recommended that the first subtype be specified as the target to minimize confusion (a similar recommendation applies to renaming and generic formal in out objects).

Examples

63

<Examples of numeric type conversion:>

64

```
Real(2*J)      <-- value is converted to floating point>
Integer(1.6)   <-- value is 2>
Integer(-0.4) <-- value is 0>
```

65

<Example of conversion between derived types:>

66

```
type A_Form is new B_Form;
```

67

```
X : A_Form;
Y : B_Form;
```

68

```
X := A_Form(Y);
Y := B_Form(X); <-- the reverse conversion >
```

69

<Examples of conversions between array types:>

70

```
type Sequence is array (Integer range <>) of Integer;
subtype Dozen is Sequence(1 .. 12);
Ledger : array(1 .. 100) of Integer;
```

71

```
Sequence(Ledger)          <-- bounds are those of Ledger>
Sequence(Ledger(31 .. 42)) <-- bounds are 31 and 42>
Dozen(Ledger(31 .. 42))   <-- bounds are those of Dozen >
```

5.7 4.7 Qualified Expressions

1

A qualified_expression is used to state explicitly the type, and to verify the subtype, of an operand that is either an expression or an aggregate.

Syntax

2

```
qualified_expression ::=
  subtype_mark'(expression) | subtype_mark'aggregate
Name Resolution Rules
```

3

The <operand> (the expression or aggregate) shall resolve to be of the type determined by the subtype_mark (see [S0028], page 56), or a universal type that covers it.

Dynamic Semantics

4

The evaluation of a qualified_expression evaluates the operand (and if of a universal type, converts it to the type determined by the subtype_mark) and checks that its value belongs to the subtype denoted by the subtype_mark. The exception Constraint_Error is raised if this check fails.

NOTES

5

22 When a given context does not uniquely identify an expected type, a qualified_expression can be used to do so. In particular, if an overloaded name or aggregate is passed to an overloaded subprogram, it might be necessary to qualify the operand to resolve its type.

Examples

6

<Examples of disambiguating expressions using qualification:>

7

```
type Mask is (Fix, Dec, Exp, Signif);
type Code is (Fix, Cla, Dec, Tnz, Sub);
```

8

```
Print (Mask'(Dec)); <-- Dec is of type Mask>
Print (Code'(Dec)); <-- Dec is of type Code >
```

9

```
for J in Code'(Fix) .. Code'(Dec) loop ... <-- qualification needed for either Fi
for J in Code range Fix .. Dec loop ... <-- qualification unnecessary>■
for J in Code'(Fix) .. Dec loop ... <-- qualification unnecessary for Dec>
```

10

```
Dozen'(1 | 3 | 5 | 7 => 2, others => 0) <-- see Section 5.6 [4.6],
page 219 >
```

5.8 4.8 Allocators

1

The evaluation of an allocator creates an object and yields an access value that designates the object.

Syntax

2

allocator ::=
new subtype_indication | new qualified_expression
Name Resolution Rules

3/1

The expected type for an allocator shall be a single access-to-object type with designated type <D> such that either <D> covers the type determined by the subtype_mark of the subtype_indication (see [S0027], page 56) or qualified_expression (see [S0128], page 229), or the expected type is anonymous and the determined type is <D>'Class.

Legality Rules

4

An <initialized> allocator is an allocator with a qualified_expression. An <uninitialized> allocator is one with a subtype_indication. In the subtype_indication of an uninitialized allocator, a constraint is permitted only if the subtype_mark denotes an unconstrained composite subtype; if there is no constraint, then the subtype_mark shall denote a definite subtype.

5/2

If the type of the allocator is an access-to-constant type, the allocator shall be an initialized allocator.

5.1/2

If the designated type of the type of the allocator is class-wide, the accessibility level of the type determined by the subtype_indication or qualified_expression shall not be statically deeper than that of the type of the allocator.

5.2/2

If the designated subtype of the type of the allocator has one or more unconstrained access discriminants, then the accessibility level of the anonymous access type of each access discriminant, as determined by the subtype_indication or qualified_expression of the allocator, shall not be statically deeper than that of the type of the allocator (see Section 4.10.2 [3.10.2], page 164).

5.3/2

An allocator shall not be of an access type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit. This rule does not apply in the body of a generic unit or within a body declared within the declarative region of a generic unit, if the type of the allocator is a descendant of a formal access type declared within the formal part of the generic unit.

Static Semantics

6/2

If the designated type of the type of the allocator is elementary, then the subtype of the created object is the designated subtype. If the designated type is composite, then the subtype of the created object is the designated subtype when the designated subtype is constrained or there is a partial view of the designated type that is constrained; otherwise,

the created object is constrained by its initial value (even if the designated subtype is unconstrained with defaults).

Dynamic Semantics

7/2

For the evaluation of an initialized allocator, the evaluation of the `qualified_expression` is performed first. An object of the designated type is created and the value of the `qualified_expression` is converted to the designated subtype and assigned to the object.

8

For the evaluation of an uninitialized allocator, the elaboration of the `subtype_indication` is performed first. Then:

9/2

- If the designated type is elementary, an object of the designated subtype is created and any implicit initial value is assigned;

10/2

- If the designated type is composite, an object of the designated type is created with tag, if any, determined by the `subtype_mark` of the `subtype_indication`. This object is then initialized by default (see Section 4.3.1 [3.3.1], page 61) using the `subtype_indication` to determine its nominal subtype. A check is made that the value of the object belongs to the designated subtype. `Constraint_Error` is raised if this check fails. This check and the initialization of the object are performed in an arbitrary order.

10.1/2

For any allocator, if the designated type of the type of the allocator is class-wide, then a check is made that the accessibility level of the type determined by the `subtype_indication`, or by the tag of the value of the `qualified_expression`, is not deeper than that of the type of the allocator. If the designated subtype of the allocator has one or more unconstrained access discriminants, then a check is made that the accessibility level of the anonymous access type of each access discriminant is not deeper than that of the type of the allocator. `Program_Error` is raised if either such check fails.

10.2/2

If the object to be created by an allocator has a controlled or protected part, and the finalization of the collection of the type of the allocator (see Section 8.6.1 [7.6.1], page 299) has started, `Program_Error` is raised.

10.3/2

If the object to be created by an allocator contains any tasks, and the master of the type of the allocator is completed, and all of the dependent tasks of the master are terminated (see Section 10.3 [9.3], page 335), then `Program_Error` is raised.

11

If the created object contains any tasks, they are activated (see Section 10.2 [9.2], page 333). Finally, an access value that designates the created object is returned.

Bounded (Run-Time) Errors

11.1/2

It is a bounded error if the finalization of the collection of the type (see Section 8.6.1 [7.6.1],

page 299) of the allocator has started. If the error is detected, `Program_Error` is raised. Otherwise, the allocation proceeds normally.

NOTES

12

23 Allocators cannot create objects of an abstract type. See Section 4.9.3 [3.9.3], page 149.

13

24 If any part of the created object is controlled, the initialization includes calls on corresponding `Initialize` or `Adjust` procedures. See Section 8.6 [7.6], page 295.

14

25 As explained in Section 14.11 [13.11], page 526, "Section 14.11 [13.11], page 526, Storage Management", the storage for an object allocated by an allocator comes from a storage pool (possibly user defined). The exception `Storage_Error` is raised by an allocator if there is not enough storage. Instances of `Unchecked_Deallocation` may be used to explicitly reclaim storage.

15

26 Implementations are permitted, but not required, to provide garbage collection (see Section 14.11.3 [13.11.3], page 534).

Examples

16

<Examples of allocators:>

17

```
new Cell'(0, null, null)           <-- initialized explicitly, see
[3.10.1], page 160>
new Cell'(Value => 0, Succ => null, Pred => null) <-- initialized explicitly>■
new Cell                           <-- not initialized>■
```

18

```
new Matrix(1 .. 10, 1 .. 20)       <-- the bounds only are given>■
new Matrix'(1 .. 10 => (1 .. 20 => 0.0)) <-- initialized explicitly>■
```

19

```
new Buffer(100)                     <-- the discriminant only is gi
new Buffer'(Size => 80, Pos => 0, Value => (1 .. 80 => 'A')) <-- initialized expl
```

```

Expr_Ptr'(new Literal)                <-- allocator for access-to-class-wide
[3.9.1], page 143>
Expr_Ptr'(new Literal'(Expression with 3.5))    <-- initialized explicitly>■

```

5.9 4.9 Static Expressions and Static Subtypes

1

Certain expressions of a scalar or string type are defined to be static. Similarly, certain discrete ranges are defined to be static, and certain scalar and string subtypes are defined to be static subtypes. <Static> means determinable at compile time, using the declared properties or values of the program entities.

2

A static expression is a scalar or string expression that is one of the following:

3

- a numeric_literal;

4

- a string_literal of a static string subtype;

5

- a name that denotes the declaration of a named number or a static constant;

6

- a function_call whose <function_>name or <function_>prefix statically denotes a static function, and whose actual parameters, if any (whether given explicitly or by default), are all static expressions;

7

- an attribute_reference that denotes a scalar value, and whose prefix denotes a static scalar subtype;

8

- an attribute_reference whose prefix statically denotes a statically constrained array object or array subtype, and whose attribute_designator is First, Last, or Length, with an optional dimension;

9

- a type_conversion whose subtype_mark denotes a static scalar subtype, and whose operand is a static expression;

10

- a qualified_expression whose subtype_mark denotes a static (scalar or string) subtype, and whose operand is a static expression;

11

- a membership test whose simple_expression is a static expression, and whose range is a static range or whose subtype_mark denotes a static (scalar or string) subtype;

12

- a short-circuit control form both of whose relations are static expressions;

13

- a static expression enclosed in parentheses.

14

A name <statically denotes> an entity if it denotes the entity and:

15

- It is a direct_name, expanded name, or character_literal, and it denotes a declaration other than a renaming_declaration; or

16

- It is an attribute_reference whose prefix statically denotes some entity; or

17

- It denotes a renaming_declaration with a name that statically denotes the renamed entity.

18

A <static function> is one of the following:

19

- a predefined operator whose parameter and result types are all scalar types none of which are descendants of formal scalar types;

20

- a predefined concatenation operator whose result type is a string type;

21

- an enumeration literal;

22

- a language–defined attribute that is a function, if the prefix denotes a static scalar subtype, and if the parameter and result types are scalar.

23

In any case, a generic formal subprogram is not a static function.

24

A <static constant> is a constant view declared by a full constant declaration or an object_renaming_declaration (see [S0183], page 317) with a static nominal subtype, having a value defined by a static scalar expression or by a static string expression whose value has a length not exceeding the maximum length of a string_literal (see [S0016], page 42) in the implementation.

25

A <static range> is a range whose bounds are static expressions, or a range_attribute_reference (see [S0102], page 187) that is equivalent to such a range. A <static discrete_range (see [S0058], page 117)> is one that is a static range or is a subtype_indication (see [S0027], page 56) that defines a static scalar subtype. The base range of a scalar type is a static range, unless the type is a descendant of a formal scalar type.

26/2

A <static subtype> is either a <static scalar subtype> or a <static string subtype>. A static scalar subtype is an unconstrained scalar subtype whose type is not a descendant of a formal type, or a constrained scalar subtype formed by imposing a compatible static constraint on a static scalar subtype. A static string subtype is an unconstrained string subtype whose index subtype and component subtype are static, or a constrained string subtype formed by imposing a compatible static constraint on a static string subtype. In any case, the subtype of a generic formal object of mode in out, and the result subtype of a generic formal function, are not static.

27

The different kinds of <static constraint> are defined as follows:

28

- A null constraint is always static;

29

- A scalar constraint is static if it has no range_constraint, or one with a static range;

30

- An index constraint is static if each discrete_range is static, and each index subtype of the corresponding array type is static;

31

- A discriminant constraint is static if each expression of the constraint is static, and the subtype of each discriminant is static.

31.1/2

In any case, the constraint of the first subtype of a scalar formal type is neither static nor null.

32

A subtype is <statically constrained> if it is constrained, and its constraint is static. An object is <statically constrained> if its nominal subtype is statically constrained, or if it is a static string constant.

Legality Rules

33

A static expression is evaluated at compile time except when it is part of the right operand of a static short-circuit control form whose value is determined by its left operand. This evaluation is performed exactly, without performing `Overflow_Checks`. For a static expression that is evaluated:

34

- The expression is illegal if its evaluation fails a language-defined check other than `Overflow_Check`.

35/2

- If the expression is not part of a larger static expression and the expression is expected to be of a single specific type, then its value shall be within the base range of its expected type. Otherwise, the value may be arbitrarily large or small.

36/2

- If the expression is of type <universal_real> and its expected type is a decimal fixed point type, then its value shall be a multiple of the <small> of the decimal type. This restriction does not apply if the expected type is a descendant of a formal scalar type (or a corresponding actual type in an instance).

37/2

In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), the above restrictions also apply in the private part of an instance of a generic unit.

Implementation Requirements

38/2

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the implementation shall round or truncate the value (according to the `Machine_Rounds` attribute of the expected type) to the nearest machine number of the expected type; if the value is exactly half-way between two machine numbers, the rounding performed is implementation-defined. If the expected type is a descendant of a formal type, or if the static expression appears in the body of an instance of a generic unit and the corresponding expression is nonstatic in the corresponding generic body, then no special rounding or truncating is required -- normal accuracy rules apply (see Chapter 21 [Annex G], page 1083).

Implementation Advice

38.1/2

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the rounding should be the same as the default rounding for the target system.

NOTES

39

27 An expression can be static even if it occurs in a context where staticness is not required.

40

28 A static (or run-time) type_conversion from a real type to an integer type performs rounding. If the operand value is exactly half-way between two integers, the rounding is performed away from zero.

Examples

41

<Examples of static expressions:>

42

```
1 + 1      <-- 2>
abs(-10)*3 <-- 30>
```

43

```
Kilo : constant := 1000;
Mega : constant := Kilo*Kilo;  <-- 1_000_000>
Long : constant := Float'Digits*2;
```

44

```
Half_Pi    : constant := Pi/2;           <-- see Section 4.3.2 [3.3.2], ■
page 65>
Deg_To_Rad : constant := Half_Pi/90;
Rad_To_Deg : constant := 1.0/Deg_To_Rad; <-- equivalent to 1.0/((3.14159_26536/2))
```

5.9.1 4.9.1 Statically Matching Constraints and Subtypes

Static Semantics

1/2

A constraint <statically matches> another constraint if:

1.1/2

- both are null constraints;

1.2/2

- both are static and have equal corresponding bounds or discriminant values;

1.3/2

- both are nonstatic and result from the same elaboration of a constraint of a subtype-indication (see [S0027], page 56) or the same evaluation of a range of a discrete-subtype_definition (see [S0055], page 114); or

1.4/2

- both are nonstatic and come from the same formal_type_declaration.

2/2

A subtype <statically matches> another subtype of the same type if they have statically matching constraints, and, for access subtypes, either both or neither exclude null. Two anonymous access-to-object subtypes statically match if their designated subtypes statically match, and either both or neither exclude null, and either both or neither are access-to-constant. Two anonymous access-to-subprogram subtypes statically match if their designated profiles are subtype conformant, and either both or neither exclude null.

3

Two ranges of the same type <statically match> if both result from the same evaluation of a range, or if both are static and have equal corresponding bounds.

4

A constraint is <statically compatible> with a scalar subtype if it statically matches the constraint of the subtype, or if both are static and the constraint is compatible with the subtype. A constraint is <statically compatible> with an access or composite subtype if it statically matches the constraint of the subtype, or if the subtype is unconstrained. One subtype is <statically compatible> with a second subtype if the constraint of the first is statically compatible with the second subtype.

6 5 Statements

1

A statement defines an action to be performed upon its execution.

2/2

This section describes the general rules applicable to all statements. Some statements are discussed in later sections: `Procedure_call_statement` (see [S0163], page 267)s and `return` statements are described in Chapter 7 [6], page 255, "Chapter 7 [6], page 255, Subprograms". `Entry_call_statement` (see [S0207], page 352)s, `requeue_statement` (see [S0208], page 356)s, `delay_statement` (see [S0209], page 359)s, `accept_statement` (see [S0201], page 347)s, `select_statement` (see [S0212], page 377)s, and `abort_statement` (see [S0227], page 385)s are described in Chapter 10 [9], page 328, "Chapter 10 [9], page 328, Tasks and Synchronization". `Raise_statement` (see [S0251], page 421)s are described in Chapter 12 [11], page 419, "Chapter 12 [11], page 419, Exceptions", and `code_statement` (see [S0294], page 518)s in Chapter 14 [13], page 481. The remaining forms of statements are presented in this section.

6.1 5.1 Simple and Compound Statements - Sequences of Statements

1

A statement is either simple or compound. A `simple_statement` encloses no other statement. A `compound_statement` can enclose `simple_statements` and other `compound_statements`.

Syntax

2

```
sequence_of_statements ::= statement {statement}
```

3

```
statement ::=  
  {label} simple_statement | {label} compound_statement
```

4/2

```
simple_statement ::= null_statement  
  | assignment_statement | exit_statement  
  | goto_statement | procedure_call_statement  
  | simple_return_statement | entry_call_statement  
  | requeue_statement | delay_statement  
  | abort_statement | raise_statement  
  | code_statement
```

5/2

```
compound_statement ::=  
  if_statement | case_statement  
  | loop_statement | block_statement
```


| extended_return_statement
| accept_statement | select_statement

6

null_statement ::= null;

7

label ::= <<<label>statement_identifier>>

8

statement_identifier ::= direct_name

9

The direct_name of a statement_identifier shall be an identifier (not an operator_symbol).

Name Resolution Rules

10

The direct_name of a statement_identifier shall resolve to denote its corresponding implicit declaration (see below).

Legality Rules

11

Distinct identifiers shall be used for all statement_identifiers that appear in the same body, including inner block_statements but excluding inner program units.

Static Semantics

12

For each statement_identifier, there is an implicit declaration (with the specified identifier) at the end of the declarative_part of the innermost block_statement or body that encloses the statement_identifier. The implicit declarations occur in the same order as the statement_identifiers occur in the source text. If a usage name denotes such an implicit declaration, the entity it denotes is the label, loop_statement, or block_statement with the given statement_identifier.

Dynamic Semantics

13

The execution of a null_statement has no effect.

14/2

A <transfer of control> is the run-time action of an exit_statement, return statement, goto_statement, or requeue_statement, selection of a terminate_alternative, raising of an exception, or an abort, which causes the next action performed to be one other than what would normally be expected from the other rules of the language. As explained in Section 8.6.1 [7.6.1], page 299, a transfer of control can cause the execution of constructs to be completed and then left, which may trigger finalization.

15

The execution of a sequence_of_statements consists of the execution of the individual statements in succession until the sequence_ is completed.

NOTES

16

1 A `statement_identifier` that appears immediately within the declarative region of a named `loop_statement` or an `accept_statement` is nevertheless implicitly declared immediately within the declarative region of the innermost enclosing body or `block_statement`; in other words, the expanded name for a named statement is not affected by whether the statement occurs inside or outside a named loop or an `accept_statement` — only nesting within `block_statements` is relevant to the form of its expanded name.

Examples

17

<Examples of labeled statements:>

18

```
<<Here>> <<Ici>> <<Aqui>> <<Hier>> null;
```

19

```
<<After>> X := 1;
```

6.2 5.2 Assignment Statements

1

An `assignment_statement` replaces the current value of a variable with the result of evaluating an expression.

Syntax

2

```
assignment_statement ::=  
  <variable_>name := expression;
```

3

The execution of an `assignment_statement` includes the evaluation of the expression and the <assignment> of the value of the expression into the <target>. An assignment operation (as opposed to an `assignment_statement` (see [S0137], page 242)) is performed in other contexts as well, including object initialization and by-copy parameter passing. The <target> of an assignment operation is the view of the object to which a value is being assigned; the target of an `assignment_statement` (see [S0137], page 242) is the variable denoted by the <variable_>name.

Name Resolution Rules

4/2

The <variable_>name of an `assignment_statement` is expected to be of any type. The expected type for the expression is the type of the target.

Legality Rules

5/2

The target denoted by the <variable_>name shall be a variable of a nonlimited type.

6

If the target is of a tagged class-wide type <T>'Class, then the expression shall either be dynamically tagged, or of type <T> and tag-indeterminate (see Section 4.9.2 [3.9.2], page 145).

Dynamic Semantics

7

For the execution of an assignment_statement, the <variable_>name and the expression are first evaluated in an arbitrary order.

8

When the type of the target is class-wide:

9

- If the expression is tag-indeterminate (see Section 4.9.2 [3.9.2], page 145), then the controlling tag value for the expression is the tag of the target;

10

- Otherwise (the expression is dynamically tagged), a check is made that the tag of the value of the expression is the same as that of the target; if this check fails, Constraint_Error is raised.

11

The value of the expression is converted to the subtype of the target. The conversion might raise an exception (see Section 5.6 [4.6], page 219).

12

In cases involving controlled types, the target is finalized, and an anonymous object might be used as an intermediate in the assignment, as described in Section 8.6.1 [7.6.1], page 299, "Section 8.6.1 [7.6.1], page 299, Completion and Finalization". In any case, the converted value of the expression is then <assigned> to the target, which consists of the following two steps:

13

- The value of the target becomes the converted value.

14

- If any part of the target is controlled, its value is adjusted as explained in clause Section 8.6 [7.6], page 295.

NOTES

15

2 The tag of an object never changes; in particular, an assignment_statement does not change the tag of the target.

16/2

<This paragraph was deleted.>

Examples

17

<Examples of assignment statements:>

18

```
Value := Max_Value - 1;  
Shade := Blue;
```

19

```
Next_Frame(F)(M, N) := 2.5;      --< see Section 5.1.1 [4.1.1],  
page 181>  
U := Dot_Product(V, W);        --< see Section 7.3 [6.3], page 261>■
```

20

```
Writer := (Status => Open, Unit => Printer, Line_Count => 60); --< see Section 4  
[3.8.1], page 134>  
Next_Car.all := (72074, null); --< see Section 4.10.1 [3.10.1],  
page 160>
```

21

<Examples involving scalar subtype conversions:>

22

```
I, J : Integer range 1 .. 10 := 5;  
K    : Integer range 1 .. 20 := 15;  
...
```

23

```
I := J; --< identical ranges>  
K := J; --< compatible ranges>  
J := K; --< will raise Constraint_Error if K > 10>
```

24

<Examples involving array subtype conversions:>

25

```
A : String(1 .. 31);  
B : String(3 .. 33);  
...
```

26

```
A := B; --< same number of components>
```

27

```
A(1 .. 9) := "tar sauce";  
A(4 .. 12) := A(1 .. 9); --< A(1 .. 12) = "tartar sauce">
```

NOTES

28

3 <Notes on the examples:> Assignment_statements are allowed even in the case of overlapping slices of the same array, because the <variable_>name and expression are both evaluated before copying the value into the variable. In the above example, an implementation yielding A(1 .. 12) = "tartartartar" would be incorrect.

6.3 5.3 If Statements

1

An if_statement selects for execution at most one of the enclosed sequences_of_statements, depending on the (truth) value of one or more corresponding conditions.

Syntax

2

```
if_statement ::=  
  if condition then  
    sequence_of_statements  
  {elsif condition then  
    sequence_of_statements}  
  [else  
    sequence_of_statements]  
  end if;
```

3

```
condition ::= <boolean_>expression  
Name Resolution Rules
```

4

A condition is expected to be of any boolean type.

Dynamic Semantics

5

For the execution of an if_statement, the condition specified after if, and any conditions specified after elsif, are evaluated in succession (treating a final else as elsif True then), until one evaluates to True or all conditions are evaluated and yield False. If a condition evaluates to True, then the corresponding sequence_of_statements is executed; otherwise none of them is executed.

Examples

6

<Examples of if statements:>

7

```
if Month = December and Day = 31 then
    Month := January;
    Day   := 1;
    Year  := Year + 1;
end if;
```

8

```
if Line_Too_Short then
    raise Layout_Error;
elsif Line_Full then
    New_Line;
    Put(Item);
else
    Put(Item);
end if;
```

9

```
if My_Car.Owner.Vehicle /= My_Car then           --< see Section 4.10.1
[3.10.1], page 160>
    Report ("Incorrect data");
end if;
```

6.4 5.4 Case Statements

1

A `case_statement` selects for execution one of a number of alternative sequences_of_statements; the chosen alternative is defined by the value of an expression.

Syntax

2

```
case_statement ::=
    case expression is
        case_statement_alternative
        {case_statement_alternative}
    end case;
```

3

```
case_statement_alternative ::=
    when discrete_choice_list =>
        sequence_of_statements
```

Name Resolution Rules

4

The expression is expected to be of any discrete type. The expected type for each `discrete_choice` is the type of the expression.

Legality Rules

5

The expressions and `discrete_ranges` given as `discrete_choices` of a `case_statement` shall be static. A `discrete_choice` others, if present, shall appear alone and in the last `discrete_choice_list`.

6

The possible values of the expression shall be covered as follows:

7

- If the expression is a name (including a `type_conversion` or a `function_call`) having a static and constrained nominal subtype, or is a `qualified_expression` whose `subtype_mark` denotes a static and constrained scalar subtype, then each non-`others` `discrete_choice` shall cover only values in that subtype, and each value of that subtype shall be covered by some `discrete_choice` (either explicitly or by `others`).

8

- If the type of the expression is `<root_integer>`, `<universal_integer>`, or a descendant of a formal scalar type, then the `case_statement` shall have an `others` `discrete_choice`.

9

- Otherwise, each value of the base range of the type of the expression shall be covered (either explicitly or by `others`).

10

Two distinct `discrete_choices` of a `case_statement` shall not cover the same value.

Dynamic Semantics

11

For the execution of a `case_statement` the expression is first evaluated.

12

If the value of the expression is covered by the `discrete_choice_list` (see [S0073], page 134) of some `case_statement_alternative` (see [S0141], page 246), then the `sequence_of_statements` (see [S0130], page 240) of the `_alternative` is executed.

13

Otherwise (the value is not covered by any `discrete_choice_list`, perhaps due to being outside the base range), `Constraint_Error` is raised.

NOTES

14

4 The execution of a `case_statement` chooses one and only one alternative. Qualification of the expression of a `case_statement` by a

static subtype can often be used to limit the number of choices that need be given explicitly.

Examples

15

<Examples of case statements:>

16

```
case Sensor is
  when Elevation => Record_Elevation(Sensor_Value);
  when Azimuth   => Record_Azimuth (Sensor_Value);
  when Distance  => Record_Distance (Sensor_Value);
  when others    => null;
end case;
```

17

```
case Today is
  when Mon   => Compute_Initial_Balance;
  when Fri   => Compute_Closing_Balance;
  when Tue .. Thu => Generate_Report(Today);
  when Sat .. Sun  => null;
end case;
```

18

```
case Bin_Number(Count) is
  when 1   => Update_Bin(1);
  when 2   => Update_Bin(2);
  when 3 | 4 =>
    Empty_Bin(1);
    Empty_Bin(2);
  when others => raise Error;
end case;
```

6.5 5.5 Loop Statements

1

A loop_statement includes a sequence_of_statements that is to be executed repeatedly, zero or more times.

Syntax

2

```
loop_statement ::=
  [<loop_>statement_identifier:]
  [iteration_scheme] loop
    sequence_of_statements
  end loop [<loop_>identifier];
```


3

```
iteration_scheme ::= while condition
                  | for loop_parameter_specification
```

4

```
loop_parameter_specification ::=
    defining_identifier in [reverse] discrete_subtype_definition
```

5

If a loop_statement has a <loop_>statement_identifier, then the identifier shall be repeated after the end loop; otherwise, there shall not be an identifier after the end loop.

Static Semantics

6

A loop_parameter_specification declares a <loop parameter>, which is an object whose subtype is that defined by the discrete_subtype_definition.

Dynamic Semantics

7

For the execution of a loop_statement, the sequence_of_statements is executed repeatedly, zero or more times, until the loop_statement is complete. The loop_statement is complete when a transfer of control occurs that transfers control out of the loop, or, in the case of an iteration_scheme, as specified below.

8

For the execution of a loop_statement with a while iteration_scheme, the condition is evaluated before each execution of the sequence_of_statements (see [S0130], page 240); if the value of the condition is True, the sequence_of_statements (see [S0130], page 240) is executed; if False, the execution of the loop_statement (see [S0142], page 248) is complete.

9

For the execution of a loop_statement with a for iteration_scheme, the loop_parameter_specification (see [S0144], page 249) is first elaborated. This elaboration creates the loop parameter and elaborates the discrete_subtype_definition (see [S0055], page 114). If the discrete_subtype_definition (see [S0055], page 114) defines a subtype with a null range, the execution of the loop_statement is complete. Otherwise, the sequence_of_statements (see [S0130], page 240) is executed once for each value of the discrete subtype defined by the discrete_subtype_definition (see [S0055], page 114) (or until the loop is left as a consequence of a transfer of control). Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter. These values are assigned in increasing order unless the reserved word reverse is present, in which case the values are assigned in decreasing order.

NOTES

10

5 A loop parameter is a constant; it cannot be updated within the sequence_of_statements of the loop (see Section 4.3 [3.3], page 58).

11

6 An `object_declaration` should not be given for a loop parameter, since the loop parameter is automatically declared by the `loop_parameter_specification`. The scope of a loop parameter extends from the `loop_parameter_specification` to the end of the `loop_statement`, and the visibility rules are such that a loop parameter is only visible within the `sequence_of_statements` of the loop.

12

7 The `discrete_subtype_definition` of a for loop is elaborated just once. Use of the reserved word `reverse` does not alter the discrete subtype defined, so that the following `iteration_schemes` are not equivalent; the first has a null range.

13

```
for J in reverse 1 .. 0
for J in 0 .. 1
```

Examples

14

<Example of a loop statement without an iteration scheme:>

15

```
loop
  Get(Current_Character);
  exit when Current_Character = '*';
end loop;
```

16

<Example of a loop statement with a while iteration scheme:>

17

```
while Bid(N).Price < Cut_Off.Price loop
  Record_Bid(Bid(N).Price);
  N := N + 1;
end loop;
```

18

<Example of a loop statement with a for iteration scheme:>

19

```
for J in Buffer'Range loop      --< works even with a null range>
  if Buffer(J) /= Space then
    Put(Buffer(J));
  end if;
```

```
end loop;
```

20

<Example of a loop statement with a name:>

21

```
Summation:
```

```
  while Next /= Head loop      --< see Section 4.10.1 [3.10.1],  
page 160>
```

```
  Sum := Sum + Next.Value;
```

```
  Next := Next.Succ;
```

```
end loop Summation;
```

6.6 5.6 Block Statements

1

A `block_statement` encloses a `handled_sequence_of_statements` optionally preceded by a `declarative_part`.

Syntax

2

```
block_statement ::=
```

```
  [<block_>statement_identifier:]
```

```
  [declare
```

```
    declarative_part]
```

```
  begin
```

```
    handled_sequence_of_statements
```

```
  end [<block_>identifier];
```

3

If a `block_statement` has a `<block_>statement_identifier`, then the identifier shall be repeated after the end; otherwise, there shall not be an identifier after the end.

Static Semantics

4

A `block_statement` that has no explicit `declarative_part` has an implicit empty `declarative_part`.

Dynamic Semantics

5

The execution of a `block_statement` consists of the elaboration of its `declarative_part` followed by the execution of its `handled_sequence_of_statements`.

Examples

6

<Example of a block statement with a local variable:>

7

```

Swap:
  declare
    Temp : Integer;
  begin
    Temp := V; V := U; U := Temp;
  end Swap;

```

6.7 5.7 Exit Statements

1

An `exit_statement` is used to complete the execution of an enclosing `loop_statement`; the completion is conditional if the `exit_statement` includes a condition.

Syntax

2

```

exit_statement ::=
  exit [<loop->name] [when condition];

```

Name Resolution Rules

3

The `<loop->name`, if any, in an `exit_statement` shall resolve to denote a `loop_statement`.

Legality Rules

4

Each `exit_statement` (see [S0146], page 252) <applies to> a `loop_statement` (see [S0142], page 248); this is the `loop_statement` (see [S0142], page 248) being exited. An `exit_statement` (see [S0146], page 252) with a name is only allowed within the `loop_statement` (see [S0142], page 248) denoted by the name, and applies to that `loop_statement` (see [S0142], page 248). An `exit_statement` (see [S0146], page 252) without a name is only allowed within a `loop_statement` (see [S0142], page 248), and applies to the innermost enclosing one. An `exit_statement` (see [S0146], page 252) that applies to a given `loop_statement` (see [S0142], page 248) shall not appear within a body or `accept_statement` (see [S0201], page 347), if this construct is itself enclosed by the given `loop_statement`.

Dynamic Semantics

5

For the execution of an `exit_statement`, the condition, if present, is first evaluated. If the value of the condition is `True`, or if there is no condition, a transfer of control is done to complete the `loop_statement` (see [S0142], page 248). If the value of the condition is `False`, no transfer of control takes place.

NOTES

6

8 Several nested loops can be exited by an `exit_statement` that names the outer loop.

Examples

7

<Examples of loops with exit statements:>

8

```
for N in 1 .. Max_Num_Items loop
  Get_New_Item(New_Item);
  Merge_Item(New_Item, Storage_File);
  exit when New_Item = Terminal_Item;
end loop;
```

9

```
Main_Cycle:
  loop
    --< initial statements>
    exit Main_Cycle when Found;
    --< final statements>
  end loop Main_Cycle;
```

6.8 5.8 Goto Statements

1

A goto_statement specifies an explicit transfer of control from this statement to a target statement with a given label.

Syntax

2

```
goto_statement ::= goto <label_>name;
Name Resolution Rules
```

3

The <label_>name shall resolve to denote a label; the statement with that label is the <target statement>.

Legality Rules

4

The innermost sequence_of_statements that encloses the target statement shall also enclose the goto_statement. Furthermore, if a goto_statement is enclosed by an accept_statement or a body, then the target statement shall not be outside this enclosing construct.

Dynamic Semantics

5

The execution of a goto_statement transfers control to the target statement, completing the execution of any compound_statement that encloses the goto_statement but does not enclose the target.

NOTES

6

9 The above rules allow transfer of control to a statement of an enclosing sequence_of_statements but not the reverse. Similarly, they prohibit transfers of control such as between

alternatives of a case_statement, if_statement, or select_statement;
between exception_handlers; or from an exception_handler of a
handled_sequence_of_statements back to its sequence_of_statements.

Examples

7

<Example of a loop containing a goto statement:>

8

```
<<Sort>>
for I in 1 .. N-1 loop
  if A(I) > A(I+1) then
    Exchange(A(I), A(I+1));
    goto Sort;
  end if;
end loop;
```

7 6 Subprograms

1

A subprogram is a program unit or intrinsic operation whose execution is invoked by a subprogram call. There are two forms of subprogram: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. The definition of a subprogram can be given in two parts: a subprogram declaration defining its interface, and a `subprogram_body` defining its execution. Operators and enumeration literals are functions.

2

A `<callable entity>` is a subprogram or entry (see Section 9). A callable entity is invoked by a `<call>`; that is, a subprogram call or entry call. A `<callable construct>` is a construct that defines the action of a call upon a callable entity: a `subprogram_body`, `entry_body`, or `accept_statement`.

7.1 6.1 Subprogram Declarations

1

A `subprogram_declaration` declares a procedure or function.

Syntax

2/2

```
subprogram_declaration ::=
    [overriding_indicator]
    subprogram_specification;
```

3/2

<This paragraph was deleted.>

4/2

```
subprogram_specification ::=
    procedure_specification
    | function_specification
```

4.1/2

```
procedure_specification ::= procedure defining_program_unit_name parameter_profile
```

4.2/2

```
function_specification ::= function defining_designator parameter_and_result_profile
```

5

```
designator ::= [parent_unit_name . ]identifier | operator_symbol
```

6

```
defining_designator ::= defining_program_unit_name | defining_operator_symbol
```

7

defining_program_unit_name ::= [parent_unit_name .]defining_identifier

8

The optional parent_unit_name is only allowed for library units (see Section 11.1.1 [10.1.1], page 394).

9

operator_symbol ::= string_literal

10/2

The sequence of characters in an operator_symbol shall form a reserved word, a delimiter, or compound delimiter that corresponds to an operator belonging to one of the six categories of operators defined in clause Section 5.5 [4.5], page 203.

11

defining_operator_symbol ::= operator_symbol

12

parameter_profile ::= [formal_part]

13/2

parameter_and_result_profile ::=
 [formal_part] return [null_exclusion] subtype_mark
 | [formal_part] return access_definition

14

formal_part ::=
 (parameter_specification {; parameter_specification})

15/2

parameter_specification ::=
 defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
 | defining_identifier_list : access_definition [:= default_expression]

16

mode ::= [in] | in out | out

Name Resolution Rules

17

A `<formal parameter>` is an object directly visible within a `subprogram.body` that represents the actual parameter passed to the subprogram in a call; it is declared by a `parameter_specification`. For a formal parameter, the expected type for its `default_expression`, if any, is that of the formal parameter.

Legality Rules

18

The `<parameter mode>` of a formal parameter conveys the direction of information transfer with the actual parameter: `in`, `in out`, or `out`. Mode `in` is the default, and is the mode of a parameter defined by an `access_definition`. The formal parameters of a function, if any, shall have the mode `in`.

19

A `default_expression` is only allowed in a `parameter_specification` for a formal parameter of mode `in`.

20/2

A `subprogram_declaration` or a `generic_subprogram_declaration` requires a completion: a body, a `renaming_declaration` (see Section 9.5 [8.5], page 316), or a `pragma Import` (see Section 16.1 [B.1], page 894). A completion is not allowed for an `abstract_subprogram_declaration` (see Section 4.9.3 [3.9.3], page 149) or a `null_procedure_declaration` (see Section 7.7 [6.7], page 277).

21

A name that denotes a formal parameter is not allowed within the `formal_part` in which it is declared, nor within the `formal_part` of a corresponding body or `accept_statement`.

Static Semantics

22

The `<profile>` of (a view of) a callable entity is either a `parameter_profile` or `parameter_and_result_profile`; it embodies information about the interface to that entity — for example, the profile includes information about parameters passed to the callable entity. All callable entities have a profile — enumeration literals, other subprograms, and entries. An `access-to-subprogram` type has a designated profile. Associated with a profile is a calling convention. A `subprogram_declaration` declares a procedure or a function, as indicated by the initial reserved word, with name and profile as given by its specification.

23/2

The nominal subtype of a formal parameter is the subtype determined by the optional `null_exclusion` and the `subtype_mark`, or defined by the `access_definition`, in the `parameter_specification`. The nominal subtype of a function result is the subtype determined by the optional `null_exclusion` and the `subtype_mark`, or defined by the `access_definition`, in the `parameter_and_result_profile`.

24/2

An `<access parameter>` is a formal in parameter specified by an `access_definition`. An `<access result type>` is a function result type specified by an `access_definition`. An access parameter or result type is of an anonymous access type (see Section 4.10 [3.10], page 156). Access parameters of an `access-to-object` type allow dispatching calls to be controlled by access values. Access parameters of an `access-to-subprogram` type permit calls to subprograms passed as parameters irrespective of their accessibility level.

25

The <subtypes of a profile> are:

26

- For any non–access parameters, the nominal subtype of the parameter.

27/2

- For any access parameters of an access–to–object type, the designated subtype of the parameter type.

27.1/2

- For any access parameters of an access–to–subprogram type, the subtypes of the profile of the parameter type.

28/2

- For any non–access result, the nominal subtype of the function result.

28.1/2

- For any access result type of an access–to–object type, the designated subtype of the result type.

28.2/2

- For any access result type of an access–to–subprogram type, the subtypes of the profile of the result type.

29

The <types of a profile> are the types of those subtypes.

30/2

A subprogram declared by an `abstract_subprogram_declaration` is abstract; a subprogram declared by a `subprogram_declaration` is not. See Section 4.9.3 [3.9.3], page 149, "Section 4.9.3 [3.9.3], page 149, Abstract Types and Subprograms". Similarly, a procedure defined by a `null_procedure_declaration` is a null procedure; a procedure declared by a `subprogram_declaration` is not. See Section 7.7 [6.7], page 277, "Section 7.7 [6.7], page 277, Null Procedures".

30.1/2

An `overriding_indicator` is used to indicate whether overriding is intended. See Section 9.3.1 [8.3.1], page 312, "Section 9.3.1 [8.3.1], page 312, Overriding Indicators".

Dynamic Semantics

31/2

The elaboration of a `subprogram_declaration` has no effect.

NOTES

32

1 A parameter_specification with several identifiers is equivalent to a sequence of single parameter_specifications, as explained in Section 4.3 [3.3], page 58.

33

2 Abstract subprograms do not have bodies, and cannot be used in a nondispatching call (see Section 4.9.3 [3.9.3], page 149, "Section 4.9.3 [3.9.3], page 149, Abstract Types and Subprograms").

34

3 The evaluation of default_expressions is caused by certain calls, as described in Section 7.4.1 [6.4.1], page 270. They are not evaluated during the elaboration of the subprogram declaration.

35

4 Subprograms can be called recursively and can be called concurrently from multiple tasks.

Examples

36

<Examples of subprogram declarations:>

37

```
procedure Traverse_Tree;
procedure Increment(X : in out Integer);
procedure Right_Indent(Margin : out Line_Size);           --< see Section 4.5.4 [3.5.4], page 95>
procedure Switch(From, To : in out Link);                --< see Section 4.10.1 [3.10.1], page 160>
```

38

```
function Random return Probability;                       --< see Section 4.5.7 [3.5.7], page 103>
```

39

```
function Min_Cell(X : Link) return Cell;                 --< see Section 4.10.1 [3.10.1], page 160>
function Next_Frame(K : Positive) return Frame;          --< see Section 4.10 [3.10], page 156>
function Dot_Product(Left, Right : Vector) return Real; --< see Section 4.6 [3.6], page 114>
```

40

```
function "*" (Left, Right : Matrix) return Matrix;      --< see Section 4.6 [3.6], page 114>
```

41

<Examples of in parameters with default expressions:>

42

```
procedure Print_Header(Pages  : in Natural;  
                      Header : in Line   := (1 .. Line'Last => ' '); --< see Section 4.  
[3.6], page 114>  
                      Center : in Boolean := True);
```

7.2 6.2 Formal Parameter Modes

1

A parameter_specification declares a formal parameter of mode in, in out, or out.

Static Semantics

2

A parameter is passed either <by copy> or <by reference>. When a parameter is passed by copy, the formal parameter denotes a separate object from the actual parameter, and any information transfer between the two occurs only before and after executing the subprogram. When a parameter is passed by reference, the formal parameter denotes (a view of) the object denoted by the actual parameter; reads and updates of the formal parameter directly reference the actual parameter object.

3

A type is a <by-copy type> if it is an elementary type, or if it is a descendant of a private type whose full type is a by-copy type. A parameter of a by-copy type is passed by copy.

4

A type is a <by-reference type> if it is a descendant of one of the following:

5

- a tagged type;

6

- a task or protected type;

7

- a nonprivate type with the reserved word limited in its declaration;

8

- a composite type with a subcomponent of a by-reference type;

9

- a private type whose full type is a by-reference type.

10

A parameter of a by-reference type is passed by reference. Each value of a by-reference type has an associated object. For a parenthesized expression, `qualified_expression`, or `type_conversion`, this object is the one associated with the operand.

11

For parameters of other types, it is unspecified whether the parameter is passed by copy or by reference.

Bounded (Run-Time) Errors

12

If one name denotes a part of a formal parameter, and a second name denotes a part of a distinct formal parameter or an object that is not part of a formal parameter, then the two names are considered `<distinct access paths>`. If an object is of a type for which the parameter passing mechanism is not specified, then it is a bounded error to assign to the object via one access path, and then read the value of the object via a distinct access path, unless the first access path denotes a part of a formal parameter that no longer exists at the point of the second access (due to leaving the corresponding callable construct). The possible consequences are that `Program_Error` is raised, or the newly assigned value is read, or some old value of the object is read.

NOTES

13

5 A formal parameter of mode `in` is a constant view (see Section 4.3 [3.3], page 58); it cannot be updated within the `subprogram_body`.

7.3 6.3 Subprogram Bodies

1

A `subprogram_body` specifies the execution of a subprogram.

Syntax

2/2

```
subprogram_body ::=
  [overriding_indicator]
  subprogram_specification is
    declarative_part
  begin
    handled_sequence_of_statements
  end [designator];
```

3

If a designator appears at the end of a `subprogram_body`, it shall repeat the `defining_designator` of the `subprogram_specification`.

Legality Rules

4

In contrast to other bodies, a `subprogram_body` need not be the completion of a previous

declaration, in which case the body declares the subprogram. If the body is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a subprogram_body that completes a declaration shall conform fully to that of the declaration.

Static Semantics

5

A subprogram_body is considered a declaration. It can either complete a previous declaration, or itself be the initial declaration of the subprogram.

Dynamic Semantics

6

The elaboration of a non-generic subprogram_body has no other effect than to establish that the subprogram can from then on be called without failing the Elaboration_Check.

7

The execution of a subprogram_body is invoked by a subprogram call. For this execution the declarative_part is elaborated, and the handled_sequence_of_statements is then executed.

Examples

8

<Example of procedure body:>

9

```
procedure Push(E : in Element_Type; S : in out Stack) is
begin
  if S.Index = S.Size then
    raise Stack_Overflow;
  else
    S.Index := S.Index + 1;
    S.Space(S.Index) := E;
  end if;
end Push;
```

10

<Example of a function body:>

11

```
function Dot_Product(Left, Right : Vector) return Real is
  Sum : Real := 0.0;
begin
  Check(Left'First = Right'First and Left'Last = Right'Last);
  for J in Left'Range loop
    Sum := Sum + Left(J)*Right(J);
  end loop;
  return Sum;
end Dot_Product;
```

7.3.1 6.3.1 Conformance Rules

1

When subprogram profiles are given in more than one place, they are required to conform in one of four ways: type conformance, mode conformance, subtype conformance, or full conformance.

Static Semantics

2/1

As explained in Section 16.1 [B.1], page 894, "Section 16.1 [B.1], page 894, Interfacing Pragmas", a <convention> can be specified for an entity. Unless this International Standard states otherwise, the default convention of an entity is Ada. For a callable entity or access-to-subprogram type, the convention is called the <calling convention>. The following conventions are defined by the language:

3

- The default calling convention for any subprogram not listed below is <Ada>. A pragma Convention, Import, or Export may be used to override the default calling convention (see Section 16.1 [B.1], page 894).

4

- The <Intrinsic> calling convention represents subprograms that are "built in" to the compiler. The default calling convention is Intrinsic for the following:

5

- an enumeration literal;

6

- a "/=" operator declared implicitly due to the declaration of "=" (see Section 7.6 [6.6], page 276);

7

- any other implicitly declared subprogram unless it is a dispatching operation of a tagged type;

8

- an inherited subprogram of a generic formal tagged type with unknown discriminants;

9

- an attribute that is a subprogram;

10/2

- a subprogram declared immediately within a `protected_body`;

10.1/2

- any prefixed view of a subprogram (see Section 5.1.3 [4.1.3], page 183).

11

The `Access` attribute is not allowed for Intrinsic subprograms.

12

- The default calling convention is `<protected>` for a protected subprogram, and for an `access-to-subprogram` type with the reserved word `protected` in its definition.

13

- The default calling convention is `<entry>` for an entry.

13.1/2

- The calling convention for an anonymous `access-to-subprogram` parameter or anonymous `access-to-subprogram` result is `<protected>` if the reserved word `protected` appears in its definition and otherwise is the convention of the subprogram that contains the parameter.

13.2/1

- If not specified above as Intrinsic, the calling convention for any inherited or overriding dispatching operation of a tagged type is that of the corresponding subprogram of the parent type. The default calling convention for a new dispatching operation of a tagged type is the convention of the type.

14

Of these four conventions, only `Ada` and `Intrinsic` are allowed as a `<convention_>identifier` in a pragma `Convention`, `Import`, or `Export`.

15/2

Two profiles are `<type conformant>` if they have the same number of parameters, and both have a result if either does, and corresponding parameter and result types are the same, or, for access parameters or access results, corresponding designated types are the same, or corresponding designated profiles are type conformant.

16/2

Two profiles are `<mode conformant>` if they are type-conformant, and corresponding parameters have identical modes, and, for access parameters or access result types, the designated subtypes statically match, or the designated profiles are subtype conformant.

17

Two profiles are <subtype conformant> if they are mode-conformant, corresponding subtypes of the profile statically match, and the associated calling conventions are the same. The profile of a generic formal subprogram is not subtype-conformant with any other profile.

18

Two profiles are <fully conformant> if they are subtype-conformant, and corresponding parameters have the same names and have default_expressions that are fully conformant with one another.

19

Two expressions are <fully conformant> if, after replacing each use of an operator with the equivalent function_call:

20

- each constituent construct of one corresponds to an instance of the same syntactic category in the other, except that an expanded name may correspond to a direct_name (or character_literal) or to a different expanded name in the other; and

21

- each direct_name, character_literal, and selector_name that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding direct_name, character_literal, or selector_name in the other; and

21.1/1

- each attribute_designator in one must be the same as the corresponding attribute_designator in the other; and

22

- each primary that is a literal in one has the same value as the corresponding literal in the other.

23

Two known_discriminant_parts are <fully conformant> if they have the same number of discriminants, and discriminants in the same positions have the same names, statically matching subtypes, and default_expressions that are fully conformant with one another.

24

Two discrete_subtype_definitions are <fully conformant> if they are both subtype_indications or are both ranges, the subtype_marks (if any) denote the same subtype, and the corresponding simple_expressions of the ranges (if any) fully conform.

24.1/2

The <prefixed view profile> of a subprogram is the profile obtained by omitting the first parameter of that subprogram. There is no prefixed view profile for a parameterless subprogram. For the purposes of defining subtype and mode conformance, the convention of a prefixed view profile is considered to match that of either an entry or a protected operation.

25

An implementation may declare an operator declared in a language-defined library unit to be intrinsic.

7.3.2 6.3.2 Inline Expansion of Subprograms

1

Subprograms may be expanded in line at the call site.

Syntax

2

The form of a pragma Inline, which is a program unit pragma (see Section 11.1.5 [10.1.5], page 407), is as follows:

3

```
pragma Inline(name {, name});
```

Legality Rules

4

The pragma shall apply to one or more callable entities or generic subprograms.

Static Semantics

5

If a pragma Inline applies to a callable entity, this indicates that inline expansion is desired for all calls to that entity. If a pragma Inline applies to a generic subprogram, this indicates that inline expansion is desired for all calls to all instances of that generic subprogram.

Implementation Permissions

6

For each call, an implementation is free to follow or to ignore the recommendation expressed by the pragma.

6.1/2

An implementation may allow a pragma Inline that has an argument which is a `direct_name` denoting a `subprogram_body` of the same `declarative_part`.

NOTES

7

6 The name in a pragma Inline can denote more than one entity in the case of overloading. Such a pragma applies to all of the denoted entities.

7.4 6.4 Subprogram Calls

1

A `<subprogram call>` is either a `procedure_call_statement` or a `function_call`; it invokes the execution of the `subprogram_body`. The call specifies the association of the actual parameters, if any, with formal parameters of the subprogram.

Syntax

2

```
procedure_call_statement ::=  
    <procedure_>name;  
    | <procedure_>prefix actual_parameter_part;
```

3

```
function_call ::=  
    <function_>name  
    | <function_>prefix actual_parameter_part
```

4

```
actual_parameter_part ::=  
    (parameter_association {, parameter_association})
```

5

```
parameter_association ::=  
    [<formal_parameter_>selector_name =>] explicit_actual_parameter
```

6

```
explicit_actual_parameter ::= expression | <variable_>name
```

7

A `parameter_association` is `<named>` or `<positional>` according to whether or not the `<formal_parameter_>selector_name` (see [S0099], page 184) is specified. Any positional associations shall precede any named associations. Named associations are not allowed if the prefix in a subprogram call is an `attribute_reference` (see [S0100], page 187).

Name Resolution Rules

8/2

The name or prefix given in a `procedure_call_statement` shall resolve to denote a callable entity that is a procedure, or an entry renamed as (viewed as) a procedure. The name or prefix given in a `function_call` shall resolve to denote a callable entity that is a function. The name or prefix shall not resolve to denote an abstract subprogram unless it is also a dispatching subprogram. When there is an `actual_parameter_part` (see [S0165], page 267), the prefix can be an `implicit_dereference` (see [S0095], page 179) of an `access-to-subprogram` value.

9

A subprogram call shall contain at most one association for each formal parameter. Each formal parameter without an association shall have a `default_expression` (in the profile of the view denoted by the name or prefix). This rule is an overloading rule (see Section 9.6 [8.6], page 324).

Dynamic Semantics

10/2

For the execution of a subprogram call, the name or prefix of the call is evaluated, and each parameter_association (see [S0166], page 267) is evaluated (see Section 7.4.1 [6.4.1], page 270). If a default_expression (see [S0063], page 123) is used, an implicit parameter_association (see [S0166], page 267) is assumed for this rule. These evaluations are done in an arbitrary order. The subprogram_body (see [S0162], page 261) is then executed, or a call on an entry or protected subprogram is performed (see Section 4.9.2 [3.9.2], page 145). Finally, if the subprogram completes normally, then after it is left, any necessary assigning back of formal to actual parameters occurs (see Section 7.4.1 [6.4.1], page 270).

10.1/2

If the name or prefix of a subprogram call denotes a prefixed view (see Section 5.1.3 [4.1.3], page 183), the subprogram call is equivalent to a call on the underlying subprogram, with the first actual parameter being provided by the prefix of the prefixed view (or the Access attribute of this prefix if the first formal parameter is an access parameter), and the remaining actual parameters given by the actual_parameter_part, if any.

11/2

The exception Program_Error is raised at the point of a function_call if the function completes normally without executing a return statement.

12/2

A function_call denotes a constant, as defined in Section 7.5 [6.5], page 272; the nominal subtype of the constant is given by the nominal subtype of the function result.

Examples

13

<Examples of procedure calls:>

14

```
Traverse_Tree;                                --< see Section 7.1
[6.1], page 255>
Print_Header(128, Title, True);                --< see Section 7.1
[6.1], page 255>
```

15

```
Switch(From => X, To => Next);                  --< see Section 7.1
[6.1], page 255>
Print_Header(128, Header => Title, Center => True); --< see Section 7.1
[6.1], page 255>
Print_Header(Header => Title, Center => True, Pages => 128); --< see Section 7.1
[6.1], page 255>
```

16

<Examples of function calls:>

17

```
Dot_Product(U, V)  --< see Section 7.1 [6.1], page 255 and Section 7.3
[6.3], page 261>
```

```
Clock          --< see Section 10.6 [9.6], page 358>
F.all          --< presuming F is of an access-to-subprogram type -- see
[3.10], page 156>
```

18

<Examples of procedures with default expressions:>

19

```
procedure Activate(Process : in Process_Name;
                   After   : in Process_Name := No_Process;
                   Wait    : in Duration := 0.0;
                   Prior   : in Boolean := False);
```

20

```
procedure Pair(Left, Right : in Person_Name := new Person); --< see Section 4.
[3.10.1], page 160>
```

21

<Examples of their calls:>

22

```
Activate(X);
Activate(X, After => Y);
Activate(X, Wait => 60.0, Prior => True);
Activate(X, Y, 10.0, False);
```

23

```
Pair;
Pair(Left => new Person, Right => new Person);
```

NOTES

24

7 If a default_expression is used for two or more parameters in a multiple parameter_specification (see [S0160], page 256), the default_expression (see [S0063], page 123) is evaluated once for each omitted parameter. Hence in the above examples, the two calls of Pair are equivalent.

Examples

25

<Examples of overloaded subprograms:>

26

```
procedure Put(X : in Integer);
procedure Put(X : in String);
```

27

```
procedure Set(Tint    : in Color);  
procedure Set(Signal : in Light);
```

28

<Examples of their calls:>

29

```
Put(28);  
Put("no possible ambiguity here");
```

30

```
Set(Tint    => Red);  
Set(Signal => Red);  
Set(Color'(Red));
```

31

```
--< Set(Red) would be ambiguous since Red may>  
--< denote a value either of type Color or of type Light>
```

7.4.1 6.4.1 Parameter Associations

1

A parameter association defines the association between an actual parameter and a formal parameter.

Name Resolution Rules

2

The <formal_parameter.>selector_name of a parameter_association (see [S0166], page 267) shall resolve to denote a parameter_specification (see [S0160], page 256) of the view being called.

3

The <actual parameter> is either the explicit_actual_parameter given in a parameter_association for a given formal parameter, or the corresponding default_expression if no parameter_association is given for the formal parameter. The expected type for an actual parameter is the type of the corresponding formal parameter.

4

If the mode is in, the actual is interpreted as an expression; otherwise, the actual is interpreted only as a name, if possible.

Legality Rules

5

If the mode is in out or out, the actual shall be a name that denotes a variable.

6

The type of the actual parameter associated with an access parameter shall be convertible (see Section 5.6 [4.6], page 219) to its anonymous access type.

Dynamic Semantics

7

For the evaluation of a `parameter_association`:

8

- The actual parameter is first evaluated.

9

- For an access parameter, the `access_definition` is elaborated, which creates the anonymous access type.

10

- For a parameter (of any mode) that is passed by reference (see Section 7.2 [6.2], page 260), a view conversion of the actual parameter to the nominal subtype of the formal parameter is evaluated, and the formal parameter denotes that conversion.

11

- For an in or in out parameter that is passed by copy (see Section 7.2 [6.2], page 260), the formal parameter object is created, and the value of the actual parameter is converted to the nominal subtype of the formal parameter and assigned to the formal.

12

- For an out parameter that is passed by copy, the formal parameter object is created, and:

13

- For an access type, the formal parameter is initialized from the value of the actual, without a constraint check;

14

- For a composite type with discriminants or that has implicit initial values for any subcomponents (see Section 4.3.1 [3.3.1], page 61), the behavior is as for an in out parameter passed by copy.

15

- For any other type, the formal parameter is uninitialized. If composite, a view conversion of the actual parameter to the nominal subtype of the formal is evaluated (which might raise `Constraint_Error`), and the actual subtype of the formal is that of the view conversion. If elementary, the actual subtype of the formal is given by its nominal subtype.

16

A formal parameter of mode in out or out with discriminants is constrained if either its nominal subtype or the actual parameter is constrained.

17

After normal completion and leaving of a subprogram, for each in out or out parameter that is passed by copy, the value of the formal parameter is converted to the subtype of the variable given as the actual parameter and assigned to it. These conversions and assignments occur in an arbitrary order.

7.5 6.5 Return Statements

1/2

A `simple_return_statement` (see [S0168], page 272) or `extended_return_statement` (see [S0170], page 272) (collectively called a `<return statement>`) is used to complete the execution of the innermost enclosing subprogram_body (see [S0162], page 261), `entry_body` (see [S0203], page 348), or `accept_statement` (see [S0201], page 347).

Syntax

2/2

```
simple_return_statement ::= return [expression];
```

2.1/2

```
extended_return_statement ::=  
    return defining_identifier : [aliased] return_subtype_indication [:= expression] [do  
        handled_sequence_of_statements  
    end return];
```

2.2/2

```
return_subtype_indication ::= subtype_indication | access_definition
```

Name Resolution Rules

3/2

The `<result subtype>` of a function is the subtype denoted by the `subtype_mark`, or defined by the `access_definition`, after the reserved word `return` in the profile of the function. The expected type for the expression, if any, of a `simple_return_statement` (see [S0168], page 272) is the result type of the corresponding function. The expected type for the expression of an `extended_return_statement` is that of the `return_subtype_indication` (see [S0171], page 272).

Legality Rules

4/2

A return statement shall be within a callable construct, and it `<applies to>` the innermost callable construct or `extended_return_statement` that contains it. A return statement shall not be within a body that is within the construct to which the return statement applies.

5/2

A function body shall contain at least one return statement that applies to the function body, unless the function contains `code_statements`. A `simple_return_statement` (see

[S0168], page 272) shall include an expression if and only if it applies to a function body. An `extended_return_statement` shall apply to a function body.

5.1/2

For an `extended_return_statement` (see [S0170], page 272) that applies to a function body:

5.2/2

- If the result subtype of the function is defined by a `subtype_mark`, the `return_subtype_indication` (see [S0171], page 272) shall be a `subtype_indication`. The type of the `subtype_indication` shall be the result type of the function. If the result subtype of the function is constrained, then the subtype defined by the `subtype_indication` shall also be constrained and shall statically match this result subtype. If the result subtype of the function is unconstrained, then the subtype defined by the `subtype_indication` shall be a definite subtype, or there shall be an expression.

5.3/2

- If the result subtype of the function is defined by an `access_definition`, the `return_subtype_indication` (see [S0171], page 272) shall be an `access_definition`. The subtype defined by the `access_definition` shall statically match the result subtype of the function. The accessibility level of this anonymous access subtype is that of the result subtype.

5.4/2

For any return statement that applies to a function body:

5.5/2

- If the result subtype of the function is limited, then the expression of the return statement (if any) shall be an aggregate, a function call (or equivalent use of an operator), or a `qualified_expression` or `parenthesized_expression` whose operand is one of these.

5.6/2

- If the result subtype of the function is class-wide, the accessibility level of the type of the expression of the return statement shall not be statically deeper than that of the master that elaborated the function body. If the result subtype has one or more unconstrained access discriminants, the accessibility level of the anonymous access type of each access discriminant, as determined by the expression of the `simple_return_statement` (see [S0168], page 272) or the `return_subtype_indication` (see [S0171], page 272), shall not be statically deeper than that of the master that elaborated the function body.

Static Semantics

5.7/2

Within an `extended_return_statement`, the `<return object>` is declared with the given `defining_identifier`, with the nominal subtype defined by the `return_subtype_indication` (see [S0171], page 272).

Dynamic Semantics

5.8/2

For the execution of an `extended_return_statement`, the `subtype_indication` or

access_definition is elaborated. This creates the nominal subtype of the return object. If there is an expression, it is evaluated and converted to the nominal subtype (which might raise Constraint_Error — see Section 5.6 [4.6], page 219); the return object is created and the converted value is assigned to the return object. Otherwise, the return object is created and initialized by default as for a stand-alone object of its nominal subtype (see Section 4.3.1 [3.3.1], page 61). If the nominal subtype is indefinite, the return object is constrained by its initial value.

6/2

For the execution of a simple_return_statement (see [S0168], page 272), the expression (if any) is first evaluated, converted to the result subtype, and then is assigned to the anonymous <return object>.

7/2

If the return object has any parts that are tasks, the activation of those tasks does not occur until after the function returns (see Section 10.2 [9.2], page 333).

8/2

If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the value of the expression. A check is made that the accessibility level of the type identified by the tag of the result is not deeper than that of the master that elaborated the function body. If this check fails, Program_Error is raised.

<Paragraphs 9 through 20 were deleted.>

21/2

If the result subtype of a function has one or more unconstrained access discriminants, a check is made that the accessibility level of the anonymous access type of each access discriminant, as determined by the expression or the return_subtype_indication (see [S0171], page 272) of the function, is not deeper than that of the master that elaborated the function body. If this check fails, Program_Error is raised.

22/2

For the execution of an extended_return_statement (see [S0170], page 272), the handled_sequence_of_statements (see [S0247], page 420) is executed. Within this handled_sequence_of_statements (see [S0247], page 420), the execution of a simple_return_statement (see [S0168], page 272) that applies to the extended_return_statement (see [S0170], page 272) causes a transfer of control that completes the extended_return_statement (see [S0170], page 272). Upon completion of a return statement that applies to a callable construct, a transfer of control is performed which completes the execution of the callable construct, and returns to the caller.

23/2

In the case of a function, the function_call denotes a constant view of the return object.

Implementation Permissions

24/2

If the result subtype of a function is unconstrained, and a call on the function is used to provide the initial value of an object with a constrained nominal subtype, Constraint_Error may be raised at the point of the call (after abandoning the execution of the function body) if, while elaborating the return_subtype_indication (see [S0171], page 272) or evaluating the

expression of a return statement that applies to the function body, it is determined that the value of the result will violate the constraint of the subtype of this object.

Examples

25

<Examples of return statements:>

26/2

```
return;                                --< in a procedure body, >entry_body<,>█  
                                        -- accept_statement<, or >extended_return_stateme
```

27

```
return Key_Value(Last_Index);          --< in a function body>
```

28/2

```
return Node : Cell do                  --< in a function body, see Section 4.10.1█  
[3.10.1], page 160 for Cell>  
    Node.Value := Result;  
    Node.Succ := Next_Node;  
end return;
```

7.5.1 6.5.1 Pragma No_Return

1/2

A pragma No_Return indicates that a procedure cannot return normally; it may propagate an exception or loop forever.

Syntax

2/2

The form of a pragma No_Return, which is a representation pragma (see Section 14.1 [13.1], page 481), is as follows:

3/2

```
pragma    No_Return(<procedure_>local_name{,    <proce-  
dure_>local_name});
```

Legality Rules

4/2

Each <procedure_>local_name shall denote one or more procedures or generic procedures; the denoted entities are <non-returning>. The <procedure_>local_name shall not denote a null procedure nor an instance of a generic unit.

5/2

A return statement shall not apply to a non-returning procedure or generic procedure.

6/2

A procedure shall be non-returning if it overrides a dispatching non-returning procedure. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit.

7/2

If a renaming-as-body completes a non-returning procedure declaration, then the re-named procedure shall be non-returning.

Static Semantics

8/2

If a generic procedure is non-returning, then so are its instances. If a procedure declared within a generic unit is non-returning, then so are the corresponding copies of that procedure in instances.

Dynamic Semantics

9/2

If the body of a non-returning procedure completes normally, Program_Error is raised at the point of the call.

Examples

10/2

```
procedure Fail(Msg : String); --< raises Fatal_Error exception>
pragma No_Return(Fail);
--< Inform compiler and reader that procedure never returns normally>■
```

7.6 6.6 Overloading of Operators

1

An <operator> is a function whose designator is an operator_symbol. Operators, like other functions, may be overloaded.

Name Resolution Rules

2

Each use of a unary or binary operator is equivalent to a function_call with <function_>prefix being the corresponding operator_symbol, and with (respectively) one or two positional actual parameters being the operand(s) of the operator (in order).

Legality Rules

3

The subprogram_specification of a unary or binary operator shall have one or two parameters, respectively. A generic function instantiation whose designator is an operator_symbol is only allowed if the specification of the generic function has the corresponding number of parameters.

4

Default_expressions are not allowed for the parameters of an operator (whether the operator is declared with an explicit subprogram_specification or by a generic_instantiation).

5

An explicit declaration of "/=" shall not have a result type of the predefined type Boolean.

Static Semantics

6

A declaration of "=" whose result type is Boolean implicitly declares a declaration of "/=" that gives the complementary result.

NOTES

7

8 The operators "+" and "-" are both unary and binary operators, and hence may be overloaded with both one- and two-parameter functions.

Examples

8

<Examples of user-defined operators:>

9

```
function "+" (Left, Right : Matrix) return Matrix;
function "+" (Left, Right : Vector) return Vector;

--< assuming that A, B, and C are of the type Vector>
--< the following two statements are equivalent:>

A := B + C;
A := "+"(B, C);
```

7.7 6.7 Null Procedures

1/2

A `null_procedure_declaration` provides a shorthand to declare a procedure with an empty body.

Syntax

2/2

```
null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification is null;
```

Static Semantics

3/2

A `null_procedure_declaration` declares a <null procedure>. A completion is not allowed for a `null_procedure_declaration`.

Dynamic Semantics

4/2

The execution of a null procedure is invoked by a subprogram call. For the execution of a subprogram call on a null procedure, the execution of the `subprogram_body` has no effect.

5/2

The elaboration of a `null_procedure_declaration` has no effect.

Examples

6/2

```
procedure Simplify(Expr : in out Expression) is null; --< see Section 4.9
[3.9], page 136>
```

--< By default, Simplify does nothing, but it may be overridden in extensions of

8 7 Packages

1

Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.

8.1 7.1 Package Specifications and Declarations

1

A package is generally provided in two parts: a `package_specification` and a `package_body`. Every package has a `package_specification`, but not all packages have a `package_body`.

Syntax

2

```
package_declaration ::= package_specification;
```

3

```
package_specification ::=  
    package defining_program_unit_name is  
        {basic_declarative_item}  
    [private  
        {basic_declarative_item}]  
    end [[parent_unit_name.]identifier]
```

4

If an identifier or `parent_unit_name.identifier` appears at the end of a `package_specification`, then this sequence of lexical elements shall repeat the `defining_program_unit_name`.

Legality Rules

5/2

A `package_declaration` or `generic_package_declaration` requires a completion (a body) if it contains any `basic_declarative_item` that requires a completion, but whose completion is not in its `package_specification`.

Static Semantics

6/2

The first list of `basic_declarative_items` of a `package_specification` of a package other than a generic formal package is called the <visible part> of the package. The optional list of `basic_declarative_items` after the reserved word `private` (of any `package_specification`) is called the <private part> of the package. If the reserved word `private` does not appear, the package has an implicit empty `private` part. Each list of `basic_declarative_items` of a `package_specification` forms a <declaration list> of the package.

7

An entity declared in the private part of a package is visible only within the declarative region of the package itself (including any child units -- see Section 11.1.1 [10.1.1], page 394). In contrast, expanded names denoting entities declared in the visible part can be used even outside the package; furthermore, direct visibility of such entities can be achieved by means of use_clauses (see Section 5.1.3 [4.1.3], page 183, and Section 9.4 [8.4], page 314).

Dynamic Semantics

8

The elaboration of a package_declaration consists of the elaboration of its basic_declarative_items in the given order.

NOTES

9

1 The visible part of a package contains all the information that another program unit is able to know about the package.

10

2 If a declaration occurs immediately within the specification of a package, and the declaration has a corresponding completion that is a body, then that body has to occur immediately within the body of the package.

Examples

11

<Example of a package declaration:>

12

```
package Rational_Numbers is
```

13

```
    type Rational is
      record
        Numerator   : Integer;
        Denominator : Positive;
      end record;
```

14

```
    function "="(X,Y : Rational) return Boolean;
```

15

```
    function "/" (X,Y : Integer) return Rational; --< to construct a rational
```

16

```
    function "+" (X,Y : Rational) return Rational;
```



```

        function "-" (X,Y : Rational) return Rational;
        function "*" (X,Y : Rational) return Rational;
        function "/" (X,Y : Rational) return Rational;
    end Rational_Numbers;

```

17

There are also many examples of package declarations in the predefined language environment (see Chapter 15 [Annex A], page 553).

8.2 7.2 Package Bodies

1

In contrast to the entities declared in the visible part of a package, the entities declared in the `package_body` are visible only within the `package_body` itself. As a consequence, a package with a `package_body` can be used for the construction of a group of related subprograms in which the logical operations available to clients are clearly isolated from the internal entities.

Syntax

2

```

package_body ::=
    package body defining_program_unit_name is
        declarative_part
    [begin
        handled_sequence_of_statements]
    end [[parent_unit_name.]identifier];

```

3

If an identifier or `parent_unit_name.identifier` appears at the end of a `package_body`, then this sequence of lexical elements shall repeat the `defining_program_unit_name`.

Legality Rules

4

A `package_body` shall be the completion of a previous `package_declaration` (see [S0173], page 279) or `generic_package_declaration` (see [S0254], page 450). A library `package_declaration` (see [S0173], page 279) or library `generic_package_declaration` (see [S0254], page 450) shall not have a body unless it requires a body; pragma `Elaborate_Body` can be used to require a library `unit_declaration` (see [S0231], page 395) to have a body (see Section 11.2.1 [10.2.1], page 413) if it would not otherwise require one.

Static Semantics

5

In any `package_body` without statements there is an implicit `null_statement` (see [S0134], page 241). For any `package_declaration` (see [S0173], page 279) without an explicit completion, there is an implicit `package_body` (see [S0175], page 281) containing a single `null_statement`. For a noninstance, nonlibrary package, this body occurs at the end of the `declarative_part` (see [S0086], page 175) of the innermost enclosing program unit or `block_statement` (see [S0145], page 251); if there are several such packages, the order of

the implicit `package_bodies` is unspecified. (For an instance, the implicit `package_body` (see [S0175], page 281) occurs at the place of the instantiation (see Section 13.3 [12.3], page 454). For a library package, the place is partially determined by the elaboration dependences (see Section 10).)

Dynamic Semantics

6

For the elaboration of a nongeneric `package_body`, its `declarative_part` (see [S0086], page 175) is first elaborated, and its `handled_sequence_of_statements` (see [S0247], page 420) is then executed.

NOTES

7

3 A variable declared in the body of a package is only visible within this body and, consequently, its value can only be changed within the `package_body`. In the absence of local tasks, the value of such a variable remains unchanged between calls issued from outside the package to subprograms declared in the visible part. The properties of such a variable are similar to those of a "static" variable of C.

8

4 The elaboration of the body of a subprogram explicitly declared in the visible part of a package is caused by the elaboration of the body of the package. Hence a call of such a subprogram by an outside program unit raises the exception `Program_Error` if the call takes place before the elaboration of the `package_body` (see Section 4.11 [3.11], page 175).

Examples

9

<Example of a package body (see Section 8.1 [7.1], page 279):>

10

```
package body Rational_Numbers is
```

11

```
    procedure Same_Denominator (X,Y : in out Rational) is
    begin
        --< reduces X and Y to the same denominator:>
        ...
    end Same_Denominator;
```

12

```
function "="(X,Y : Rational) return Boolean is
    U : Rational := X;
    V : Rational := Y;
```

```

begin
  Same_Denominator (U,V);
  return U.Numerator = V.Numerator;
end "=";

```

13

```

function "/" (X,Y : Integer) return Rational is
begin
  if Y > 0 then
    return (Numerator => X, Denominator => Y);
  else
    return (Numerator => -X, Denominator => -Y);
  end if;
end "/";

```

14

```

function "+" (X,Y : Rational) return Rational is ... end "+";
function "-" (X,Y : Rational) return Rational is ... end "-";
function "*" (X,Y : Rational) return Rational is ... end "*";
function "/" (X,Y : Rational) return Rational is ... end "/";

```

15

```

end Rational_Numbers;

```

8.3 7.3 Private Types and Private Extensions

1

The declaration (in the visible part of a package) of a type as a private type or private extension serves to separate the characteristics that can be used directly by outside program units (that is, the logical properties) from other characteristics whose direct use is confined to the package (the details of the definition of the type itself). See Section 4.9.1 [3.9.1], page 143, for an overview of type extensions.

Syntax

2

```

private_type_declaration ::=
  type defining_identifier [discriminant_part] is [[abstract] tagged] [limited] private;

```

3/2

```

private_extension_declaration ::=
  type defining_identifier [discriminant_part] is
    [abstract] [limited | synchronized] new <ancestor->subtype_indication
    [and interface_list] with private;

```

Legality Rules

4

A `private_type_declaration` or `private_extension_declaration` declares a <partial view> of the type; such a declaration is allowed only as a `declarative_item` of the visible part of a package, and it requires a completion, which shall be a `full_type_declaration` that occurs as a `declarative_item` of the private part of the package. The view of the type declared by the `full_type_declaration` is called the <full view>. A generic formal private type or a generic formal private extension is also a partial view.

5

A type shall be completely defined before it is frozen (see Section 4.11.1 [3.11.1], page 177, and Section 14.14 [13.14], page 550). Thus, neither the declaration of a variable of a partial view of a type, nor the creation by an allocator of an object of the partial view are allowed before the full declaration of the type. Similarly, before the full declaration, the name of the partial view cannot be used in a `generic_instantiation` or in a `representation_item`.

6/2

A private type is limited if its declaration includes the reserved word `limited`; a private extension is limited if its ancestor type is a limited type that is not an interface type, or if the reserved word `limited` or `synchronized` appears in its definition. If the partial view is nonlimited, then the full view shall be nonlimited. If a tagged partial view is limited, then the full view shall be limited. On the other hand, if an untagged partial view is limited, the full view may be limited or nonlimited.

7

If the partial view is tagged, then the full view shall be tagged. On the other hand, if the partial view is untagged, then the full view may be tagged or untagged. In the case where the partial view is untagged and the full view is tagged, no derivatives of the partial view are allowed within the immediate scope of the partial view; derivatives of the full view are allowed.

7.1/2

If a full type has a partial view that is tagged, then:

7.2/2

- the partial view shall be a `synchronized tagged type` (see Section 4.9.4 [3.9.4], page 152) if and only if the full type is a `synchronized tagged type`;

7.3/2

- the partial view shall be a descendant of an interface type (see 3.9.4) if and only if the full type is a descendant of the interface type.

8

The <ancestor subtype> of a `private_extension_declaration` is the subtype defined by the <ancestor_>`subtype_indication` (see [S0027], page 56); the ancestor type shall be a specific tagged type. The full view of a private extension shall be derived (directly or indirectly) from the ancestor type. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), the requirement that the ancestor be specific applies also in the private part of an instance of a generic unit.

8.1/2

If the reserved word `limited` appears in a `private_extension_declaration`, the ancestor

type shall be a limited type. If the reserved word `synchronized` appears in a `private_extension_declaration`, the ancestor type shall be a limited interface.

9

If the declaration of a partial view includes a `known_discriminant_part`, then the `full_type_declaration` shall have a fully conforming (explicit) `known_discriminant_part` (see Section 7.3.1 [6.3.1], page 263, "Section 7.3.1 [6.3.1], page 263, Conformance Rules"). The ancestor subtype may be unconstrained; the parent subtype of the full view is required to be constrained (see Section 4.7 [3.7], page 123).

10

If a private extension inherits known discriminants from the ancestor subtype, then the full view shall also inherit its discriminants from the ancestor subtype, and the parent subtype of the full view shall be constrained if and only if the ancestor subtype is constrained.

10.1/2

If the `full_type_declaration` for a private extension is defined by a `derived_type_definition`, then the reserved word `limited` shall appear in the `full_type_declaration` if and only if it also appears in the `private_extension_declaration`.

11

If a partial view has unknown discriminants, then the `full_type_declaration` may define a definite or an indefinite subtype, with or without discriminants.

12

If a partial view has neither known nor unknown discriminants, then the `full_type_declaration` shall define a definite subtype.

13

If the ancestor subtype of a private extension has constrained discriminants, then the parent subtype of the full view shall impose a statically matching constraint on those discriminants.

Static Semantics

14

A `private_type_declaration` declares a private type and its first subtype. Similarly, a `private_extension_declaration` (see [S0177], page 283) declares a private extension and its first subtype.

15

A declaration of a partial view and the corresponding `full_type_declaration` define two views of a single type. The declaration of a partial view together with the visible part define the operations that are available to outside program units; the declaration of the full view together with the private part define other operations whose direct use is possible only within the declarative region of the package itself. Moreover, within the scope of the declaration of the full view, the `<characteristics>` of the type are determined by the full view; in particular, within its scope, the full view determines the classes that include the type, which components, entries, and protected subprograms are visible, what attributes and other predefined operations are allowed, and whether the first subtype is static. See Section 8.3.1 [7.3.1], page 287.

16/2

A private extension inherits components (including discriminants unless there is a new `discriminant_part` specified) and user-defined primitive subprograms from its ancestor type

and its progenitor types (if any), in the same way that a record extension inherits components and user-defined primitive subprograms from its parent type and its progenitor types (see Section 4.4 [3.4], page 66).

Dynamic Semantics

17

The elaboration of a `private_type_declaration` creates a partial view of a type. The elaboration of a `private_extension_declaration` elaborates the `<ancestor_>subtype_indication`, and creates a partial view of a type.

NOTES

18

5 The partial view of a type as declared by a `private_type_declaration` is defined to be a composite view (in Section 4.2 [3.2], page 50). The full view of the type might or might not be composite. A private extension is also composite, as is its full view.

19/2

6 Declaring a private type with an `unknown_discriminant_part` is a way of preventing clients from creating uninitialized objects of the type; they are then forced to initialize each object by calling some operation declared in the visible part of the package.

20/2

7 The ancestor type specified in a `private_extension_declaration` and the parent type specified in the corresponding declaration of a record extension given in the private part need not be the same. If the ancestor type is not an interface type, the parent type of the full view can be any descendant of the ancestor type. In this case, for a primitive subprogram that is inherited from the ancestor type and not overridden, the formal parameter names and default expressions (if any) come from the corresponding primitive subprogram of the specified ancestor type, while the body comes from the corresponding primitive subprogram of the parent type of the full view. See Section 4.9.2 [3.9.2], page 145.

20.1/2

8 If the ancestor type specified in a `private_extension_declaration` is an interface type, the parent type can be any type so long as the full view is a descendant of the ancestor type. The progenitor types specified in a `private_extension_declaration` and the progenitor types specified in the corresponding declaration of a record extension given in the private part need not be the same — the only requirement is that the private extension and the record extension be descended from the same set of interfaces.

Examples

21

<Examples of private type declarations:>

22

```
type Key is private;  
type File_Name is limited private;
```

23

<Example of a private extension declaration:>

24

```
type List is new Ada.Finalization.Controlled with private;
```

8.3.1 7.3.1 Private Operations

1

For a type declared in the visible part of a package or generic package, certain operations on the type do not become visible until later in the package — either in the private part or the body. Such <private operations> are available only inside the declarative region of the package or generic package.

Static Semantics

2

The predefined operators that exist for a given type are determined by the classes to which the type belongs. For example, an integer type has a predefined "+" operator. In most cases, the predefined operators of a type are declared immediately after the definition of the type; the exceptions are explained below. Inherited subprograms are also implicitly declared immediately after the definition of the type, except as stated below.

3/1

For a composite type, the characteristics (see Section 8.3 [7.3], page 283) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later immediately within the declarative region in which the composite type is declared additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place.

4/1

The corresponding rule applies to a type defined by a `derived_type_definition`, if there is a place immediately within the declarative region in which the type is declared where additional characteristics of its parent type become visible.

5/1

For example, an array type whose component type is limited private becomes nonlimited if the full view of the component type is nonlimited and visible at some later place immediately within the declarative region in which the array type is declared. In such a case, the predefined "=" operator is implicitly declared at that place, and assignment is allowed after that place.

6/1

Inherited primitive subprograms follow a different rule. For a `derived_type_definition`, each inherited primitive subprogram is implicitly declared at the earliest place, if any, immediately within the declarative region in which the `type_declaration` occurs, but after the `type_declaration`, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all. An inherited subprogram that is not declared at all cannot be named in a call and cannot be overridden, but for a tagged type, it is possible to dispatch to it.

7

For a `private_extension_declaration`, each inherited subprogram is declared immediately after the `private_extension_declaration` if the corresponding declaration from the ancestor is visible at that place. Otherwise, the inherited subprogram is not declared for the private extension, though it might be for the full type.

8

The `Class` attribute is defined for tagged subtypes in Section 4.9 [3.9], page 136. In addition, for every subtype `S` of an untagged private type whose full view is tagged, the following attribute is defined:

9

`S'Class`

Denotes the class-wide subtype corresponding to the full view of `S`. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. After the full view, the `Class` attribute of the full view can be used.

NOTES

10

9 Because a partial view and a full view are two different views of one and the same type, outside of the defining package the characteristics of the type are those defined by the visible part. Within these outside program units the type is just a private type or private extension, and any language rule that applies only to another class of types does not apply. The fact that the full declaration might implement a private type with a type of a particular class (for example, as

an array type) is relevant only within the declarative region of the package itself including any child units.

11

The consequences of this actual implementation are, however, valid everywhere. For example: any default initialization of components takes place; the attribute `Size` provides the size of the full view; finalization is still done for controlled components of the full view; task dependence rules still apply to components that are task objects.

12/2

10 Partial views provide initialization, membership tests, selected components for the selection of discriminants and inherited components, qualification, and explicit conversion. Nonlimited partial views also allow use of assignment statements.

13

11 For a subtype `S` of a partial view, `S'Size` is defined (see Section 14.3 [13.3], page 486). For an object `A` of a partial view, the attributes `A'Size` and `A'Address` are defined (see Section 14.3 [13.3], page 486). The `Position`, `First_Bit`, and `Last_Bit` attributes are also defined for discriminants and inherited components.

Examples

14

<Example of a type with private operations:>

15

```
package Key_Manager is
  type Key is private;
  Null_Key : constant Key; --< a deferred constant declaration (see Section 8.4
[7.4], page 290)>
  procedure Get_Key(K : out Key);
  function "<" (X, Y : Key) return Boolean;
private
  type Key is new Natural;
  Null_Key : constant Key := Key'First;
end Key_Manager;
```

16

```
package body Key_Manager is
  Last_Key : Key := Null_Key;
  procedure Get_Key(K : out Key) is
  begin
    Last_Key := Last_Key + 1;
```

```
        K := Last_Key;
    end Get_Key;
```

17

```
function "<" (X, Y : Key) return Boolean is
begin
    return Natural(X) < Natural(Y);
end "<";
end Key_Manager;
```

NOTES

18

12 <Notes on the example:> Outside of the package `Key_Manager`, the operations available for objects of type `Key` include assignment, the comparison for equality or inequality, the procedure `Get_Key` and the operator `<`; they do not include other relational operators such as `>=`, or arithmetic operators.

19

The explicitly declared operator `<` hides the predefined operator `<` implicitly declared by the `full_type_declaration`. Within the body of the function, an explicit conversion of `X` and `Y` to the subtype `Natural` is necessary to invoke the `<` operator of the parent type. Alternatively, the result of the function could be written as `not (X >= Y)`, since the operator `>=` is not redefined.

20

The value of the variable `Last_Key`, declared in the package body, remains unchanged between calls of the procedure `Get_Key`. (See also the NOTES of Section 8.2 [7.2], page 281.)

8.4 7.4 Deferred Constants

1

Deferred constant declarations may be used to declare constants in the visible part of a package, but with the value of the constant given in the private part. They may also be used to declare constants imported from other languages (see Chapter 16 [Annex B], page 894).

Legality Rules

2

A <deferred constant declaration> is an `object_declaration` with the reserved word `constant` but no initialization expression. The constant declared by a deferred constant declaration is called a <deferred constant>. A deferred constant declaration requires a completion, which shall be a full constant declaration (called the <full declaration> of the deferred constant), or a pragma `Import` (see Chapter 16 [Annex B], page 894).

3

A deferred constant declaration that is completed by a full constant declaration shall occur immediately within the visible part of a `package_specification`. For this case, the following additional rules apply to the corresponding full declaration:

4

- The full declaration shall occur immediately within the private part of the same package;

5/2

- The deferred and full constants shall have the same type, or shall have statically matching anonymous access subtypes;

6/2

- If the deferred constant declaration includes a `subtype_indication` that defines a constrained subtype, then the subtype defined by the `subtype_indication` in the full declaration shall match it statically. On the other hand, if the subtype of the deferred constant is unconstrained, then the full declaration is still allowed to impose a constraint. The constant itself will be constrained, like all constants;

7/2

- If the deferred constant declaration includes the reserved word `aliased`, then the full declaration shall also;

7.1/2

- If the subtype of the deferred constant declaration excludes `null`, the subtype of the full declaration shall also exclude `null`.

8

A deferred constant declaration that is completed by a `pragma Import` need not appear in the visible part of a `package_specification`, and has no full constant declaration.

9/2

The completion of a deferred constant declaration shall occur before the constant is frozen (see Section 14.14 [13.14], page 550).

Dynamic Semantics

10

The elaboration of a deferred constant declaration elaborates the `subtype_indication` or (only allowed in the case of an imported constant) the `array_type_definition`.

NOTES

11

13 The full constant declaration for a deferred constant that is of a given private type or private extension is not allowed before the

corresponding `full_type_declaration`. This is a consequence of the freezing rules for types (see Section 14.14 [13.14], page 550).

Examples

12

<Examples of deferred constant declarations:>

13

```
Null_Key : constant Key;      --< see Section 8.3.1 [7.3.1], page 287>■
```

14

```
CPU_Identifier : constant String(1..8);  
pragma Import(Assembler, CPU_Identifier, Link_Name => "CPU_ID");  
--< see Section 16.1 [B.1], page 894>
```

8.5 7.5 Limited Types

1/2

A limited type is (a view of) a type for which copying (such as for an `assignment_statement`) is not allowed. A nonlimited type is a (view of a) type for which copying is allowed.

Legality Rules

2/2

If a tagged record type has any limited components, then the reserved word `limited` shall appear in its `record_type_definition`. If the reserved word `limited` appears in the definition of a `derived_type_definition`, its parent type and any progenitor interfaces shall be limited.

2.1/2

In the following contexts, an expression of a limited type is not permitted unless it is an aggregate, a `function_call`, or a parenthesized expression or `qualified_expression` whose operand is permitted by this rule:

2.2/2

- the initialization expression of an `object_declaration` (see Section 4.3.1 [3.3.1], page 61)

2.3/2

- the `default_expression` of a `component_declaration` (see Section 4.8 [3.8], page 130)

2.4/2

- the expression of a `record_component_association` (see Section 5.3.1 [4.3.1], page 191)

2.5/2

- the expression for an `ancestor_part` of an `extension_aggregate` (see Section 5.3.2 [4.3.2], page 194)

2.6/2

- an expression of a `positional_array_aggregate` or the expression of an `array_component_association` (see Section 5.3.3 [4.3.3], page 196)

2.7/2

- the `qualified_expression` of an initialized allocator (see Section 5.8 [4.8], page 230)

2.8/2

- the expression of a return statement (see Section 7.5 [6.5], page 272)

2.9/2

- the `default_expression` or actual parameter for a formal object of mode `in` (see Section 13.4 [12.4], page 458)

Static Semantics

3/2

A type is `<limited>` if it is one of the following:

4/2

- a type with the reserved word `limited`, `synchronized`, `task`, or `protected` in its definition;

5/2

- `<This paragraph was deleted.>`

6/2

- a composite type with a limited component;

6.1/2

- a derived type whose parent is limited and is not an interface.

7

Otherwise, the type is nonlimited.

8

There are no predefined equality operators for a limited type.

Implementation Requirements

8.1/2

For an aggregate of a limited type used to initialize an object as allowed above, the implementation shall not create a separate anonymous object for the aggregate. For a `function_call` of a type with a part that is of a `task`, `protected`, or explicitly limited record type that is used to initialize an object as allowed above, the implementation shall not create a separate return object (see 6.5) for the `function_call`. The aggregate or `function_call` shall be constructed directly in the new object.

NOTES

9/2

14 While it is allowed to write initializations of limited objects, such initializations never copy a limited object. The source of such an assignment operation must be an aggregate or function_call, and such aggregates and function_calls must be built directly in the target object.

<Paragraphs 10 through 15 were deleted.>

16

15 As illustrated in Section 8.3.1 [7.3.1], page 287, an untagged limited type can become nonlimited under certain circumstances.

Examples

17

<Example of a package with a limited type:>

18

```
package IO_Package is
  type File_Name is limited private;

  procedure Open (F : in out File_Name);
  procedure Close(F : in out File_Name);
  procedure Read (F : in File_Name; Item : out Integer);
  procedure Write(F : in File_Name; Item : in Integer);
private
  type File_Name is
    limited record
      Internal_Name : Integer := 0;
    end record;
end IO_Package;
```

19

20

```
package body IO_Package is
  Limit : constant := 200;
  type File_Descriptor is record ... end record;
  Directory : array (1 .. Limit) of File_Descriptor;
  ...
  procedure Open (F : in out File_Name) is ... end;
  procedure Close(F : in out File_Name) is ... end;
  procedure Read (F : in File_Name; Item : out Integer) is ... end;
  procedure Write(F : in File_Name; Item : in Integer) is ... end;
begin
```

```
...
end IO_Package;
```

NOTES

21

16 <Notes on the example:> In the example above, an outside subprogram making use of `IO_Package` may obtain a file name by calling `Open` and later use it in calls to `Read` and `Write`. Thus, outside the package, a file name obtained from `Open` acts as a kind of password; its internal properties (such as containing a numeric value) are not known and no other operations (such as addition or comparison of internal names) can be performed on a file name. Most importantly, clients of the package cannot make copies of objects of type `File_Name`.

22

This example is characteristic of any case where complete control over the operations of a type is desired. Such packages serve a dual purpose. They prevent a user from making use of the internal structure of the type. They also implement the notion of an encapsulated data type where the only operations on the type are those given in the package specification.

23/2

The fact that the full view of `File_Name` is explicitly declared limited means that parameter passing will always be by reference and function results will always be built directly in the result object (see Section 7.2 [6.2], page 260, and Section 7.5 [6.5], page 272).

8.6 7.6 User-Defined Assignment and Finalization

1

Three kinds of actions are fundamental to the manipulation of objects: initialization, finalization, and assignment. Every object is initialized, either explicitly or by default, after being created (for example, by an `object_declaration` or `allocator`). Every object is finalized before being destroyed (for example, by leaving a `subprogram_body` containing an `object_declaration`, or by a call to an instance of `Unchecked_Deallocation`). An assignment operation is used as part of `assignment_statements`, explicit initialization, parameter passing, and other operations.

2

Default definitions for these three fundamental operations are provided by the language, but a <controlled> type gives the user additional control over parts of these operations. In particular, the user can define, for a controlled type, an `Initialize` procedure which is invoked immediately after the normal default initialization of a controlled object, a `Finalize` procedure which is invoked immediately before finalization of any of the components of a

controlled object, and an Adjust procedure which is invoked as the last step of an assignment to a (nonlimited) controlled object.

Static Semantics

3

The following language-defined library package exists:

4/1

```
package Ada.Finalization is
```

```
    pragma Preelaborate(Finalization);  
    pragma Remote_Types(Finalization);
```

5/2

```
    type  
    Controlled is abstract tagged private;  
    pragma Preelaborable_Initialization(Controlled);
```

6/2

```
    procedure  
    Initialize (Object : in out Controlled) is null;  
    procedure  
    Adjust    (Object : in out Controlled) is null;  
    procedure  
    Finalize  (Object : in out Controlled) is null;
```

7/2

```
    type  
    Limited_Controlled is abstract tagged limited private;  
    pragma Preelaborable_Initialization(Limited_Controlled);
```

8/2

```
    procedure  
    Initialize (Object : in out Limited_Controlled) is null;  
    procedure  
    Finalize  (Object : in out Limited_Controlled) is null;  
private  
    ... -- <not specified by the language>  
end Ada.Finalization;
```

9/2

A controlled type is a descendant of Controlled or Limited_Controlled. The predefined "=" operator of type Controlled always returns True, since this operator is incorporated into the implementation of the predefined equality operator of types derived from Controlled, as

explained in Section 5.5.2 [4.5.2], page 206. The type `Limited_Controlled` is like `Controlled`, except that it is limited and it lacks the primitive subprogram `Adjust`.

9.1/2

A type is said to `<need finalization>` if:

9.2/2

- it is a controlled type, a task type or a protected type; or

9.3/2

- it has a component that needs finalization; or

9.4/2

- it is a limited type that has an access discriminant whose designated type needs finalization; or

9.5/2

- it is one of a number of language-defined types that are explicitly defined to need finalization.

Dynamic Semantics

10/2

During the elaboration or evaluation of a construct that causes an object to be initialized by default, for every controlled subcomponent of the object that is not assigned an initial value (as defined in Section 4.3.1 [3.3.1], page 61), `Initialize` is called on that subcomponent. Similarly, if the object that is initialized by default as a whole is controlled, `Initialize` is called on the object.

11/2

For an `extension_aggregate` whose `ancestor_part` is a `subtype_mark` denoting a controlled subtype, the `Initialize` procedure of the ancestor type is called, unless that `Initialize` procedure is abstract.

12

`Initialize` and other initialization operations are done in an arbitrary order, except as follows. `Initialize` is applied to an object after initialization of its subcomponents, if any (including both implicit initialization and `Initialize` calls). If an object has a component with an access discriminant constrained by a per-object expression, `Initialize` is applied to this component after any components that do not have such discriminants. For an object with several components with such a discriminant, `Initialize` is applied to them in order of their `component_declarations`. For an allocator, any task activations follow all calls on `Initialize`.

13

When a target object with any controlled parts is assigned a value, either when created or in a subsequent `assignment_statement`, the `<assignment operation>` proceeds as follows:

14

- The value of the target becomes the assigned value.

15

- The value of the target is <adjusted.>

16

To adjust the value of a (nonlimited) composite object, the values of the components of the object are first adjusted in an arbitrary order, and then, if the object is controlled, Adjust is called. Adjusting the value of an elementary object has no effect, nor does adjusting the value of a composite object with no controlled parts.

17

For an assignment_statement, after the name and expression have been evaluated, and any conversion (including constraint checking) has been done, an anonymous object is created, and the value is assigned into it; that is, the assignment operation is applied. (Assignment includes value adjustment.) The target of the assignment_statement is then finalized. The value of the anonymous object is then assigned into the target of the assignment_statement. Finally, the anonymous object is finalized. As explained below, the implementation may eliminate the intermediate anonymous object, so this description subsumes the one given in Section 6.2 [5.2], page 242, "Section 6.2 [5.2], page 242, Assignment Statements".

Implementation Requirements

17.1/2

For an aggregate of a controlled type whose value is assigned, other than by an assignment_statement, the implementation shall not create a separate anonymous object for the aggregate. The aggregate value shall be constructed directly in the target of the assignment operation and Adjust is not called on the target object.

Implementation Permissions

18

An implementation is allowed to relax the above rules (for nonlimited controlled types) in the following ways:

19

- For an assignment_statement that assigns to an object the value of that same object, the implementation need not do anything.

20

- For an assignment_statement for a noncontrolled type, the implementation may finalize and assign each component of the variable separately (rather than finalizing the entire variable and assigning the entire new value) unless a discriminant of the variable is changed by the assignment.

21/2

- For an aggregate or function call whose value is assigned into a target object, the implementation need not create a separate anonymous object if it can safely create the value of the aggregate or function call directly in the target object. Similarly, for an assignment_statement (see [S0137], page 242), the implementation need not create an anonymous object if the value being assigned is the result of evaluating a name

denoting an object (the source object) whose storage cannot overlap with the target. If the source object might overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object).

22/2

Furthermore, an implementation is permitted to omit implicit Initialize, Adjust, and Finalize calls and associated assignment operations on an object of a nonlimited controlled type provided that:

23/2

- any omitted Initialize call is not a call on a user-defined Initialize procedure, and

24/2

- any usage of the value of the object after the implicit Initialize or Adjust call and before any subsequent Finalize call on the object does not change the external effect of the program, and

25/2

- after the omission of such calls and operations, any execution of the program that executes an Initialize or Adjust call on an object or initializes an object by an aggregate will also later execute a Finalize call on the object and will always do so prior to assigning a new value to the object, and

26/2

- the assignment operations associated with omitted Adjust calls are also omitted.

27/2

This permission applies to Adjust and Finalize calls even if the implicit calls have additional external effects.

8.6.1 7.6.1 Completion and Finalization

1

This subclause defines <completion> and <leaving> of the execution of constructs and entities. A <master> is the execution of a construct that includes finalization of local objects after it is complete (and after waiting for any local tasks — see Section 10.3 [9.3], page 335), but before leaving. Other constructs and entities are left immediately upon completion.

Dynamic Semantics

2/2

The execution of a construct or entity is <complete> when the end of that execution has been reached, or when a transfer of control (see Section 6.1 [5.1], page 240) causes it to be abandoned. Completion due to reaching the end of execution, or due to the transfer of control of an `exit_statement`, `return_statement`, `goto_statement`, or `requeue_statement` or of

the selection of a `terminate_alternative` is `<normal completion>`. Completion is `<abnormal>` otherwise — when control is transferred out of a construct due to abort or the raising of an exception.

3/2

After execution of a construct or entity is complete, it is `<left>`, meaning that execution continues with the next action, as defined for the execution that is taking place. Leaving an execution happens immediately after its completion, except in the case of a `<master>`: the execution of a body other than a `package_body`; the execution of a statement; or the evaluation of an expression, `function_call`, or range that is not part of an enclosing expression, `function_call`, range, or `simple_statement` (see [S0132], page 240) other than a `simple_return_statement` (see [S0168], page 272). A master is finalized after it is complete, and before it is left.

4

For the `<finalization>` of a master, dependent tasks are first awaited, as explained in Section 10.3 [9.3], page 335. Then each object whose accessibility level is the same as that of the master is finalized if the object was successfully initialized and still exists. These actions are performed whether the master is left by reaching the last statement or via a transfer of control. When a transfer of control causes completion of an execution, each included master is finalized in order, from innermost outward.

5

For the `<finalization>` of an object:

6

- If the object is of an elementary type, finalization has no effect;

7

- If the object is of a controlled type, the `Finalize` procedure is called;

8

- If the object is of a protected type, the actions defined in Section 10.4 [9.4], page 337, are performed;

9/2

- If the object is of a composite type, then after performing the above actions, if any, every component of the object is finalized in an arbitrary order, except as follows: if the object has a component with an access discriminant constrained by a per-object expression, this component is finalized before any components that do not have such discriminants; for an object with several components with such a discriminant, they are finalized in the reverse of the order of their `component_declarations`;

9.1/2

- If the object has coextensions (see Section 4.10.2 [3.10.2], page 164), each coextension is finalized after the object whose access discriminant designates it.

10

Immediately before an instance of `Unchecked_Deallocation` reclaims the storage of an object, the object is finalized. If an instance of `Unchecked_Deallocation` is never applied to an object created by an allocator, the object will still exist when the corresponding master completes, and it will be finalized then.

11/2

The order in which the finalization of a master performs finalization of objects is as follows: Objects created by declarations in the master are finalized in the reverse order of their creation. For objects that were created by allocators for an access type whose ultimate ancestor is declared in the master, this rule is applied as though each such object that still exists had been created in an arbitrary order at the first freezing point (see Section 14.14 [13.14], page 550) of the ultimate ancestor type; the finalization of these objects is called the `<finalization of the collection>`. After the finalization of a master is complete, the objects finalized as part of its finalization cease to `<exist>`, as do any types and subtypes defined and created within the master.

12/2

The target of an `assignment_statement` is finalized before copying in the new value, as explained in Section 8.6 [7.6], page 295.

13/2

The master of an object is the master enclosing its creation whose accessibility level (see Section 4.10.2 [3.10.2], page 164) is equal to that of the object.

13.1/2

In the case of an expression that is a master, finalization of any (anonymous) objects occurs as the final part of evaluation of the expression.

Bounded (Run-Time) Errors

14/1

It is a bounded error for a call on `Finalize` or `Adjust` that occurs as part of object finalization or assignment to propagate an exception. The possible consequences depend on what action invoked the `Finalize` or `Adjust` operation:

15

- For a `Finalize` invoked as part of an `assignment_statement`, `Program_Error` is raised at that point.

16/2

- For an `Adjust` invoked as part of assignment operations other than those invoked as part of an `assignment_statement`, other adjustments due to be performed might or might not be performed, and then `Program_Error` is raised. During its propagation, finalization might or might not be applied to objects whose `Adjust` failed. For an `Adjust` invoked as part of an `assignment_statement`, any other adjustments due to be performed are performed, and then `Program_Error` is raised.

17

- For a `Finalize` invoked as part of a call on an instance of `Unchecked_Deallocation`, any other finalizations due to be performed are performed, and then `Program_Error` is raised.

17.1/1

- For a `Finalize` invoked as part of the finalization of the anonymous object created by a function call or aggregate, any other finalizations due to be performed are performed, and then `Program_Error` is raised.

17.2/1

- For a `Finalize` invoked due to reaching the end of the execution of a master, any other finalizations associated with the master are performed, and `Program_Error` is raised immediately after leaving the master.

18/2

- For a `Finalize` invoked by the transfer of control of an `exit_statement`, `return_statement`, `goto_statement`, or `requeue_statement` (see [S0208], page 356), `Program_Error` is raised no earlier than after the finalization of the master being finalized when the exception occurred, and no later than the point where normal execution would have continued. Any other finalizations due to be performed up to that point are performed before raising `Program_Error`.

19

- For a `Finalize` invoked by a transfer of control that is due to raising an exception, any other finalizations due to be performed for the same master are performed; `Program_Error` is raised immediately after leaving the master.

20

- For a `Finalize` invoked by a transfer of control due to an abort or selection of a terminate alternative, the exception is ignored; any other finalizations due to be performed are performed.

NOTES

21

17 The rules of Section 10 imply that immediately prior to partition termination, `Finalize` operations are applied to library-level controlled objects (including those created by allocators of library-level access types, except those already finalized). This occurs after waiting for library-level tasks to terminate.

22

18 A constant is only constant between its initialization and finalization. Both initialization and finalization are allowed to change the value of a constant.

23

19 Abort is deferred during certain operations related to controlled types, as explained in Section 10.8 [9.8], page 385. Those rules prevent an abort from causing a controlled object to be left in an ill-defined state.

24

20 The Finalize procedure is called upon finalization of a controlled object, even if Finalize was called earlier, either explicitly or as part of an assignment; hence, if a controlled type is visibly controlled (implying that its Finalize primitive is directly callable), or is non-limited (implying that assignment is allowed), its Finalize procedure should be designed to have no ill effect if it is applied a second time to the same object.

9 8 Visibility Rules

1

The rules defining the scope of declarations and the rules defining which identifiers, character_literals, and operator_symbols are visible at (or from) various places in the text of the program are described in this section. The formulation of these rules uses the notion of a declarative region.

2

As explained in Section 3, a declaration declares a view of an entity and associates a defining name with that view. The view comprises an identification of the viewed entity, and possibly additional properties. A usage name denotes a declaration. It also denotes the view declared by that declaration, and denotes the entity of that view. Thus, two different usage names might denote two different views of the same entity; in this case they denote the same entity.

9.1 8.1 Declarative Region

Static Semantics

1

For each of the following constructs, there is a portion of the program text called its <declarative region>, within which nested declarations can occur:

2

- any declaration, other than that of an enumeration type, that is not a completion of a previous declaration;

3

- a block_statement;

4

- a loop_statement;

4.1/2

- an extended_return_statement;

5

- an accept_statement;

6

- an exception_handler.

7

The declarative region includes the text of the construct together with additional text determined (recursively), as follows:

8

- If a declaration is included, so is its completion, if any.

9

- If the declaration of a library unit (including Standard — see Section 11.1.1 [10.1.1], page 394) is included, so are the declarations of any child units (and their completions, by the previous rule). The child declarations occur after the declaration.

10

- If a `body_stub` is included, so is the corresponding subunit.

11

- If a `type_declaration` is included, then so is a corresponding `record_representation_clause`, if any.

12

The declarative region of a declaration is also called the `<declarative region>` of any view or entity declared by the declaration.

13

A declaration occurs `<immediately within>` a declarative region if this region is the innermost declarative region that encloses the declaration (the `<immediately enclosing>` declarative region), not counting the declarative region (if any) associated with the declaration itself.

14

A declaration is `<local>` to a declarative region if the declaration occurs immediately within the declarative region. An entity is `<local>` to a declarative region if the entity is declared by a declaration that is local to the declarative region.

15

A declaration is `<global>` to a declarative region if the declaration occurs immediately within another declarative region that encloses the declarative region. An entity is `<global>` to a declarative region if the entity is declared by a declaration that is global to the declarative region.

NOTES

16

1 The children of a parent library unit are inside the parent's declarative region, even though they do not occur inside the parent's declaration or body. This implies that one can use (for example) "P.Q" to refer to a child of P whose defining name is Q, and that after "use P;" Q can refer (directly) to that child.

17

2 As explained above and in Section 11.1.1 [10.1.1], page 394, "Section 11.1.1 [10.1.1], page 394, Compilation Units - Library

Units", all library units are descendants of Standard, and so are contained in the declarative region of Standard. They are <not> inside the declaration or body of Standard, but they <are> inside its declarative region.

18

3 For a declarative region that comes in multiple parts, the text of the declarative region does not contain any text that might appear between the parts. Thus, when a portion of a declarative region is said to extend from one place to another in the declarative region, the portion does not contain any text that might appear between the parts of the declarative region.

9.2 8.2 Scope of Declarations

1

For each declaration, the language rules define a certain portion of the program text called the <scope> of the declaration. The scope of a declaration is also called the scope of any view or entity declared by the declaration. Within the scope of an entity, and only there, there are places where it is legal to refer to the declared entity. These places are defined by the rules of visibility and overloading.

Static Semantics

2

The <immediate scope> of a declaration is a portion of the declarative region immediately enclosing the declaration. The immediate scope starts at the beginning of the declaration, except in the case of an overloadable declaration, in which case the immediate scope starts just after the place where the profile of the callable entity is determined (which is at the end of the `_specification` for the callable entity, or at the end of the `generic_instantiation` if an instance). The immediate scope extends to the end of the declarative region, with the following exceptions:

3

- The immediate scope of a `library_item` includes only its semantic dependents.

4

- The immediate scope of a declaration in the private part of a library unit does not include the visible part of any public descendant of that library unit.

5

The <visible part> of (a view of) an entity is a portion of the text of its declaration containing declarations that are visible from outside. The <private part> of (a view of) an entity that has a visible part contains all declarations within the declaration of (the view of) the entity, except those in the visible part; these are not visible from outside. Visible and private parts are defined only for these kinds of entities: callable entities, other program units, and composite types.

6

- The visible part of a view of a callable entity is its profile.

7

- The visible part of a composite type other than a task or protected type consists of the declarations of all components declared (explicitly or implicitly) within the `type_declaration`.

8

- The visible part of a generic unit includes the `generic_formal_part`. For a generic package, it also includes the first list of `basic_declarative_items` of the `package_specification`. For a generic subprogram, it also includes the profile.

9

- The visible part of a package, task unit, or protected unit consists of declarations in the program unit's declaration other than those following the reserved word `private`, if any; see Section 8.1 [7.1], page 279, and Section 13.7 [12.7], page 474, for packages, Section 10.1 [9.1], page 329, for task units, and Section 10.4 [9.4], page 337, for protected units.

10

The scope of a declaration always contains the immediate scope of the declaration. In addition, for a given declaration that occurs immediately within the visible part of an outer declaration, or is a public child of an outer declaration, the scope of the given declaration extends to the end of the scope of the outer declaration, except that the scope of a `library_item` includes only its semantic dependents.

10.1/2

The scope of an `attribute_definition_clause` is identical to the scope of a declaration that would occur at the point of the `attribute_definition_clause`.

11

The immediate scope of a declaration is also the immediate scope of the entity or view declared by the declaration. Similarly, the scope of a declaration is also the scope of the entity or view declared by the declaration.

NOTES

12

4 There are notations for denoting visible declarations that are not directly visible. For example, `parameter_specification` (see [S0160], page 256)s are in the visible part of a `subprogram_declaration` (see [S0148], page 255) so that they can be used in named–notation calls appearing outside the called subprogram. For another example, declarations of the visible part of a package can be denoted by expanded names appearing outside the package, and can be made directly visible by a `use_clause`.

9.3 8.3 Visibility

1

The <visibility rules>, given below, determine which declarations are visible and directly visible at each place within a program. The visibility rules apply to both explicit and implicit declarations.

Static Semantics

2

A declaration is defined to be <directly visible> at places where a name consisting of only an identifier or operator_symbol is sufficient to denote the declaration; that is, no selected_component notation or special context (such as preceding => in a named association) is necessary to denote the declaration. A declaration is defined to be <visible> wherever it is directly visible, as well as at other places where some name (such as a selected_component) can denote the declaration.

3

The syntactic category direct_name is used to indicate contexts where direct visibility is required. The syntactic category selector_name is used to indicate contexts where visibility, but not direct visibility, is required.

4

There are two kinds of direct visibility: <immediate visibility> and <use-visibility>. A declaration is immediately visible at a place if it is directly visible because the place is within its immediate scope. A declaration is use-visible if it is directly visible because of a use_clause (see Section 9.4 [8.4], page 314). Both conditions can apply.

5

A declaration can be <hidden>, either from direct visibility, or from all visibility, within certain parts of its scope. Where <hidden from all visibility>, it is not visible at all (neither using a direct_name nor a selector_name). Where <hidden from direct visibility>, only direct visibility is lost; visibility using a selector_name is still possible.

6

Two or more declarations are <overloaded> if they all have the same defining name and there is a place where they are all directly visible.

7

The declarations of callable entities (including enumeration literals) are <overloadable>, meaning that overloading is allowed for them.

8

Two declarations are <homographs> if they have the same defining name, and, if both are overloadable, their profiles are type conformant. An inner declaration hides any outer homograph from direct visibility.

9/1

Two homographs are not generally allowed immediately within the same declarative region unless one <overrides> the other (see Legality Rules below). The only declarations that are <overridable> are the implicit declarations for predefined operators and inherited primitive subprograms. A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases:

10/1

- A declaration that is not overridable overrides one that is overridable, regardless of which declaration occurs first;

11

- The implicit declaration of an inherited operator overrides that of a predefined operator;

12

- An implicit declaration of an inherited subprogram overrides a previous implicit declaration of an inherited subprogram.

12.1/2

- If two or more homographs are implicitly declared at the same place:

12.2/2

- If at least one is a subprogram that is neither a null procedure nor an abstract subprogram, and does not require overriding (see Section 4.9.3 [3.9.3], page 149), then they override those that are null procedures, abstract subprograms, or require overriding. If more than one such homograph remains that is not thus overridden, then they are all hidden from all visibility.

12.3/2

- Otherwise (all are null procedures, abstract subprograms, or require overriding), then any null procedure overrides all abstract subprograms and all subprograms that require overriding; if more than one such homograph remains that is not thus overridden, then if they are all fully conformant with one another, one is chosen arbitrarily; if not, they are all hidden from all visibility.

13

- For an implicit declaration of a primitive subprogram in a generic unit, there is a copy of this declaration in an instance. However, a whole new set of primitive subprograms is implicitly declared for each type declared within the visible part of the instance. These new declarations occur immediately after the type declaration, and override the copied ones. The copied ones can be called only from within the instance; the new ones

can be called only from outside the instance, although for tagged types, the body of a new one can be executed by a call to an old one.

14

A declaration is visible within its scope, except where hidden from all visibility, as follows:

15

- An overridden declaration is hidden from all visibility within the scope of the overriding declaration.

16

- A declaration is hidden from all visibility until the end of the declaration, except:

17

- For a record type or record extension, the declaration is hidden from all visibility only until the reserved word `record`;

18/2

- For a `package_declaration`, `generic-package_declaration` (see [S0254], page 450), or `subprogram_body` (see [S0162], page 261), the declaration is hidden from all visibility only until the reserved word is of the declaration;

18.1/2

- For a task declaration or protected declaration, the declaration is hidden from all visibility only until the reserved word `with` of the declaration if there is one, or the reserved word is of the declaration if there is no `with`.

19

- If the completion of a declaration is a declaration, then within the scope of the completion, the first declaration is hidden from all visibility. Similarly, a `discriminant_specification` (see [S0062], page 123) or `parameter_specification` (see [S0160], page 256) is hidden within the scope of a corresponding `discriminant_specification` (see [S0062], page 123) or `parameter_specification` (see [S0160], page 256) of a corresponding completion, or of a corresponding `accept_statement` (see [S0201], page 347).

20/2

- The declaration of a library unit (including a `library_unit_renaming_declaration`) is hidden from all visibility at places outside its declarative region that are not within the scope of a `nonlimited_with_clause` that mentions it. The limited view of a library package is hidden from all visibility at places that are not within the scope of a `limited_with_clause` that mentions it; in addition, the limited view is hidden from all visibility within the declarative region of the package, as well as within the scope of any `nonlimited_with_clause` that mentions the package. Where the declaration of the limited view of a package is visible, any name that denotes the package denotes the limited view, including those provided by a package renaming.

20.1/2

- For each declaration or renaming of a generic unit as a child of some parent generic package, there is a corresponding declaration nested immediately within each instance of the parent. Such a nested declaration is hidden from all visibility except at places that are within the scope of a `with_clause` that mentions the child.

21

A declaration with a `defining_identifier` or `defining_operator_symbol` is immediately visible (and hence directly visible) within its immediate scope except where hidden from direct visibility, as follows:

22

- A declaration is hidden from direct visibility within the immediate scope of a homograph of the declaration, if the homograph occurs within an inner declarative region;

23

- A declaration is also hidden from direct visibility where hidden from all visibility.

23.1/2

An `attribute_definition_clause` is `<visible>` everywhere within its scope.

Name Resolution Rules

24

A `direct_name` shall resolve to denote a directly visible declaration whose defining name is the same as the `direct_name`. A `selector_name` shall resolve to denote a visible declaration whose defining name is the same as the `selector_name`.

25

These rules on visibility and direct visibility do not apply in a `context_clause`, a `parent_unit_name`, or a pragma that appears at the place of a `compilation_unit`. For those contexts, see the rules in Section 11.1.6 [10.1.6], page 409, "Section 11.1.6 [10.1.6], page 409, Environment-Level Visibility Rules".

Legality Rules

26/2

A non-overridable declaration is illegal if there is a homograph occurring immediately

within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the non-overridable declaration. In addition, a type extension is illegal if somewhere within its immediate scope it has two visible components with the same name. Similarly, the `context_clause` for a compilation unit is illegal if it mentions (in a `with_clause`) some library unit, and there is a homograph of the library unit that is visible at the place of the compilation unit, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant.

NOTES

27

5 Visibility for compilation units follows from the definition of the environment in Section 11.1.4 [10.1.4], page 406, except that it is necessary to apply a `with_clause` to obtain visibility to a `library_unit_declaration` or `library_unit_renaming_declaration`.

28

6 In addition to the visibility rules given above, the meaning of the occurrence of a `direct_name` or `selector_name` at a given place in the text can depend on the overloading rules (see Section 9.6 [8.6], page 324).

29

7 Not all contexts where an identifier, `character_literal`, or `operator_symbol` are allowed require visibility of a corresponding declaration. Contexts where visibility is not required are identified by using one of these three syntactic categories directly in a syntax rule, rather than using `direct_name` or `selector_name`.

9.3.1 8.3.1 Overriding Indicators

1/2

An `overriding_indicator` is used to declare that an operation is intended to override (or not override) an inherited operation.

Syntax

2/2

`overriding_indicator ::= [not] overriding`
Legality Rules

3/2

If an `abstract_subprogram_declaration` (see [S0076], page 150), `null_procedure_declaration` (see [S0172], page 277), `subprogram_body`, `subprogram_body_stub` (see [S0241], page 403),

subprogram_renaming_declaration (see [S0186], page 320), generic_instantiation (see [S0257], page 455) of a subprogram, or subprogram_declaration (see [S0148], page 255) other than a protected subprogram has an overriding_indicator (see [S0178], page 312), then:

4/2

- the operation shall be a primitive operation for some type;

5/2

- if the overriding_indicator is overriding, then the operation shall override a homograph at the place of the declaration or body;

6/2

- if the overriding_indicator is not overriding, then the operation shall not override any homograph (at any place).

7/2

In addition to the places where Legality Rules normally apply, these rules also apply in the private part of an instance of a generic unit.

NOTES

8/2

8 Rules for overriding_indicators of task and protected entries and of protected subprograms are found in Section 10.5.2 [9.5.2], page 347, and Section 10.4 [9.4], page 337, respectively.

Examples

9/2

The use of overriding_indicators allows the detection of errors at compile-time that otherwise might not be detected at all. For instance, we might declare a security queue derived from the Queue interface of 3.9.4 as:

10/2

```
type Security_Queue is new Queue with record ...;
```

11/2

```
overriding
procedure Append(Q : in out Security_Queue; Person : in Person_Name);■
```

12/2

```
overriding
procedure Remove_First(Q : in out Security_Queue; Person : in Person_Name);■
```

13/2

```
overriding
```

```
function Cur_Count(Q : in Security_Queue) return Natural;
```

14/2

```
overriding
```

```
function Max_Count(Q : in Security_Queue) return Natural;
```

15/2

```
not overriding
```

```
procedure Arrest(Q : in out Security_Queue; Person : in Person_Name);
```

16/2

The first four subprogram declarations guarantee that these subprograms will override the four subprograms inherited from the Queue interface. A misspelling in one of these subprograms will be detected by the implementation. Conversely, the declaration of Arrest guarantees that this is a new operation.

9.4 8.4 Use Clauses

1

A `use_package_clause` achieves direct visibility of declarations that appear in the visible part of a package; a `use_type_clause` achieves direct visibility of the primitive operators of a type.

Syntax

2

```
use_clause ::= use_package_clause | use_type_clause
```

3

```
use_package_clause ::= use <package_>name {, <package_>name};
```

4

```
use_type_clause ::= use type subtype_mark {, subtype_mark};
```

Legality Rules

5/2

A `<package_>name` of a `use_package_clause` shall denote a nonlimited view of a package.

Static Semantics

6

For each `use_clause`, there is a certain region of text called the `<scope>` of the `use_clause`. For a `use_clause` within a `context_clause` of a `library_unit_declaration` or `library_unit_renaming_declaration`, the scope is the entire declarative region of the declaration. For a `use_clause` within a `context_clause` of a body, the scope is the entire body and any subunits (including multiply nested subunits). The scope does not include `context_clauses` themselves.

7

For a `use_clause` immediately within a declarative region, the scope is the portion of the

declarative region starting just after the `use_clause` and extending to the end of the declarative region. However, the scope of a `use_clause` in the private part of a library unit does not include the visible part of any public descendant of that library unit.

7.1/2

A package is `<named>` in a `use_package_clause` if it is denoted by a `<package_>name` of that clause. A type is `<named>` in a `use_type_clause` if it is determined by a `subtype_mark` of that clause.

8/2

For each package named in a `use_package_clause` whose scope encloses a place, each declaration that occurs immediately within the declarative region of the package is `<potentially use-visible>` at this place if the declaration is visible at this place. For each type `<T>` or `<T>'Class` named in a `use_type_clause` whose scope encloses a place, the declaration of each primitive operator of type `<T>` is `<potentially use-visible>` at this place if its declaration is visible at this place.

9

A declaration is `<use-visible>` if it is `<potentially use-visible>`, except in these naming-conflict cases:

10

- A `<potentially use-visible>` declaration is not `<use-visible>` if the place considered is within the immediate scope of a homograph of the declaration.

11

- `<Potentially use-visible>` declarations that have the same identifier are not `<use-visible>` unless each of them is an overloadable declaration.

Dynamic Semantics

12

The elaboration of a `use_clause` has no effect.

Examples

13

`<Example of a use clause in a context clause:>`

14

```
with Ada.Calendar; use Ada;
```

15

`<Example of a use type clause:>`

16

```
use type Rational_Numbers.Rational; --< see Section 8.1 [7.1], page 279>■  
Two_Thirds: Rational_Numbers.Rational := 2/3;
```

9.5 8.5 Renaming Declarations

1

A renaming_declaration declares another name for an entity, such as an object, exception, package, subprogram, entry, or generic unit. Alternatively, a subprogram_renaming_declaration can be the completion of a previous subprogram_declaration.

Syntax

2

```
renaming_declaration ::=  
    object_renaming_declaration  
  | exception_renaming_declaration  
  | package_renaming_declaration  
  | subprogram_renaming_declaration  
  | generic_renaming_declaration
```

Dynamic Semantics

3

The elaboration of a renaming_declaration evaluates the name that follows the reserved word renames and thereby determines the view and entity denoted by this name (the <renamed view> and <renamed entity>). A name that denotes the renaming_declaration denotes (a new view of) the renamed entity.

NOTES

4

9 Renaming may be used to resolve name conflicts and to act as a shorthand. Renaming with a different identifier or operator_symbol does not hide the old name; the new name and the old name need not be visible at the same places.

5

10 A task or protected object that is declared by an explicit object_declaration can be renamed as an object. However, a single task or protected object cannot be renamed since the corresponding type is anonymous (meaning it has no nameable subtypes). For similar reasons, an object of an anonymous array or access type cannot be renamed.

6

11 A subtype defined without any additional constraint can be used to achieve the effect of renaming another subtype (including a task or protected subtype) as in

7

```
subtype Mode is Ada.Text_IO.File_Mode;
```

9.5.1 8.5.1 Object Renaming Declarations

1

An `object_renaming_declaration` is used to rename an object.

Syntax

2/2

```
object_renaming_declaration ::=  
    defining_identifier : [null_exclusion] subtype_mark renames <object_>name;
```

```
    | defining_identifier : access_definition renames <object_>name;
```

Name Resolution Rules

3/2

The type of the `<object_>name` shall resolve to the type determined by the `subtype_mark`, or in the case where the type is defined by an `access_definition`, to an anonymous access type. If the anonymous access type is an `access-to-object` type, the type of the `<object_>name` shall have the same designated type as that of the `access_definition`. If the anonymous access type is an `access-to-subprogram` type, the type of the `<object_>name` shall have a designated profile that is type conformant with that of the `access_definition`.

Legality Rules

4

The renamed entity shall be an object.

4.1/2

In the case where the type is defined by an `access_definition`, the type of the renamed object and the type defined by the `access_definition`:

4.2/2

- shall both be `access-to-object` types with statically matching designated subtypes and with both or neither being `access-to-constant` types; or

4.3/2

- shall both be `access-to-subprogram` types with subtype conformant designated profiles.

4.4/2

For an `object_renaming_declaration` with a `null_exclusion` or an `access_definition` that has a `null_exclusion`:

4.5/2

- if the `<object_>name` denotes a generic formal object of a generic unit `<G>`, and the `object_renaming_declaration` occurs within the body of `<G>` or within the body of a generic unit declared within the declarative region of `<G>`, then the declaration of the formal object of `<G>` shall have a `null_exclusion`;

4.6/2

- otherwise, the subtype of the <object_>name shall exclude null. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit.

5/2

The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is constrained by its initial value. A slice of an array shall not be renamed if this restriction disallows renaming of the array. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit. These rules also apply for a renaming that appears in the body of a generic unit, with the additional requirement that even if the nominal subtype of the variable is indefinite, its type shall not be a descendant of an untagged generic formal derived type.

Static Semantics

6/2

An object_renaming_declaration declares a new view of the renamed object whose properties are identical to those of the renamed view. Thus, the properties of the renamed object are not affected by the renaming_declaration. In particular, its value and whether or not it is a constant are unaffected; similarly, the null exclusion or constraints that apply to an object are not affected by renaming (any constraint implied by the subtype_mark or access_definition of the object_renaming_declaration is ignored).

Examples

7

<Example of renaming an object:>

8

```

declare
  L : Person renames Leftmost_Person; --< see Section 4.10.1 [3.10.1],
page 160>
begin
  L.Age := L.Age + 1;
end;
```

9.5.2 8.5.2 Exception Renaming Declarations

1

An exception_renaming_declaration is used to rename an exception.

Syntax

2

```

exception_renaming_declaration ::= defining_identifier : exception renames <exception_>name;
Legality Rules
```

3

The renamed entity shall be an exception.

Static Semantics

4

An exception_renaming_declaration declares a new view of the renamed exception.

Examples

5

<Example of renaming an exception:>

6

```
EOF : exception renames Ada.IO_Exceptions.End_Error; -- see Section 15.13
[A.13], page 752>
```

9.5.3 8.5.3 Package Renaming Declarations

1

A `package_renaming_declaration` is used to rename a package.

Syntax

2

```
package_renaming_declaration ::= package defining_program_unit_name renames <package_>name;
Legality Rules
```

3

The renamed entity shall be a package.

3.1/2

If the `<package_>name` of a `package_renaming_declaration` denotes a limited view of a package `<P>`, then a name that denotes the `package_renaming_declaration` shall occur only within the immediate scope of the renaming or the scope of a `with_clause` that mentions the package `<P>` or, if `<P>` is a nested package, the innermost library package enclosing `<P>`.

Static Semantics

4

A `package_renaming_declaration` declares a new view of the renamed package.

4.1/2

At places where the declaration of the limited view of the renamed package is visible, a name that denotes the `package_renaming_declaration` denotes a limited view of the package (see Section 11.1.1 [10.1.1], page 394).

Examples

5

<Example of renaming a package:>

6

```
package TM renames Table_Manager;
```

9.5.4 8.5.4 Subprogram Renaming Declarations

1

A `subprogram_renaming_declaration` can serve as the completion of a `subprogram_declaration`; such a renaming_declaration is called a `<renaming-as-body>`. A `subprogram_renaming_declaration` that is not a completion is called a

<renaming-as-declaration>, and is used to rename a subprogram (possibly an enumeration literal) or an entry.

Syntax

2/2

```
subprogram_renaming_declaration ::=
    [overriding_indicator]
    subprogram_specification renames <callable_entity->name;
```

Name Resolution Rules

3

The expected profile for the <callable_entity->name is the profile given in the subprogram_specification.

Legality Rules

4

The profile of a renaming-as-declaration shall be mode-conformant with that of the renamed callable entity.

4.1/2

For a parameter or result subtype of the subprogram_specification that has an explicit null_exclusion:

4.2/2

- if the <callable_entity->name denotes a generic formal subprogram of a generic unit <G>, and the subprogram_renaming_declaration occurs within the body of a generic unit <G> or within the body of a generic unit declared within the declarative region of the generic unit <G>, then the corresponding parameter or result subtype of the formal subprogram of <G> shall have a null_exclusion;

4.3/2

- otherwise, the subtype of the corresponding parameter or result type of the renamed callable entity shall exclude null. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit.

5/1

The profile of a renaming-as-body shall conform fully to that of the declaration it completes. If the renaming-as-body completes that declaration before the subprogram it declares is frozen, the profile shall be mode-conformant with that of the renamed callable entity and the subprogram it declares takes its convention from the renamed subprogram; otherwise, the profile shall be subtype-conformant with that of the renamed callable entity and the convention of the renamed subprogram shall not be Intrinsic. A renaming-as-body is illegal if the declaration occurs before the subprogram whose declaration it completes is frozen, and the renaming renames the subprogram itself, through one or more subprogram renaming declarations, none of whose subprograms has been frozen.

5.1/2

The <callable_entity->name of a renaming shall not denote a subprogram that requires overriding (see Section 4.9.3 [3.9.3], page 149).

5.2/2

The <callable_entity_>name of a renaming-as-body shall not denote an abstract subprogram.

6

A name that denotes a formal parameter of the subprogram_specification is not allowed within the <callable_entity_>name.

Static Semantics

7

A renaming-as-declaration declares a new view of the renamed entity. The profile of this new view takes its subtypes, parameter modes, and calling convention from the original profile of the callable entity, while taking the formal parameter names and default_expressions from the profile given in the subprogram_renaming_declaration. The new view is a function or procedure, never an entry.

Dynamic Semantics

7.1/1

For a call to a subprogram whose body is given as a renaming-as-body, the execution of the renaming-as-body is equivalent to the execution of a subprogram_body that simply calls the renamed subprogram with its formal parameters as the actual parameters and, if it is a function, returns the value of the call.

8

For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called.

Bounded (Run-Time) Errors

8.1/1

If a subprogram directly or indirectly renames itself, then it is a bounded error to call that subprogram. Possible consequences are that Program_Error or Storage_Error is raised, or that the call results in infinite recursion.

NOTES

9

12 A procedure can only be renamed as a procedure. A function whose defining_designator is either an identifier or an operator_symbol can be renamed with either an identifier or an operator_symbol; for renaming as an operator, the subprogram specification given in the renaming_declaration is subject to the rules given in Section 7.6 [6.6], page 276, for operator declarations. Enumeration literals can be renamed as functions; similarly, attribute_references that denote functions (such as references to Succ and Pred) can be renamed as functions. An entry can only be renamed as a procedure; the new name is only allowed to appear in contexts that allow a procedure name. An entry of a family can be renamed, but an entry family cannot be renamed as a whole.

10

13 The operators of the root numeric types cannot be renamed because the types in the profile are anonymous, so the corresponding specifications cannot be written; the same holds for certain attributes, such as Pos.

11

14 Calls with the new name of a renamed entry are procedure_call_statements and are not allowed at places where the syntax requires an entry_call_statement in conditional_ and timed_entry_calls, nor in an asynchronous_select; similarly, the Count attribute is not available for the new name.

12

15 The primitiveness of a renaming-as-declaration is determined by its profile, and by where it occurs, as for any declaration of (a view of) a subprogram; primitiveness is not determined by the renamed view. In order to perform a dispatching call, the subprogram name has to denote a primitive subprogram, not a non-primitive renaming of a primitive subprogram.

Examples

13

<Examples of subprogram renaming declarations:>

14

```
procedure My_Write(C : in Character) renames Pool(K).Write; --< see Section 5.1.
[4.1.3], page 183>
```

15

```
function Real_Plus(Left, Right : Real ) return Real renames "+";
function Int_Plus (Left, Right : Integer) return Integer renames "+";
```

16

```
function Rouge return Color renames Red; --< see Section 4.5.1 [3.5.1],
page 92>
function Rot return Color renames Red;
function Rosso return Color renames Rouge;
```

17

```
function Next(X : Color) return Color renames Color'Succ; --< see Section 4.5.1
[3.5.1], page 92>
```

18

<Example of a subprogram renaming declaration with new parameter names:>

19

```
function "*" (X,Y : Vector) return Real renames Dot_Product; --< see Section 7.1
[6.1], page 255>
```

20

<Example of a subprogram renaming declaration with a new default expression:>

21

```
function Minimum(L : Link := Head) return Cell renames Min_Cell; --< see Section
[6.1], page 255>
```

9.5.5 8.5.5 Generic Renaming Declarations

1

A `generic_renaming_declaration` is used to rename a generic unit.

Syntax

2

```
generic_renaming_declaration ::=
    generic package defining_program_unit_name renames <generic_package_>name;
    | generic procedure defining_program_unit_name renames <generic_procedure_>name;
    | generic function defining_program_unit_name renames <generic_function_>name;
```

Legality Rules

3

The renamed entity shall be a generic unit of the corresponding kind.

Static Semantics

4

A `generic_renaming_declaration` declares a new view of the renamed generic unit.

NOTES

5

16 Although the properties of the new view are the same as those of the renamed view, the place where the `generic_renaming_declaration` occurs may affect the legality of subsequent renamings and instantiations that denote the `generic_renaming_declaration`, in particular if the renamed generic unit is a library unit (see Section 11.1.1 [10.1.1], page 394).

Examples

6

<Example of renaming a generic unit:>

7

```
generic package Enum_IO renames Ada.Text_IO.Enumeration_IO; <-- see Section 15.1
[A.10.10], page 736>
```

9.6 8.6 The Context of Overload Resolution

1

Because declarations can be overloaded, it is possible for an occurrence of a usage name to have more than one possible interpretation; in most cases, ambiguity is disallowed. This clause describes how the possible interpretations resolve to the actual interpretation.

2

Certain rules of the language (the Name Resolution Rules) are considered "overloading rules". If a possible interpretation violates an overloading rule, it is assumed not to be the intended interpretation; some other possible interpretation is assumed to be the actual interpretation. On the other hand, violations of non-overloading rules do not affect which interpretation is chosen; instead, they cause the construct to be illegal. To be legal, there usually has to be exactly one acceptable interpretation of a construct that is a "complete context", not counting any nested complete contexts.

3

The syntax rules of the language and the visibility rules given in Section 9.3 [8.3], page 308, determine the possible interpretations. Most type checking rules (rules that require a particular type, or a particular class of types, for example) are overloading rules. Various rules for the matching of formal and actual parameters are overloading rules.

Name Resolution Rules

4

Overload resolution is applied separately to each <complete context>, not counting inner complete contexts. Each of the following constructs is a <complete context>:

5

- A context_item.

6

- A declarative_item or declaration.

7

- A statement.

8

- A pragma_argument_association.

9

- The expression of a case_statement.

10

An (overall) <interpretation> of a complete context embodies its meaning, and includes

the following information about the constituents of the complete context, not including constituents of inner complete contexts:

11

- for each constituent of the complete context, to which syntactic categories it belongs, and by which syntax rules; and

12

- for each usage name, which declaration it denotes (and, therefore, which view and which entity it denotes); and

13

- for a complete context that is a `declarative_item`, whether or not it is a completion of a declaration, and (if so) which declaration it completes.

14

A <possible interpretation> is one that obeys the syntax rules and the visibility rules. An <acceptable interpretation> is a possible interpretation that obeys the <overloading rules>, that is, those rules that specify an expected type or expected profile, or specify how a construct shall <resolve> or be <interpreted>.

15

The <interpretation> of a constituent of a complete context is determined from the overall interpretation of the complete context as a whole. Thus, for example, "interpreted as a `function_call`," means that the construct's interpretation says that it belongs to the syntactic category `function_call`.

16

Each occurrence of a usage name <denotes> the declaration determined by its interpretation. It also denotes the view declared by its denoted declaration, except in the following cases:

17/2

- If a usage name appears within the declarative region of a `type_declaration` and denotes that same `type_declaration`, then it denotes the <current instance> of the type (rather than the type itself); the current instance of a type is the object or value of the type that is associated with the execution that evaluates the usage name. This rule does not apply if the usage name appears within the `subtype_mark` of an `access_definition` for an `access-to-object` type, or within the subtype of a parameter or result of an `access-to-subprogram` type.

18

- If a usage name appears within the declarative region of a `generic_declaration` (but not within its `generic_formal_part`) and it denotes that same `generic_declaration`, then it denotes the <current instance> of the generic unit (rather than the generic unit itself). See also Section 13.3 [12.3], page 454.

19

A usage name that denotes a view also denotes the entity of that view.

20/2

The <expected type> for a given expression, name, or other construct determines, according to the <type resolution rules> given below, the types considered for the construct during overload resolution. The type resolution rules provide support for class-wide programming, universal literals, dispatching operations, and anonymous access types:

21

- If a construct is expected to be of any type in a class of types, or of the universal or class-wide type for a class, then the type of the construct shall resolve to a type in that class or to a universal type that covers the class.

22

- If the expected type for a construct is a specific type <T>, then the type of the construct shall resolve either to <T>, or:

23

- to <T>'Class; or

24

- to a universal type that covers <T>; or

25/2

- when <T> is a specific anonymous access-to-object type (see Section 4.10 [3.10], page 156) with designated type <D>, to an access-to-object type whose designated type is <D>'Class or is covered by <D>; or

25.1/2

- when <T> is an anonymous access-to-subprogram type (see Section 4.10 [3.10], page 156), to an access-to-subprogram type whose designated profile is type-conformant with that of <T>.

26

In certain contexts, such as in a subprogram_renaming_declaration, the Name Resolution Rules define an <expected profile> for a given name; in such cases, the name shall resolve to the name of a callable entity whose profile is type conformant with the expected profile.

Legality Rules

27/2

When a construct is one that requires that its expected type be a <single> type in a given class, the type of the construct shall be determinable solely from the context in which the construct appears, excluding the construct itself, but using the requirement that it be in the given class. Furthermore, the context shall not be one that expects any type in some class that contains types of the given class; in particular, the construct shall not be the operand of a `type_conversion`.

28

A complete context shall have at least one acceptable interpretation; if there is exactly one, then that one is chosen.

29

There is a <preference> for the primitive operators (and ranges) of the root numeric types <root_integer> and <root_real>. In particular, if two acceptable interpretations of a constituent of a complete context differ only in that one is for a primitive operator (or range) of the type <root_integer> or <root_real>, and the other is not, the interpretation using the primitive operator (or range) of the root numeric type is <preferred>.

30

For a complete context, if there is exactly one overall acceptable interpretation where each constituent's interpretation is the same as or preferred (in the above sense) over those in all other overall acceptable interpretations, then that one overall acceptable interpretation is chosen. Otherwise, the complete context is <ambiguous>.

31

A complete context other than a `pragma_argument_association` shall not be ambiguous.

32

A complete context that is a `pragma_argument_association` is allowed to be ambiguous (unless otherwise specified for the particular pragma), but only if every acceptable interpretation of the pragma argument is as a name that statically denotes a callable entity. Such a name denotes all of the declarations determined by its interpretations, and all of the views declared by these declarations.

NOTES

33

17 If a usage name has only one acceptable interpretation, then it denotes the corresponding entity. However, this does not mean that the usage name is necessarily legal since other requirements exist which are not considered for overload resolution; for example, the fact that an expression is static, whether an object is constant, mode and subtype conformance rules, freezing rules, order of elaboration, and so on.

34

Similarly, subtypes are not considered for overload resolution (the violation of a constraint does not make a program illegal but raises an exception during program execution).

10 9 Tasks and Synchronization

1

The execution of an Ada program consists of the execution of one or more <tasks>. Each task represents a separate thread of control that proceeds independently and concurrently between the points where it <interacts> with other tasks. The various forms of task interaction are described in this section, and include:

2

- the activation and termination of a task;

3

- a call on a protected subprogram of a <protected object>, providing exclusive read–write access, or concurrent read–only access to shared data;

4

- a call on an entry, either of another task, allowing for synchronous communication with that task, or of a protected object, allowing for asynchronous communication with one or more other tasks using that same protected object;

5

- a timed operation, including a simple delay statement, a timed entry call or accept, or a timed asynchronous select statement (see next item);

6

- an asynchronous transfer of control as part of an asynchronous select statement, where a task stops what it is doing and begins execution at a different point in response to the completion of an entry call or the expiration of a delay;

7

- an abort statement, allowing one task to cause the termination of another task.

8

In addition, tasks can communicate indirectly by reading and updating (unprotected) shared variables, presuming the access is properly synchronized through some other kind of task interaction.

Static Semantics

9

The properties of a task are defined by a corresponding task declaration and `task_body`, which together define a program unit called a <task unit>.

Dynamic Semantics

10

Over time, tasks proceed through various <states>. A task is initially <inactive>; upon

activation, and prior to its <termination> it is either <blocked> (as part of some task interaction) or <ready> to run. While ready, a task competes for the available <execution resources> that it requires to run.

NOTES

11

1 Concurrent task execution may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a given task are performed by different physical processors acting in parallel, it may choose to perform them in this way.

10.1 9.1 Task Units and Task Objects

1

A task unit is declared by a <task declaration>, which has a corresponding `task_body`. A task declaration may be a `task_type_declaration`, in which case it declares a named task type; alternatively, it may be a `single_task_declaration`, in which case it defines an anonymous task type, as well as declaring a named task object of that type.

Syntax

2/2

```
task_type_declaration ::=
  task type defining_identifier [known_discriminant_part] [is
    [new interface_list with]
    task_definition];
```

3/2

```
single_task_declaration ::=
  task defining_identifier [is
    [new interface_list with]
    task_definition];
```

4

```
task_definition ::=
  {task_item}
  [ private
    {task_item}]
  end [<task_>identifier]
```

5/1

```
task_item ::= entry_declaration | aspect_clause
```

6

```
task_body ::=  
  task body defining_identifier is  
    declarative_part  
  begin  
    handled_sequence_of_statements  
  end [<task->identifier];
```

7

If a <task->identifier appears at the end of a task_definition or task_body, it shall repeat the defining_identifier.

Legality Rules

8/2

<This paragraph was deleted.>

Static Semantics

9

A task_definition defines a task type and its first subtype. The first list of task_items of a task_definition (see [S0190], page 329), together with the known_discriminant_part (see [S0061], page 123), if any, is called the visible part of the task unit. The optional list of task_items after the reserved word private is called the private part of the task unit.

9.1/1

For a task declaration without a task_definition, a task_definition without task_items is assumed.

9.2/2

For a task declaration with an interface_list, the task type inherits user-defined primitive subprograms from each progenitor type (see Section 4.9.4 [3.9.4], page 152), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see Section 4.4 [3.4], page 66). If the first parameter of a primitive inherited subprogram is of the task type or an access parameter designating the task type, and there is an entry_declaration for a single entry with the same identifier within the task declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be <implemented> by the conforming task entry.

Legality Rules

9.3/2

A task declaration requires a completion, which shall be a task_body, and every task_body shall be the completion of some task declaration.

9.4/2

Each <interface->subtype_mark of an interface_list appearing within a task declaration shall denote a limited interface type that is not a protected interface.

9.5/2

The prefixed view profile of an explicitly declared primitive subprogram of a tagged task type shall not be type conformant with any entry of the task type, if the first parameter of the subprogram is of the task type or is an access parameter designating the task type.

9.6/2

For each primitive subprogram inherited by the type declared by a task declaration, at most one of the following shall apply:

9.7/2

- the inherited subprogram is overridden with a primitive subprogram of the task type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or

9.8/2

- the inherited subprogram is implemented by a single entry of the task type; in which case its prefixed view profile shall be subtype conformant with that of the task entry.

9.9/2

If neither applies, the inherited subprogram shall be a null procedure. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), these rules also apply in the private part of an instance of a generic unit.

Dynamic Semantics

10

The elaboration of a task declaration elaborates the `task_definition`. The elaboration of a `single_task_declaration` (see [S0189], page 329) also creates an object of an (anonymous) task type.

11

The elaboration of a `task_definition` creates the task type and its first subtype; it also includes the elaboration of the `entry_declarations` in the given order.

12/1

As part of the initialization of a task object, any `aspect_clauses` and any per-object constraints associated with `entry_declaration` (see [S0200], page 347)s of the corresponding `task_definition` (see [S0190], page 329) are elaborated in the given order.

13

The elaboration of a `task_body` has no effect other than to establish that tasks of the type can from then on be activated without failing the `Elaboration_Check`.

14

The execution of a `task_body` is invoked by the activation of a task of the corresponding type (see Section 10.2 [9.2], page 333).

15

The content of a task object of a given task type includes:

16

- The values of the discriminants of the task object, if any;

17

- An entry queue for each entry of the task object;

18

- A representation of the state of the associated task.

NOTES

19/2

2 Other than in an `access_definition`, the name of a task unit within the declaration or body of the task unit denotes the current instance of the unit (see Section 9.6 [8.6], page 324), rather than the first subtype of the corresponding task type (and thus the name cannot be used as a `subtype_mark`).

20

3 The notation of a `selected_component` can be used to denote a discriminant of a task (see Section 5.1.3 [4.1.3], page 183). Within a task unit, the name of a discriminant of the task type denotes the corresponding discriminant of the current instance of the unit.

21/2

4 A task type is a limited type (see Section 8.5 [7.5], page 292), and hence precludes use of `assignment_statements` and predefined equality operators. If an application needs to store and exchange task identities, it can do so by defining an access type designating the corresponding task objects and by using access values for identification purposes. Assignment is available for such an access type as for any access type. Alternatively, if the implementation supports the Systems Programming Annex, the `Identity` attribute can be used for task identification (see Section 17.7.1 [C.7.1], page 965).

Examples

22

<Examples of declarations of task types:>

23

```
task type Server is
  entry Next_Work_Item(WI : in Work_Item);
  entry Shut_Down;
end Server;
```

24/2

```
task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is
  new Serial_Device with --< see Section 4.9.4 [3.9.4], page 152>■
  entry Read (C : out Character);
  entry Write(C : in Character);
end Keyboard_Driver;
```

25

<Examples of declarations of single tasks:>

26

```
task Controller is
  entry Request(Level)(D : Item); --< a family of entries>
end Controller;
```

27

```
task Parser is
  entry Next_Lexeme(L : in Lexical_Element);
  entry Next_Action(A : out Parser_Action);
end;
```

28

```
task User; --< has no entries>
```

29

<Examples of task objects:>

30

```
Agent      : Server;
Teletype   : Keyboard_Driver(TTY_ID);
Pool       : array(1 .. 10) of Keyboard_Driver;
```

31

<Example of access type designating task objects:>

32

```
type Keyboard is access Keyboard_Driver;
Terminal : Keyboard := new Keyboard_Driver(Term_ID);
```

10.2 9.2 Task Execution - Task Activation

Dynamic Semantics

1

The execution of a task of a given task type consists of the execution of the corresponding task_body. The initial part of this execution is called the <activation> of the task; it consists of the elaboration of the declarative_part of the task_body. Should an exception be propagated by the elaboration of its declarative_part, the activation of the task is defined to have <failed>, and it becomes a completed task.

2/2

A task object (which represents one task) can be a part of a stand-alone object, of an object created by an allocator, or of an anonymous object of a limited type, or a coextension of one of these. All tasks that are part or coextensions of any of the stand-alone objects created by the elaboration of object_declaration (see [S0032], page 61)s (or generic_associations of

formal objects of mode in) of a single declarative region are activated together. All tasks that are part or coextensions of a single object that is not a stand-alone object are activated together.

3/2

For the tasks of a given declarative region, the activations are initiated within the context of the `handled_sequence_of_statements` (see [S0247], page 420) (and its associated `exception_handler` (see [S0248], page 420)s if any — see Section 12.2 [11.2], page 420), just prior to executing the statements of the `handled_sequence_of_statements`. For a package without an explicit body or an explicit `handled_sequence_of_statements` (see [S0247], page 420), an implicit body or an implicit `null_statement` (see [S0134], page 241) is assumed, as defined in Section 8.2 [7.2], page 281.

4/2

For tasks that are part or coextensions of a single object that is not a stand-alone object, activations are initiated after completing any initialization of the outermost object enclosing these tasks, prior to performing any other operation on the outermost object. In particular, for tasks that are part or coextensions of the object created by the evaluation of an allocator, the activations are initiated as the last step of evaluating the allocator, prior to returning the new access value. For tasks that are part or coextensions of an object that is the result of a function call, the activations are not initiated until after the function returns.

5

The task that created the new tasks and initiated their activations (the `<activator>`) is blocked until all of these activations complete (successfully or not). Once all of these activations are complete, if the activation of any of the tasks has failed (due to the propagation of an exception), `Tasking_Error` is raised in the activator, at the place at which it initiated the activations. Otherwise, the activator proceeds with its execution normally. Any tasks that are aborted prior to completing their activation are ignored when determining whether to raise `Tasking_Error`.

6

Should the task that created the new tasks never reach the point where it would initiate the activations (due to an abort or the raising of an exception), the newly created tasks become terminated and are never activated.

NOTES

7

5 An entry of a task can be called before the task has been activated.

8

6 If several tasks are activated together, the execution of any of these tasks need not await the end of the activation of the other tasks.

9

7 A task can become completed during its activation either because of an exception or because it is aborted (see Section 10.8 [9.8], page 385).

Examples

10

<Example of task activation:>

11

```
procedure P is
  A, B : Server;    --< elaborate the task objects A, B>
  C    : Server;    --< elaborate the task object C>
begin
  --< the tasks A, B, C are activated together before the first statement>
  ...
end;
```

10.3 9.3 Task Dependence - Termination of Tasks

Dynamic Semantics

1

Each task (other than an environment task -- see Section 11.2 [10.2], page 409) <depends> on one or more masters (see Section 8.6.1 [7.6.1], page 299), as follows:

2

- If the task is created by the evaluation of an allocator for a given access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type.

3

- If the task is created by the elaboration of an object_declaration, it depends on each master that includes this elaboration.

3.1/2

- Otherwise, the task depends on the master of the outermost object of which it is a part (as determined by the accessibility level of that object -- see Section 4.10.2 [3.10.2], page 164, and Section 8.6.1 [7.6.1], page 299), as well as on any master whose execution includes that of the master of the outermost object.

4

Furthermore, if a task depends on a given master, it is defined to depend on the task that executes the master, and (recursively) on any master of that task.

5

A task is said to be <completed> when the execution of its corresponding task_body is completed. A task is said to be <terminated> when any finalization of the task_body has been performed (see Section 8.6.1 [7.6.1], page 299). The first step of finalizing a master (including a task_body) is to wait for the termination of any tasks dependent on the master. The task executing the master is blocked until all the dependents have terminated. Any remaining finalization is then performed and the master is left.

6/1

Completion of a task (and the corresponding task_body) can occur when the task is blocked at a select_statement (see [S0212], page 377) with an open terminate_alternative (see Section 10.7.1 [9.7.1], page 378); the open terminate_alternative is selected if and only if the following conditions are satisfied:

7/2

- The task depends on some completed master; and

8

- Each task that depends on the master considered is either already terminated or similarly blocked at a select_statement with an open terminate_alternative.

9

When both conditions are satisfied, the task considered becomes completed, together with all tasks that depend on the master considered that are not yet completed.

NOTES

10

8 The full view of a limited private type can be a task type, or can have subcomponents of a task type. Creation of an object of such a type creates dependences according to the full type.

11

9 An object_renaming_declaration defines a new view of an existing entity and hence creates no further dependence.

12

10 The rules given for the collective completion of a group of tasks all blocked on select_statements with open terminate_alternatives ensure that the collective completion can occur only when there are no remaining active tasks that could call one of the tasks being collectively completed.

13

11 If two or more tasks are blocked on select_statements with open terminate_alternatives, and become completed collectively, their finalization actions proceed concurrently.

14

12 The completion of a task can occur due to any of the following:

15

- the raising of an exception during the elaboration of the declarative_part of the corresponding task_body;

16

- the completion of the handled_sequence_of_statements of the corresponding task_body;

17

- the selection of an open terminate_alternative of a select_statement in the corresponding task_body;

18

- the abort of the task.

Examples

19

<Example of task dependence:>

20

```

declare
  type Global is access Server;          --< see Section 10.1 [9.1],
page 329>
  A, B : Server;
  G    : Global;
begin
  --< activation of A and B>
  declare
    type Local is access Server;
    X : Global := new Server;  --< activation of X.all>
    L : Local  := new Server;  --< activation of L.all>
    C : Server;
  begin
    --< activation of C>
    G := X;  --< both G and X designate the same task object>
    ...
  end;  --< await termination of C and L.all (but not X.all)>
  ...
end;  --< await termination of A, B, and G.all>

```

10.4 9.4 Protected Units and Protected Objects

1

A <protected object> provides coordinated access to shared data, through calls on its visible <protected operations>, which can be <protected subprograms> or <protected entries>. A <protected unit> is declared by a <protected declaration>, which has a corresponding protected_body. A protected declaration may be a protected_type_declaration, in which case it

declares a named protected type; alternatively, it may be a `single_protected_declaration`, in which case it defines an anonymous protected type, as well as declaring a named protected object of that type.

Syntax

2/2

```
protected_type_declaration ::=  
  protected_type_defining_identifier [known_discriminant_part] is  
    [new interface_list with]  
    protected_definition;
```

3/2

```
single_protected_declaration ::=  
  protected_defining_identifier is  
    [new interface_list with]  
    protected_definition;
```

4

```
protected_definition ::=  
  { protected_operation_declaration }  
[ private  
  { protected_element_declaration } ]  
end [<protected_>identifier]
```

5/1

```
protected_operation_declaration ::= subprogram_declaration  
  | entry_declaration  
  | aspect_clause
```

6

```
protected_element_declaration ::= protected_operation_declaration  
  | component_declaration
```

7

```
protected_body ::=  
  protected_body_defining_identifier is  
    { protected_operation_item }  
  end [<protected_>identifier];
```

8/1

```
protected_operation_item ::= subprogram_declaration  
  | subprogram_body  
  | entry_body  
  | aspect_clause
```

If a `<protected_>identifier` appears at the end of a `protected_definition` or `protected_body`, it shall repeat the `defining_identifier`.

Legality Rules

10/2

`<This paragraph was deleted.>`

Static Semantics

11/2

A `protected_definition` defines a protected type and its first subtype. The list of `protected_operation_declaration` (see [S0196], page 338)s of a `protected_definition` (see [S0195], page 338), together with the `known_discriminant_part` (see [S0061], page 123), if any, is called the visible part of the protected unit. The optional list of `protected_element_declaration` (see [S0197], page 338)s after the reserved word `private` is called the private part of the protected unit.

11.1/2

For a protected declaration with an `interface_list`, the protected type inherits user-defined primitive subprograms from each progenitor type (see Section 4.9.4 [3.9.4], page 152), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see Section 4.4 [3.4], page 66). If the first parameter of a primitive inherited subprogram is of the protected type or an access parameter designating the protected type, and there is a `protected_operation_declaration` for a protected subprogram or single entry with the same identifier within the protected declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be `<implemented>` by the conforming protected subprogram or entry.

Legality Rules

11.2/2

A protected declaration requires a completion, which shall be a `protected_body` (see [S0198], page 338), and every `protected_body` (see [S0198], page 338) shall be the completion of some protected declaration.

11.3/2

Each `<interface_>subtype_mark` of an `interface_list` appearing within a protected declaration shall denote a limited interface type that is not a task interface.

11.4/2

The prefixed view profile of an explicitly declared primitive subprogram of a tagged protected type shall not be type conformant with any protected operation of the protected type, if the first parameter of the subprogram is of the protected type or is an access parameter designating the protected type.

11.5/2

For each primitive subprogram inherited by the type declared by a protected declaration, at most one of the following shall apply:

11.6/2

- the inherited subprogram is overridden with a primitive subprogram of the protected type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or

11.7/2

- the inherited subprogram is implemented by a protected subprogram or single entry of the protected type, in which case its prefixed view profile shall be subtype conformant with that of the protected subprogram or entry.

11.8/2

If neither applies, the inherited subprogram shall be a null procedure. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), these rules also apply in the private part of an instance of a generic unit.

11.9/2

If an inherited subprogram is implemented by a protected procedure or an entry, then the first parameter of the inherited subprogram shall be of mode out or in out, or an access-to-variable parameter.

11.10/2

If a protected subprogram declaration has an `overriding_indicator`, then at the point of the declaration:

11.11/2

- if the `overriding_indicator` is overriding, then the subprogram shall implement an inherited subprogram;

11.12/2

- if the `overriding_indicator` is not overriding, then the subprogram shall not implement any inherited subprogram.

11.13/2

In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), these rules also apply in the private part of an instance of a generic unit.

Dynamic Semantics

12

The elaboration of a protected declaration elaborates the `protected_definition`. The elaboration of a `single_protected_declaration` (see [S0194], page 338) also creates an object of an (anonymous) protected type.

13

The elaboration of a `protected_definition` creates the protected type and its first subtype; it also includes the elaboration of the `component_declarations` and `protected_operation_declarations` in the given order.

14

As part of the initialization of a protected object, any `per-object` constraints (see Section 4.8 [3.8], page 130) are elaborated.

15

The elaboration of a `protected_body` has no other effect than to establish that protected operations of the type can from then on be called without failing the `Elaboration_Check`.

16

The content of an object of a given protected type includes:

17

- The values of the components of the protected object, including (implicitly) an entry queue for each entry declared for the protected object;

18

- A representation of the state of the execution resource `<associated>` with the protected object (one such resource is associated with each protected object).

19

The execution resource associated with a protected object has to be acquired to read or update any components of the protected object; it can be acquired (as part of a protected action — see Section 10.5.1 [9.5.1], page 344) either for concurrent read-only access, or for exclusive read-write access.

20

As the first step of the `<finalization>` of a protected object, each call remaining on any entry queue of the object is removed from its queue and `Program_Error` is raised at the place of the corresponding `entry_call_statement` (see [S0207], page 352).

Bounded (Run-Time) Errors

20.1/2

It is a bounded error to call an entry or subprogram of a protected object after that object is finalized. If the error is detected, `Program_Error` is raised. Otherwise, the call proceeds normally, which may leave a task queued forever.

NOTES

21/2

13 Within the declaration or body of a protected unit other than in an `access_definition`, the name of the protected unit denotes the current instance of the unit (see Section 9.6 [8.6], page 324), rather than the first subtype of the corresponding protected type (and thus the name cannot be used as a `subtype_mark`).

22

14 A `selected_component` can be used to denote a discriminant of a protected object (see Section 5.1.3 [4.1.3], page 183). Within a protected unit, the name of a discriminant of the protected type denotes the corresponding discriminant of the current instance of the unit.

23/2

15 A protected type is a limited type (see Section 8.5 [7.5], page 292), and hence precludes use of assignment_statements and predefined equality operators.

24

16 The bodies of the protected operations given in the protected_body define the actions that take place upon calls to the protected operations.

25

17 The declarations in the private part are only visible within the private part and the body of the protected unit.

Examples

26

<Example of declaration of protected type and corresponding body:>

27

```
protected type Resource is
  entry Seize;
  procedure Release;
private
  Busy : Boolean := False;
end Resource;
```

28

```
protected body Resource is
  entry Seize when not Busy is
  begin
    Busy := True;
  end Seize;
```

29

```
procedure Release is
begin
  Busy := False;
end Release;
end Resource;
```

30

<Example of a single protected declaration and corresponding body:>

31

```
protected Shared_Array is
  --< Index, Item, and Item_Array are global types>
  function Component (N : in Index) return Item;
```

```

        procedure Set_Component(N : in Index; E : in Item);
private
    Table : Item_Array(Index) := (others => Null_Item);
end Shared_Array;

```

32

```

protected body Shared_Array is
    function Component(N : in Index) return Item is
begin
    return Table(N);
end Component;

```

33

```

        procedure Set_Component(N : in Index; E : in Item) is
begin
    Table(N) := E;
end Set_Component;
end Shared_Array;

```

34

<Examples of protected objects:>

35

```

Control : Resource;
Flags   : array(1 .. 100) of Resource;

```

10.5 9.5 Intertask Communication

1

The primary means for intertask communication is provided by calls on entries and protected subprograms. Calls on protected subprograms allow coordinated access to shared data objects. Entry calls allow for blocking the caller until a given condition is satisfied (namely, that the corresponding entry is open — see Section 10.5.3 [9.5.3], page 352), and then communicating data or control information directly with another task or indirectly via a shared protected object.

Static Semantics

2

Any call on an entry or on a protected subprogram identifies a <target object> for the operation, which is either a task (for an entry call) or a protected object (for an entry call or a protected subprogram call). The target object is considered an implicit parameter to the operation, and is determined by the operation name (or prefix) used in the call on the operation, as follows:

3

- If it is a `direct_name` or expanded name that denotes the declaration (or body) of the operation, then the target object is implicitly specified to be the current instance of

the task or protected unit immediately enclosing the operation; such a call is defined to be an <internal call>;

4

- If it is a `selected_component` that is not an expanded name, then the target object is explicitly specified to be the task or protected object denoted by the prefix of the name; such a call is defined to be an <external call>;

5

- If the name or prefix is a dereference (implicit or explicit) of an `access-to-protected-subprogram` value, then the target object is determined by the prefix of the `Access` attribute_reference that produced the access value originally, and the call is defined to be an <external call>;

6

- If the name or prefix denotes a `subprogram-renaming-declaration`, then the target object is as determined by the name of the renamed entity.

7

A corresponding definition of target object applies to a `requeue_statement` (see Section 10.5.4 [9.5.4], page 356), with a corresponding distinction between an <internal requeue> and an <external requeue>.

Legality Rules

7.1/2

The view of the target protected object associated with a call of a protected procedure or entry shall be a variable.

Dynamic Semantics

8

Within the body of a protected operation, the current instance (see Section 9.6 [8.6], page 324) of the immediately enclosing protected unit is determined by the target object specified (implicitly or explicitly) in the call (or requeue) on the protected operation.

9

Any call on a protected procedure or entry of a target protected object is defined to be an update to the object, as is a requeue on such an entry.

10.5.1 9.5.1 Protected Subprograms and Protected Actions

1

A <protected subprogram> is a subprogram declared immediately within a `protected_definition`. Protected procedures provide exclusive read–write access to the data of a protected object; protected functions provide concurrent read–only access to the data.

Static Semantics

2

Within the body of a protected function (or a function declared immediately within a `protected_body`), the current instance of the enclosing protected unit is defined to be a

constant (that is, its subcomponents may be read but not updated). Within the body of a protected procedure (or a procedure declared immediately within a `protected_body`), and within an `entry_body`, the current instance is defined to be a variable (updating is permitted).

Dynamic Semantics

3

For the execution of a call on a protected subprogram, the evaluation of the name or prefix and of the parameter associations, and any assigning back of in out or out parameters, proceeds as for a normal subprogram call (see Section 7.4 [6.4], page 266). If the call is an internal call (see Section 10.5 [9.5], page 343), the body of the subprogram is executed as for a normal subprogram call. If the call is an external call, then the body of the subprogram is executed as part of a new `<protected action>` on the target protected object; the protected action completes after the body of the subprogram is executed. A protected action can also be started by an entry call (see Section 10.5.3 [9.5.3], page 352).

4

A new protected action is not started on a protected object while another protected action on the same protected object is underway, unless both actions are the result of a call on a protected function. This rule is expressible in terms of the execution resource associated with the protected object:

5

- `<Starting>` a protected action on a protected object corresponds to `<acquiring>` the execution resource associated with the protected object, either for concurrent read-only access if the protected action is for a call on a protected function, or for exclusive read-write access otherwise;

6

- `<Completing>` the protected action corresponds to `<releasing>` the associated execution resource.

7

After performing an operation on a protected object other than a call on a protected function, but prior to completing the associated protected action, the entry queues (if any) of the protected object are serviced (see Section 10.5.3 [9.5.3], page 352).

Bounded (Run-Time) Errors

8

During a protected action, it is a bounded error to invoke an operation that is `<potentially blocking>`. The following are defined to be potentially blocking operations:

9

- a `select_statement`;

10

- an `accept_statement`;

11

- an `entry_call_statement`;

12

- a `delay_statement`;

13

- an `abort_statement`;

14

- task creation or activation;

15

- an external call on a protected subprogram (or an external requeue) with the same target object as that of the protected action;

16

- a call on a subprogram whose body contains a potentially blocking operation.

17

If the bounded error is detected, `Program_Error` is raised. If not detected, the bounded error might result in deadlock or a (nested) protected action on the same target object.

18

Certain language-defined subprograms are potentially blocking. In particular, the subprograms of the language-defined input-output packages that manipulate files (implicitly or explicitly) are potentially blocking. Other potentially blocking subprograms are identified where they are defined. When not specified as potentially blocking, a language-defined subprogram is nonblocking.

NOTES

19

18 If two tasks both try to start a protected action on a protected object, and at most one is calling a protected function, then only one of the tasks can proceed. Although the other task cannot proceed, it is not considered blocked, and it might be consuming processing resources while it awaits its turn. There is no language-defined ordering or queuing presumed for tasks competing to start a protected action -- on a multiprocessor such tasks might use busy-waiting; for monoprocessor considerations, see Section 18.3 [D.3], page 991, "Section 18.3 [D.3], page 991, Priority Ceiling Locking".

20

19 The body of a protected unit may contain declarations and bodies for local subprograms. These are not visible outside the protected unit.

21

20 The body of a protected function can contain internal calls on other protected functions, but not protected procedures, because the current instance is a constant. On the other hand, the body of a protected procedure can contain internal calls on both protected functions and procedures.

22

21 From within a protected action, an internal call on a protected subprogram, or an external call on a protected subprogram with a different target object is not considered a potentially blocking operation.

22.1/2

22 The pragma `Detect_Blocking` may be used to ensure that all executions of potentially blocking operations during a protected action raise `Program_Error`. See Section 22.5 [H.5], page 1163.

Examples

23

<Examples of protected subprogram calls (see Section 10.4 [9.4], page 337):>

24

```
Shared_Array.Set_Component(N, E);  
E := Shared_Array.Component(M);  
Control.Release;
```

10.5.2 9.5.2 Entries and Accept Statements

1

Entry_declarations, with the corresponding entry_bodies or accept_statements, are used to define potentially queued operations on tasks and protected objects.

Syntax

2/2

```
entry_declaration ::=  
  [overriding_indicator]  
  entry_defining_identifier [(discrete_subtype_definition)] parameter_profile;
```

3

```
accept_statement ::=  
  accept <entry_>direct_name [(entry_index)] parameter_profile [do
```

```
    handled_sequence_of_statements
end [<entry_>identifier];
```

4

```
entry_index ::= expression
```

5

```
entry_body ::=
    entry_defining_identifier entry_body_formal_part entry_barrier is
    declarative_part
begin
    handled_sequence_of_statements
end [<entry_>identifier];
```

6

```
entry_body_formal_part ::= [(entry_index_specification)] parameter_profile
```

7

```
entry_barrier ::= when condition
```

8

```
entry_index_specification ::= for defining_identifier in discrete_subtype_definition
```

9

If an <entry_>identifier appears at the end of an accept_statement, it shall repeat the <entry_>direct_name (see [S0092], page 179). If an <entry_>identifier appears at the end of an entry_body (see [S0203], page 348), it shall repeat the defining_identifier (see [S0022], page 49).

10

An entry_declaration is allowed only in a protected or task declaration.

10.1/2

An overriding_indicator is not allowed in an entry_declaration that includes a discrete_subtype_definition.

Name Resolution Rules

11

In an accept_statement, the expected profile for the <entry_>direct_name is that of the entry_declaration (see [S0200], page 347); the expected type for an entry_index is that of the subtype defined by the discrete_subtype_definition (see [S0055], page 114) of the corresponding entry_declaration (see [S0200], page 347).

12

Within the `handled_sequence_of_statements` of an `accept_statement`, if a `selected_component` (see [S0098], page 183) has a prefix that denotes the corresponding `entry_declaration` (see [S0200], page 347), then the entity denoted by the prefix is the `accept_statement` (see [S0201], page 347), and the `selected_component` (see [S0098], page 183) is interpreted as an expanded name (see Section 5.1.3 [4.1.3], page 183); the `selector_name` of the `selected_component` (see [S0098], page 183) has to be the identifier for some formal parameter of the `accept_statement` (see [S0201], page 347).

Legality Rules

13

An `entry_declaration` in a task declaration shall not contain a specification for an access parameter (see Section 4.10 [3.10], page 156).

13.1/2

If an `entry_declaration` has an `overriding_indicator`, then at the point of the declaration:

13.2/2

- if the `overriding_indicator` is overriding, then the entry shall implement an inherited subprogram;

13.3/2

- if the `overriding_indicator` is not overriding, then the entry shall not implement any inherited subprogram.

13.4/2

In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), these rules also apply in the private part of an instance of a generic unit.

14

For an `accept_statement`, the innermost enclosing body shall be a `task_body`, and the `<entry->direct_name` (see [S0092], page 179) shall denote an `entry_declaration` (see [S0200], page 347) in the corresponding task declaration; the profile of the `accept_statement` (see [S0201], page 347) shall conform fully to that of the corresponding `entry_declaration` (see [S0200], page 347). An `accept_statement` (see [S0201], page 347) shall have a parenthesized `entry_index` (see [S0202], page 348) if and only if the corresponding `entry_declaration` (see [S0200], page 347) has a `discrete_subtype_definition` (see [S0055], page 114).

15

An `accept_statement` shall not be within another `accept_statement` that corresponds to the same `entry_declaration` (see [S0200], page 347), nor within an `asynchronous_select` (see [S0223], page 384) inner to the enclosing `task_body`.

16

An `entry_declaration` of a protected unit requires a completion, which shall be an `entry_body`, and every `entry_body` (see [S0203], page 348) shall be the completion of an `entry_declaration` (see [S0200], page 347) of a protected unit. The profile of the `entry_body` (see [S0203], page 348) shall conform fully to that of the corresponding declaration.

17

An `entry_body_formal_part` shall have an `entry_index_specification` (see [S0206], page 348) if and only if the corresponding `entry_declaration` (see [S0200], page 347) has a `discrete_subtype_definition` (see [S0055], page 114). In this case, the `discrete_subtype_definition` (see [S0055], page 114)s of the `entry_declaration` (see [S0200], page 347) and the `entry_index_specification` (see [S0206], page 348) shall fully conform to one another (see Section 7.3.1 [6.3.1], page 263).

18

A name that denotes a formal parameter of an `entry_body` is not allowed within the `entry_barrier` of the `entry_body`.

Static Semantics

19

The parameter modes defined for parameters in the `parameter_profile` of an `entry_declaration` are the same as for a `subprogram_declaration` and have the same meaning (see Section 7.2 [6.2], page 260).

20

An `entry_declaration` with a `discrete_subtype_definition` (see Section 4.6 [3.6], page 114) declares a <family> of distinct entries having the same profile, with one such entry for each value of the <entry index subtype> defined by the `discrete_subtype_definition` (see [S0055], page 114). A name for an entry of a family takes the form of an `indexed_component`, where the prefix denotes the `entry_declaration` for the family, and the index value identifies the entry within the family. The term <single entry> is used to refer to any entry other than an entry of an entry family.

21

In the `entry_body` for an entry family, the `entry_index_specification` declares a named constant whose subtype is the entry index subtype defined by the corresponding `entry_declaration`; the value of the <named entry index> identifies which entry of the family was called.

Dynamic Semantics

22/1

The elaboration of an `entry_declaration` for an entry family consists of the elaboration of the `discrete_subtype_definition` (see [S0055], page 114), as described in Section 4.8 [3.8], page 130. The elaboration of an `entry_declaration` (see [S0200], page 347) for a single entry has no effect.

23

The actions to be performed when an entry is called are specified by the corresponding `accept_statement` (see [S0201], page 347)s (if any) for an entry of a task unit, and by the corresponding `entry_body` (see [S0203], page 348) for an entry of a protected unit.

24

For the execution of an `accept_statement`, the `entry_index`, if any, is first evaluated and converted to the entry index subtype; this index value identifies which entry of the family is to be accepted. Further execution of the `accept_statement` is then blocked until a caller of the corresponding entry is selected (see Section 10.5.3 [9.5.3], page 352), whereupon the `handled_sequence_of_statements`, if any, of the `accept_statement` is executed, with the formal parameters associated with the corresponding actual parameters of the selected entry call.

Upon completion of the `handled_sequence_of_statements`, the `accept_statement` completes and is left. When an exception is propagated from the `handled_sequence_of_statements` of an `accept_statement`, the same exception is also raised by the execution of the corresponding `entry_call_statement`.

25

The above interaction between a calling task and an accepting task is called a `<rendezvous>`. After a `rendezvous`, the two tasks continue their execution independently.

26

An `entry_body` is executed when the condition of the `entry_barrier` evaluates to `True` and a caller of the corresponding single entry, or entry of the corresponding entry family, has been selected (see Section 10.5.3 [9.5.3], page 352). For the execution of the `entry_body` (see [S0203], page 348), the `declarative_part` (see [S0086], page 175) of the `entry_body` (see [S0203], page 348) is elaborated, and the `handled_sequence_of_statements` (see [S0247], page 420) of the body is executed, as for the execution of a `subprogram_body`. The value of the named entry index, if any, is determined by the value of the entry index specified in the `<entry_>name` of the selected entry call (or intermediate `requeue_statement` (see [S0208], page 356) — see Section 10.5.4 [9.5.4], page 356).

NOTES

27

23 A task entry has corresponding `accept_statements` (zero or more), whereas a protected entry has a corresponding `entry_body` (exactly one).

28

24 A consequence of the rule regarding the allowed placements of `accept_statements` is that a task can execute `accept_statements` only for its own entries.

29/2

25 A `return_statement` (see Section 7.5 [6.5], page 272) or a `requeue_statement` (see Section 10.5.4 [9.5.4], page 356) may be used to complete the execution of an `accept_statement` or an `entry_body`.

30

26 The condition in the `entry_barrier` may reference anything visible except the formal parameters of the entry. This includes the entry index (if any), the components (including discriminants) of the protected object, the `Count` attribute of an entry of that protected object, and data global to the protected unit.

31

The restriction against referencing the formal parameters within an `entry_barrier` ensures that all calls of the same entry see the same

barrier value. If it is necessary to look at the parameters of an entry call before deciding whether to handle it, the `entry_barrier` can be "when True" and the caller can be requeued (on some private entry) when its parameters indicate that it cannot be handled immediately.

Examples

32

<Examples of entry declarations:>

33

```
entry Read(V : out Item);
entry Seize;
entry Request(Level)(D : Item);  --< a family of entries>
```

34

<Examples of accept statements:>

35

```
accept Shut_Down;
```

36

```
accept Read(V : out Item) do
  V := Local_Item;
end Read;
```

37

```
accept Request(Low)(D : Item) do
  ...
end Request;
```

10.5.3 9.5.3 Entry Calls

1

An `entry_call_statement` (an <entry call>) can appear in various contexts. A <simple> entry call is a stand-alone statement that represents an unconditional call on an entry of a target task or a protected object. Entry calls can also appear as part of `select_statements` (see Section 10.7 [9.7], page 377).

Syntax

2

```
entry_call_statement ::= <entry_>name [actual_parameter_part];
```

Name Resolution Rules

3

The <entry_>name given in an `entry_call_statement` shall resolve to denote an entry. The rules for parameter associations are the same as for subprogram calls (see Section 7.4 [6.4], page 266, and Section 7.4.1 [6.4.1], page 270).

Static Semantics

4

The `<entry_name>` of an `entry_call_statement` specifies (explicitly or implicitly) the target object of the call, the entry or entry family, and the entry index, if any (see Section 10.5 [9.5], page 343).

Dynamic Semantics

5

Under certain circumstances (detailed below), an entry of a task or protected object is checked to see whether it is `<open>` or `<closed>`:

6

- An entry of a task is open if the task is blocked on an `accept_statement` that corresponds to the entry (see Section 10.5.2 [9.5.2], page 347), or on a `selective_accept` (see Section 10.7.1 [9.7.1], page 378) with an open `accept_alternative` that corresponds to the entry; otherwise it is closed.

7

- An entry of a protected object is open if the condition of the `entry_barrier` of the corresponding `entry_body` evaluates to `True`; otherwise it is closed. If the evaluation of the condition propagates an exception, the exception `Program_Error` is propagated to all current callers of all entries of the protected object.

8

For the execution of an `entry_call_statement`, evaluation of the name and of the parameter associations is as for a subprogram call (see Section 7.4 [6.4], page 266). The entry call is then `<issued>`: For a call on an entry of a protected object, a new protected action is started on the object (see Section 10.5.1 [9.5.1], page 344). The named entry is checked to see if it is open; if open, the entry call is said to be `<selected immediately>`, and the execution of the call proceeds as follows:

9

- For a call on an open entry of a task, the accepting task becomes ready and continues the execution of the corresponding `accept_statement` (see Section 10.5.2 [9.5.2], page 347).

10

- For a call on an open entry of a protected object, the corresponding `entry_body` is executed (see Section 10.5.2 [9.5.2], page 347) as part of the protected action.

11

If the `accept_statement` or `entry_body` completes other than by a `requeue` (see Section 10.5.4 [9.5.4], page 356), return is made to the caller (after servicing the entry queues — see below); any necessary assigning back of formal to actual parameters occurs, as for a subprogram call (see Section 7.4.1 [6.4.1], page 270); such assignments take place outside of any protected action.

12

If the named entry is closed, the entry call is added to an `<entry queue>` (as part of the

protected action, for a call on a protected entry), and the call remains queued until it is selected or cancelled; there is a separate (logical) entry queue for each entry of a given task or protected object (including each entry of an entry family).

13

When a queued call is <selected>, it is removed from its entry queue. Selecting a queued call from a particular entry queue is called <servicing> the entry queue. An entry with queued calls can be serviced under the following circumstances:

14

- When the associated task reaches a corresponding `accept_statement`, or a `selective_accept` with a corresponding open `accept_alternative`;

15

- If after performing, as part of a protected action on the associated protected object, an operation on the object other than a call on a protected function, the entry is checked and found to be open.

16

If there is at least one call on a queue corresponding to an open entry, then one such call is selected according to the <entry queuing policy> in effect (see below), and the corresponding `accept_statement` or `entry_body` is executed as above for an entry call that is selected immediately.

17

The entry queuing policy controls selection among queued calls both for task and protected entry queues. The default entry queuing policy is to select calls on a given entry queue in order of arrival. If calls from two or more queues are simultaneously eligible for selection, the default entry queuing policy does not specify which queue is serviced first. Other entry queuing policies can be specified by pragmas (see Section 18.4 [D.4], page 994).

18

For a protected object, the above servicing of entry queues continues until there are no open entries with queued calls, at which point the protected action completes.

19

For an entry call that is added to a queue, and that is not the `triggering_statement` of an `asynchronous_select` (see [S0223], page 384) (see Section 10.7.4 [9.7.4], page 383), the calling task is blocked until the call is cancelled, or the call is selected and a corresponding `accept_statement` or `entry_body` completes without requeuing. In addition, the calling task is blocked during a rendezvous.

20

An attempt can be made to cancel an entry call upon an abort (see Section 10.8 [9.8], page 385) and as part of certain forms of `select_statement` (see Section 10.7.2 [9.7.2], page 381, Section 10.7.3 [9.7.3], page 382, and Section 10.7.4 [9.7.4], page 383). The cancellation does not take place until a point (if any) when the call is on some entry queue, and not protected from cancellation as part of a requeue (see Section 10.5.4 [9.5.4], page 356); at such a point, the call is removed from the entry queue and the call completes due to the cancellation. The cancellation of a call on an entry of a protected object is a protected

action, and as such cannot take place while any other protected action is occurring on the protected object. Like any protected action, it includes servicing of the entry queues (in case some entry barrier depends on a Count attribute).

21

A call on an entry of a task that has already completed its execution raises the exception `Tasking_Error` at the point of the call; similarly, this exception is raised at the point of the call if the called task completes its execution or becomes abnormal before accepting the call or completing the rendezvous (see Section 10.8 [9.8], page 385). This applies equally to a simple entry call and to an entry call as part of a `select_statement`.

Implementation Permissions

22

An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action. If an `entry_body` completes without requeuing, then the corresponding calling task may be made ready without waiting for the entire protected action to complete.

23

When the entry of a protected object is checked to see whether it is open, the implementation need not reevaluate the condition of the corresponding `entry_barrier` if no variable or attribute referenced by the condition (directly or indirectly) has been altered by the execution (or cancellation) of a protected procedure or entry call on the object since the condition was last evaluated.

24

An implementation may evaluate the conditions of all `entry_barriers` of a given protected object any time any entry of the object is checked to see if it is open.

25

When an attempt is made to cancel an entry call, the implementation need not make the attempt using the thread of control of the task (or interrupt) that initiated the cancellation; in particular, it may use the thread of control of the caller itself to attempt the cancellation, even if this might allow the entry call to be selected in the interim.

NOTES

26

27 If an exception is raised during the execution of an `entry_body`, it is propagated to the corresponding caller (see Section 12.4 [11.4], page 422).

27

28 For a call on a protected entry, the entry is checked to see if it is open prior to queuing the call, and again thereafter if its `Count` attribute (see Section 10.9 [9.9], page 388) is referenced in some entry barrier.

28

29 In addition to simple entry calls, the language permits timed, conditional, and asynchronous entry calls (see Section 10.7.2 [9.7.2],

page 381, Section 10.7.3 [9.7.3], page 382, and see Section 10.7.4 [9.7.4], page 383).

29

30 The condition of an `entry_barrier` is allowed to be evaluated by an implementation more often than strictly necessary, even if the evaluation might have side effects. On the other hand, an implementation need not reevaluate the condition if nothing it references was updated by an intervening protected action on the protected object, even if the condition references some global variable that might have been updated by an action performed from outside of a protected action.

Examples

30

<Examples of entry calls:>

31

<code>Agent.Shut_Down;</code>	--< see Section 10.1 [9.1], page 329>
<code>Parser.Next_Lexeme(E);</code>	--< see Section 10.1 [9.1], page 329>
<code>Pool(5).Read(Next_Char);</code>	--< see Section 10.1 [9.1], page 329>
<code>Controller.Request(Low)(Some_Item);</code>	--< see Section 10.1 [9.1], page 329>
<code>Flags(3).Seize;</code>	--< see Section 10.4 [9.4], page 337>

10.5.4 9.5.4 Requeue Statements

1

A `requeue_statement` can be used to complete an `accept_statement` or `entry_body`, while redirecting the corresponding entry call to a new (or the same) entry queue. Such a <requeue> can be performed with or without allowing an intermediate cancellation of the call, due to an abort or the expiration of a delay.

Syntax

2

`requeue_statement ::= requeue <entry_>name [with abort];`

Name Resolution Rules

3

The <entry_>name of a `requeue_statement` shall resolve to denote an entry (the <target entry>) that either has no parameters, or that has a profile that is type conformant (see Section 7.3.1 [6.3.1], page 263) with the profile of the innermost enclosing `entry_body` (see [S0203], page 348) or `accept_statement` (see [S0201], page 347).

Legality Rules

4

A `requeue_statement` shall be within a callable construct that is either an `entry_body` or an `accept_statement`, and this construct shall be the innermost enclosing body or callable construct.

5

If the target entry has parameters, then its profile shall be subtype conformant with the profile of the innermost enclosing callable construct.

6

In a `requeue_statement` of an `accept_statement` of some task unit, either the target object shall be a part of a formal parameter of the `accept_statement`, or the accessibility level of the target object shall not be equal to or statically deeper than any enclosing `accept_statement` of the task unit. In a `requeue_statement` (see [S0208], page 356) of an `entry_body` (see [S0203], page 348) of some protected unit, either the target object shall be a part of a formal parameter of the `entry_body` (see [S0203], page 348), or the accessibility level of the target object shall not be statically deeper than that of the `entry_declaration`.

Dynamic Semantics

7

The execution of a `requeue_statement` proceeds by first evaluating the `<entry_>name`, including the prefix identifying the target task or protected object and the expression identifying the entry within an entry family, if any. The `entry_body` or `accept_statement` enclosing the `requeue_statement` is then completed, finalized, and left (see Section 8.6.1 [7.6.1], page 299).

8

For the execution of a `requeue` on an entry of a target task, after leaving the enclosing callable construct, the named entry is checked to see if it is open and the `requeued` call is either selected immediately or queued, as for a normal entry call (see Section 10.5.3 [9.5.3], page 352).

9

For the execution of a `requeue` on an entry of a target protected object, after leaving the enclosing callable construct:

10

- if the `requeue` is an internal `requeue` (that is, the `requeue` is back on an entry of the same protected object — see Section 10.5 [9.5], page 343), the call is added to the queue of the named entry and the ongoing protected action continues (see Section 10.5.1 [9.5.1], page 344);

11

- if the `requeue` is an external `requeue` (that is, the target protected object is not implicitly the same as the current object — see Section 10.5 [9.5], page 343), a protected action is started on the target object and proceeds as for a normal entry call (see Section 10.5.3 [9.5.3], page 352).

12

If the new entry named in the `requeue_statement` has formal parameters, then during the execution of the `accept_statement` or `entry_body` corresponding to the new entry, the formal

parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. In any case, no parameters are specified in a requeue_statement; any parameter passing is implicit.

13

If the requeue_statement includes the reserved words with abort (it is a <requeue-with-abort>), then:

14

- if the original entry call has been aborted (see Section 10.8 [9.8], page 385), then the requeue acts as an abort completion point for the call, and the call is cancelled and no requeue is performed;

15

- if the original entry call was timed (or conditional), then the original expiration time is the expiration time for the requeued call.

16

If the reserved words with abort do not appear, then the call remains protected against cancellation while queued as the result of the requeue_statement.

NOTES

17

31 A requeue is permitted from a single entry to an entry of an entry family, or vice-versa. The entry index, if any, plays no part in the subtype conformance check between the profiles of the two entries; an entry index is part of the <entry_>name for an entry of a family.

Examples

18

<Examples of requeue statements:>

19

```
requeue Request(Medium) with abort;  
--< requeue on a member of an entry family of the current task  
[9.1], page 329>
```

20

```
requeue Flags(I).Seize;  
--< requeue on an entry of an array component, see Section 10.8  
[9.4], page 337>
```

10.6 9.6 Delay Statements, Duration, and Time

1

A delay_statement is used to block further execution until a specified <expiration time> is reached. The expiration time can be specified either as a particular point in time (in a

delay_until_statement (see [S0210], page 359)), or in seconds from the current time (in a delay_relative_statement (see [S0211], page 359)). The language-defined package Calendar provides definitions for a type Time and associated operations, including a function Clock that returns the current time.

Syntax

2

```
delay_statement ::= delay_until_statement | delay_relative_statement
```

3

```
delay_until_statement ::= delay until <delay_>expression;
```

4

```
delay_relative_statement ::= delay <delay_>expression;
```

Name Resolution Rules

5

The expected type for the <delay_>expression in a delay_relative_statement is the predefined type Duration. The <delay_>expression in a delay_until_statement is expected to be of any nonlimited type.

Legality Rules

6

There can be multiple time bases, each with a corresponding clock, and a corresponding <time type>. The type of the <delay_>expression in a delay_until_statement shall be a time type -- either the type Time defined in the language-defined package Calendar (see below), or some other implementation-defined time type (see Section 18.8 [D.8], page 1008).

Static Semantics

7

There is a predefined fixed point type named Duration, declared in the visible part of package Standard; a value of type Duration is used to represent the length of an interval of time, expressed in seconds. The type Duration is not specific to a particular time base, but can be used with any time base.

8

A value of the type Time in package Calendar, or of some other implementation-defined time type, represents a time as reported by a corresponding clock.

9

The following language-defined library package exists:

10

```
package Ada.Calendar is
  type
    Time is private;
```

11/2

```
    subtype
Year_Number is Integer range 1901 .. 2399;
    subtype
Month_Number is Integer range 1 .. 12;
    subtype
Day_Number   is Integer range 1 .. 31;
    subtype
Day_Duration is Duration range 0.0 .. 86_400.0;
```

12

```
    function
Clock return Time;
```

13

```
    function
Year (Date : Time) return Year_Number;
    function
Month (Date : Time) return Month_Number;
    function
Day   (Date : Time) return Day_Number;
    function
Seconds(Date : Time) return Day_Duration;
```

14

```
    procedure
Split (Date : in Time;
      Year   : out Year_Number;
      Month  : out Month_Number;
      Day    : out Day_Number;
      Seconds : out Day_Duration);
```

15

```
    function
Time_Of (Year : Year_Number;
      Month : Month_Number;
      Day   : Day_Number;
      Seconds : Day_Duration := 0.0)
    return Time;
```

16

```
function "+" (Left : Time; Right : Duration) return Time;
function "+" (Left : Duration; Right : Time) return Time;
```



```
function "-" (Left : Time;   Right : Duration) return Time;
function "-" (Left : Time;   Right : Time) return Duration;
```

17

```
function "<" (Left, Right : Time) return Boolean;
function "<=" (Left, Right : Time) return Boolean;
function ">" (Left, Right : Time) return Boolean;
function ">=" (Left, Right : Time) return Boolean;
```

18

```
Time_Error : exception;
```

19

```
private
  ... -- <not specified by the language>
end Ada.Calendar;
```

Dynamic Semantics

20

For the execution of a `delay_statement`, the `<delay_>expression` is first evaluated. For a `delay_until_statement`, the expiration time for the delay is the value of the `<delay_>expression`, in the time base associated with the type of the expression. For a `delay_relative_statement`, the expiration time is defined as the current time, in the time base associated with relative delays, plus the value of the `<delay_>expression` converted to the type `Duration`, and then rounded up to the next clock tick. The time base associated with relative delays is as defined in Section 18.9 [D.9], page 1013, "Section 18.9 [D.9], page 1013, Delay Accuracy" or is implementation defined.

21

The task executing a `delay_statement` is blocked until the expiration time is reached, at which point it becomes ready again. If the expiration time has already passed, the task is not blocked.

22

If an attempt is made to `<cancel>` the `delay_statement` (as part of an `asynchronous_select` (see [S0223], page 384) or `abort` — see Section 10.7.4 [9.7.4], page 383, and Section 10.8 [9.8], page 385), the `_statement` is cancelled if the expiration time has not yet passed, thereby completing the `delay_statement`.

23

The time base associated with the type `Time` of package `Calendar` is implementation defined. The function `Clock` of package `Calendar` returns a value representing the current time for this time base. The implementation-defined value of the named number `System.Tick` (see Section 14.7 [13.7], page 510) is an approximation of the length of the real-time interval during which the value of `Calendar.Clock` remains constant.

24/2

The functions `Year`, `Month`, `Day`, and `Seconds` return the corresponding values for a given

value of the type `Time`, as appropriate to an implementation—defined time zone; the procedure `Split` returns all four corresponding values. Conversely, the function `Time_Of` combines a year number, a month number, a day number, and a duration, into a value of type `Time`. The operators "+" and "-" for addition and subtraction of times and durations, and the relational operators for times, have the conventional meaning.

25

If `Time_Of` is called with a seconds value of 86_400.0, the value returned is equal to the value of `Time_Of` for the next day with a seconds value of 0.0. The value returned by the function `Seconds` or through the `Seconds` parameter of the procedure `Split` is always less than 86_400.0.

26/1

The exception `Time_Error` is raised by the function `Time_Of` if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "-" if the result is not representable in the type `Time` or `Duration`, as appropriate. This exception is also raised by the functions `Year`, `Month`, `Day`, and `Seconds` and the procedure `Split` if the year number of the given date is outside of the range of the subtype `Year_Number`.

Implementation Requirements

27

The implementation of the type `Duration` shall allow representation of time intervals (both positive and negative) up to at least 86400 seconds (one day); `Duration'Small` shall not be greater than twenty milliseconds. The implementation of the type `Time` shall allow representation of all dates with year numbers in the range of `Year_Number`; it may allow representation of other dates as well (both earlier and later).

Implementation Permissions

28

An implementation may define additional time types (see Section 18.8 [D.8], page 1008).

29

An implementation may raise `Time_Error` if the value of a <delay_>expression in a `delay_until_statement` of a `select_statement` represents a time more than 90 days past the current time. The actual limit, if any, is implementation—defined.

Implementation Advice

30

Whenever possible in an implementation, the value of `Duration'Small` should be no greater than 100 microseconds.

31

The time base for `delay_relative_statements` should be monotonic; it need not be the same time base as used for `Calendar.Clock`.

NOTES

32

32 A `delay_relative_statement` with a negative value of the <delay_>expression is equivalent to one with a zero value.

33

33 A `delay_statement` may be executed by the environment task; consequently `delay_statements` may be executed as part of the elaboration of a `library_item` or the execution of the main subprogram. Such statements delay the environment task (see Section 11.2 [10.2], page 409).

34

34 A `delay_statement` is an abort completion point and a potentially blocking operation, even if the task is not actually blocked.

35

35 There is no necessary relationship between `System.Tick` (the resolution of the clock of package `Calendar`) and `Duration'Small` (the `<small>` of type `Duration`).

36

36 Additional requirements associated with `delay_statements` are given in Section 18.9 [D.9], page 1013, "Section 18.9 [D.9], page 1013, Delay Accuracy".

Examples

37

<Example of a relative delay statement:>

38

```
delay 3.0;  --< delay 3.0 seconds>
```

39

<Example of a periodic task:>

40

```
declare
  use Ada.Calendar;
  Next_Time : Time := Clock + Period;
  --< Period is a global constant of type Duration>
begin
  loop
    --< repeated every Period seconds>
    delay until Next_Time;
    ... --< perform some actions>
    Next_Time := Next_Time + Period;
  end loop;
end;
```

10.6.1 9.6.1 Formatting, Time Zones, and other operations for Time

Static Semantics

1/2

The following language-defined library packages exist:

2/2

```
package Ada.Calendar.Time_Zones is
```

3/2

```
-- <Time zone manipulation:>
```

4/2

```
type  
Time_Offset is range -28*60 .. 28*60;
```

5/2

```
Unknown_Zone_Error : exception;
```

6/2

```
function  
UTC_Time_Offset (Date : Time := Clock) return Time_Offset;
```

7/2

```
end Ada.Calendar.Time_Zones;
```

8/2

```
package Ada.Calendar.Arithmetic is
```

9/2

```
-- <Arithmetic on days:>
```

10/2

```
type  
Day_Count is range  
-366*(1+Year_Number'Last - Year_Number'First)  
..  
366*(1+Year_Number'Last - Year_Number'First);
```

11/2

```
    subtype
Leap_Seconds_Count is Integer range -2047 .. 2047;
```

12/2

```
    procedure
Difference (Left, Right : in Time;
           Days : out Day_Count;
           Seconds : out Duration;
           Leap_Seconds : out Leap_Seconds_Count);
```

13/2

```
    function "+" (Left : Time; Right : Day_Count) return Time;
    function "+" (Left : Day_Count; Right : Time) return Time;
    function "-" (Left : Time; Right : Day_Count) return Time;
    function "-" (Left, Right : Time) return Day_Count;
```

14/2

```
end Ada.Calendar.Arithmetic;
```

15/2

```
with Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting is
```

16/2

```
-- <Day of the week:>
```

17/2

```
    type
Day_Name is (
Monday,
Tuesday,
Wednesday,
Thursday,

Friday,
Saturday,
Sunday);
```

18/2

```

function
Day_of_Week (Date : Time) return Day_Name;
19/2

-- <Hours:Minutes:Seconds access:>
20/2

subtype
Hour_Number      is Natural range 0 .. 23;
subtype
Minute_Number    is Natural range 0 .. 59;
subtype
Second_Number    is Natural range 0 .. 59;
subtype
Second_Duration is Day_Duration range 0.0 .. 1.0;
21/2

function
Year      (Date : Time;
           Time_Zone : Time_Zones.Time_Offset := 0)
return Year_Number;
22/2

function
Month     (Date : Time;
           Time_Zone : Time_Zones.Time_Offset := 0)
return Month_Number;
23/2

function
Day       (Date : Time;
           Time_Zone : Time_Zones.Time_Offset := 0)
return Day_Number;
24/2

function
Hour      (Date : Time;
           Time_Zone : Time_Zones.Time_Offset := 0)
return Hour_Number;
25/2

function
Minute    (Date : Time;
           Time_Zone : Time_Zones.Time_Offset := 0)

```

```

                return Minute_Number;
26/2

    function
    Second      (Date : Time)
                return Second_Number;
27/2

    function
    Sub_Second (Date : Time)
                return Second_Duration;
28/2

    function
    Seconds_Of (Hour   : Hour_Number;
                Minute : Minute_Number;
                Second : Second_Number := 0;
                Sub_Second : Second_Duration := 0.0)
                return Day_Duration;
29/2

    procedure
    Split (Seconds      : in Day_Duration;
          Hour         : out Hour_Number;
          Minute       : out Minute_Number;
          Second       : out Second_Number;
          Sub_Second   : out Second_Duration);
30/2

    function
    Time_Of (Year      : Year_Number;
            Month      : Month_Number;
            Day        : Day_Number;
            Hour       : Hour_Number;
            Minute     : Minute_Number;
            Second     : Second_Number;
            Sub_Second : Second_Duration := 0.0;
            Leap_Second: Boolean := False;
            Time_Zone  : Time_Zones.Time_Offset := 0)
            return Time;
31/2

    function
    Time_Of (Year      : Year_Number;

```

```

Month      : Month_Number;
Day        : Day_Number;
Seconds    : Day_Duration := 0.0;
Leap_Second: Boolean := False;
Time_Zone  : Time_Zones.Time_Offset := 0)
    return Time;

```

32/2

```

    procedure
Split (Date      : in Time;
Year           : out Year_Number;
Month          : out Month_Number;
Day            : out Day_Number;
Hour           : out Hour_Number;
Minute         : out Minute_Number;
Second         : out Second_Number;
Sub_Second    : out Second_Duration;
Time_Zone     : in Time_Zones.Time_Offset := 0);

```

33/2

```

    procedure
Split (Date      : in Time;
Year           : out Year_Number;
Month          : out Month_Number;
Day            : out Day_Number;
Hour           : out Hour_Number;
Minute         : out Minute_Number;
Second         : out Second_Number;
Sub_Second    : out Second_Duration;
Leap_Second   : out Boolean;
Time_Zone     : in Time_Zones.Time_Offset := 0);

```

34/2

```

    procedure
Split (Date      : in Time;
Year           : out Year_Number;
Month          : out Month_Number;
Day            : out Day_Number;
Seconds        : out Day_Duration;
Leap_Second   : out Boolean;
Time_Zone     : in Time_Zones.Time_Offset := 0);

```

35/2

```

-- <Simple image and value:>
function

```



```
Image (Date : Time;
      Include_Time_Fraction : Boolean := False;
      Time_Zone : Time_Zones.Time_Offset := 0) return String;■
```

36/2

```
function
Value (Date : String;
      Time_Zone : Time_Zones.Time_Offset := 0) return Time;■
```

37/2

```
function
Image (Elapsed_Time : Duration;
      Include_Time_Fraction : Boolean := False) return String;■
```

38/2

```
function
Value (Elapsed_Time : String) return Duration;
```

39/2

```
end Ada.Calendar.Formatting;
```

40/2

Type `Time_Offset` represents the number of minutes difference between the implementation-defined time zone used by `Calendar` and another time zone.

41/2

```
function UTC_Time_Offset (Date : Time := Clock) return Time_Offset;
```

42/2

Returns, as a number of minutes, the difference between the implementation-defined time zone of `Calendar`, and UTC time, at the time `Date`. If the time zone of the `Calendar` implementation is unknown, then `Unknown_Zone_Error` is raised.

43/2

```
procedure Difference (Left, Right : in Time;
                    Days : out Day_Count;
                    Seconds : out Duration;
                    Leap_Seconds : out Leap_Seconds_Count);
```

44/2

Returns the difference between Left and Right. Days is the number of days of difference, Seconds is the remainder seconds of difference excluding leap seconds, and Leap_Seconds is the number of leap seconds. If Left < Right, then Seconds <= 0.0, Days <= 0, and Leap_Seconds <= 0. Otherwise, all values are nonnegative. The absolute value of Seconds is always less than 86_400.0. For the returned values, if Days = 0, then Seconds + Duration(Leap_Seconds) = Calendar."-" (Left, Right).

45/2

```
function "+" (Left : Time; Right : Day_Count) return Time;  
function "+" (Left : Day_Count; Right : Time) return Time;
```

46/2

Adds a number of days to a time value. Time_Error is raised if the result is not representable as a value of type Time.

47/2

```
function "-" (Left : Time; Right : Day_Count) return Time;
```

48/2

Subtracts a number of days from a time value. Time_Error is raised if the result is not representable as a value of type Time.

49/2

```
function "-" (Left, Right : Time) return Day_Count;
```

50/2

Subtracts two time values, and returns the number of days between them. This is the same value that Difference would return in Days.

51/2

```
function Day_of_Week (Date : Time) return Day_Name;
```

52/2

Returns the day of the week for Time. This is based on the Year, Month, and Day values of Time.

53/2

```
function Year      (Date : Time;  
                  Time_Zone : Time_Zones.Time_Offset := 0)  
    return Year_Number;
```

54/2

Returns the year for Date, as appropriate for the specified time zone offset.

55/2

```
function Month    (Date : Time;  
                  Time_Zone : Time_Zones.Time_Offset := 0)  
    return Month_Number;
```

56/2

Returns the month for Date, as appropriate for the specified time zone offset.

57/2

```
function Day      (Date : Time;  
                  Time_Zone : Time_Zones.Time_Offset := 0)  
    return Day_Number;
```

58/2

Returns the day number for Date, as appropriate for the specified time zone offset.

59/2

```
function Hour     (Date : Time;  
                  Time_Zone : Time_Zones.Time_Offset := 0)  
    return Hour_Number;
```

60/2

Returns the hour for Date, as appropriate for the specified time zone offset.

61/2

```
function Minute   (Date : Time;  
                  Time_Zone : Time_Zones.Time_Offset := 0)
```

```
return Minute_Number;
```

62/2

Returns the minute within the hour for Date, as appropriate for the specified time zone offset.

63/2

```
function Second (Date : Time)
    return Second_Number;
```

64/2

Returns the second within the hour and minute for Date.

65/2

```
function Sub_Second (Date : Time)
    return Second_Duration;
```

66/2

Returns the fraction of second for Date (this has the same accuracy as Day_Duration). The value returned is always less than 1.0.

67/2

```
function Seconds_Of (Hour   : Hour_Number;
                    Minute  : Minute_Number;
                    Second  : Second_Number := 0;
                    Sub_Second : Second_Duration := 0.0)
    return Day_Duration;
```

68/2

Returns a Day_Duration value for the combination of the given Hour, Minute, Second, and Sub_Second. This value can be used in Calendar.Time_Of as well as the argument to Calendar."+" and Calendar."-". If Seconds_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Seconds_Of for the next second with a Sub_Second value of 0.0.

69/2

```
procedure Split (Seconds : in Day_Duration;
```

```

Hour      : out Hour_Number;
Minute    : out Minute_Number;
Second    : out Second_Number;
Sub_Second : out Second_Duration);

```

70/2

Splits Seconds into Hour, Minute, Second and Sub_Second in such a way that the resulting values all belong to their respective subtypes. The value returned in the Sub_Second parameter is always less than 1.0.

71/2

```

function Time_Of (Year      : Year_Number;
                 Month     : Month_Number;
                 Day       : Day_Number;
                 Hour      : Hour_Number;
                 Minute    : Minute_Number;
                 Second     : Second_Number;
                 Sub_Second : Second_Duration := 0.0;
                 Leap_Second: Boolean := False;
                 Time_Zone  : Time_Zones.Time_Offset := 0)
    return Time;

```

72/2

If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Time_Of for the next second with a Sub_Second value of 0.0.

73/2

```

function Time_Of (Year      : Year_Number;
                 Month     : Month_Number;
                 Day       : Day_Number;
                 Seconds    : Day_Duration := 0.0;
                 Leap_Second: Boolean := False;
                 Time_Zone  : Time_Zones.Time_Offset := 0)

```

```
return Time;
```

74/2

If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a Seconds value of 0.0.

75/2

```
procedure Split (Date      : in Time;
                Year       : out Year_Number;
                Month      : out Month_Number;
                Day        : out Day_Number;
                Hour       : out Hour_Number;
                Minute     : out Minute_Number;
                Second     : out Second_Number;
                Sub_Second : out Second_Duration;
                Leap_Second: out Boolean;
                Time_Zone  : in Time_Zones.Time_Offset := 0);
```

76/2

If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Sub_Second parameter is always less than 1.0.

77/2

```
procedure Split (Date      : in Time;
```

```

Year      : out Year_Number;
Month     : out Month_Number;
Day       : out Day_Number;
Hour      : out Hour_Number;
Minute    : out Minute_Number;
Second    : out Second_Number;
Sub_Second : out Second_Duration;
Time_Zone : in Time_Zones.Time_Offset := 0);

```

78/2

Splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset. The value returned in the Sub_Second parameter is always less than 1.0.

79/2

```

procedure Split (Date      : in Time;
Year           : out Year_Number;
Month          : out Month_Number;
Day           : out Day_Number;
Seconds       : out Day_Duration;
Leap_Second   : out Boolean;
Time_Zone     : in Time_Zones.Time_Offset := 0);

```

80/2

If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Seconds), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Seconds parameter is always less than 86_400.0.

81/2

```

function Image (Date : Time;
Include_Time_Fraction : Boolean := False;
Time_Zone : Time_Zones.Time_Offset := 0) return String;

```

82/2

Returns a string form of the Date relative to the given Time_Zone. The format is "Year-Month-Day Hour:Minute:Second", where the Year is a 4-digit value, and all others are 2-digit values, of the functions defined in Calendar and Calendar.Formatting, including a leading zero, if needed. The separators between the values are a minus, another minus, a colon, and a single space between the Day and Hour. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value.

83/2

```
function Value (Date : String;  
               Time_Zone : Time_Zones.Time_Offset := 0) return Time;■
```

84/2

Returns a Time value for the image given as Date, relative to the given time zone. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Time value.

85/2

```
function Image (Elapsed_Time : Duration;  
               Include_Time_Fraction : Boolean := False) return String;■
```

86/2

Returns a string form of the Elapsed_Time. The format is "Hour:Minute:Second", where all values are 2-digit values, including a leading zero, if needed. The separators between the values are colons. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value. If Elapsed_Time < 0.0, the result is Image (abs Elapsed_Time, Include_Time_Fraction) prefixed with a minus sign. If abs

Elapsed_Time represents 100 hours or more,
the result is implementation–defined.

87/2

```
function Value (Elapsed_Time : String) return Duration;
```

88/2

Returns a Duration value for the image given
as Elapsed_Time. Constraint_Error is raised
if the string is not formatted as described for
Image, or the function cannot interpret the
given string as a Duration value.

Implementation Advice

89/2

An implementation should support leap seconds if the target system supports them. If leap seconds are not supported, Difference should return zero for Leap_Seconds, Split should return False for Leap_Second, and Time_Of should raise Time_Error if Leap_Second is True.

NOTES

90/2

37 The implementation–defined time zone of package Calendar may, but need not, be the local time zone. UTC_Time_Offset always returns the difference relative to the implementation–defined time zone of package Calendar. If UTC_Time_Offset does not raise Unknown_Zone_Error, UTC time can be safely calculated (within the accuracy of the underlying time–base).

91/2

38 Calling Split on the results of subtracting Duration(UTC_Time_Offset*60) from Clock provides the components (hours, minutes, and so on) of the UTC time. In the United States, for example, UTC_Time_Offset will generally be negative.

10.7 9.7 Select Statements

1

There are four forms of the select_statement. One form provides a selective wait for one or more select_alternatives. Two provide timed and conditional entry calls. The fourth provides asynchronous transfer of control.

Syntax

2

```
select_statement ::=
    selective_accept
```

```
| timed_entry_call
| conditional_entry_call
| asynchronous_select
```

Examples

3

<Example of a select statement:>

4

```
select
  accept Driver_Awake_Signal;
or
  delay 30.0*Seconds;
  Stop_The_Train;
end select;
```

10.7.1 9.7.1 Selective Accept

1

This form of the select_statement allows a combination of waiting for, and selecting from, one or more alternatives. The selection may depend on conditions associated with each alternative of the selective_accept.

Syntax

2

```
selective_accept ::=
  select
    [guard]
    select_alternative
  { or
    [guard]
    select_alternative }
  [ else
    sequence_of_statements ]
  end select;
```

3

```
guard ::= when condition =>
```

4

```
select_alternative ::=
  accept_alternative
  | delay_alternative
  | terminate_alternative
```

5

accept_alternative ::=
 accept_statement [sequence_of_statements]

6

delay_alternative ::=
 delay_statement [sequence_of_statements]

7

terminate_alternative ::= terminate;

8

A selective_accept shall contain at least one accept_alternative. In addition, it can contain:

9

- a terminate_alternative (only one); or
- one or more delay_alternatives; or
- an <else part> (the reserved word else followed by a sequence_of_statements).

10

11

12

These three possibilities are mutually exclusive.

Legality Rules

13

If a selective_accept contains more than one delay_alternative, then all shall be delay_relative_statement (see [S0211], page 359)s, or all shall be delay_until_statement (see [S0210], page 359)s for the same time type.

Dynamic Semantics

14

A select_alternative is said to be <open> if it is not immediately preceded by a guard, or if the condition of its guard evaluates to True. It is said to be <closed> otherwise.

15

For the execution of a selective_accept, any guard conditions are evaluated; open alternatives are thus determined. For an open delay_alternative, the <delay->expression is also evaluated. Similarly, for an open accept_alternative for an entry of a family, the entry_index is also evaluated. These evaluations are performed in an arbitrary order, except that a <delay->expression or entry_index is not evaluated until after evaluating the corresponding condition, if any. Selection and execution of one open alternative, or of the else part, then

completes the execution of the `selective_accept`; the rules for this selection are described below.

16

Open `accept_alternatives` are first considered. Selection of one such alternative takes place immediately if the corresponding entry already has queued calls. If several alternatives can thus be selected, one of them is selected according to the entry queuing policy in effect (see Section 10.5.3 [9.5.3], page 352, and Section 18.4 [D.4], page 994). When such an alternative is selected, the selected call is removed from its entry queue and the `handled_sequence_of_statements` (see [S0247], page 420) (if any) of the corresponding `accept_statement` is executed; after the rendezvous completes any subsequent `sequence_of_statements` (see [S0130], page 240) of the alternative is executed. If no selection is immediately possible (in the above sense) and there is no else part, the task blocks until an open alternative can be selected.

17

Selection of the other forms of alternative or of an else part is performed as follows:

18

- An open `delay_alternative` is selected when its expiration time is reached if no `accept_alternative` (see [S0216], page 379) or other `delay_alternative` (see [S0217], page 379) can be selected prior to the expiration time. If several `delay_alternative` (see [S0217], page 379)s have this same expiration time, one of them is selected according to the queuing policy in effect (see Section 18.4 [D.4], page 994); the default queuing policy chooses arbitrarily among the `delay_alternative` (see [S0217], page 379)s whose expiration time has passed.

19

- The else part is selected and its `sequence_of_statements` (see [S0130], page 240) is executed if no `accept_alternative` can immediately be selected; in particular, if all alternatives are closed.

20

- An open `terminate_alternative` is selected if the conditions stated at the end of clause Section 10.3 [9.3], page 335, are satisfied.

21

The exception `Program_Error` is raised if all alternatives are closed and there is no else part.

NOTES

22

39 A `selective_accept` is allowed to have several open `delay_alternatives`. A `selective_accept` is allowed to have several open `accept_alternatives` for the same entry.

Examples

23

<Example of a task body with a selective accept:>

```

task body Server is
  Current_Work_Item : Work_Item;
begin
  loop
    select
      accept Next_Work_Item(WI : in Work_Item) do
        Current_Work_Item := WI;
      end;
      Process_Work_Item(Current_Work_Item);
    or
      accept Shut_Down;
      exit;          --< Premature shut down requested>
    or
      terminate;   --< Normal shutdown at end of scope>
    end select;
  end loop;
end Server;

```

10.7.2 9.7.2 Timed Entry Calls

1/2

A `timed_entry_call` issues an entry call that is cancelled if the call (or a `requeue-with-abort` of the call) is not selected before the expiration time is reached. A procedure call may appear rather than an entry call for cases where the procedure might be implemented by an entry.

Syntax

2

```

timed_entry_call ::=
  select
    entry_call_alternative
  or
    delay_alternative
  end select;

```

3/2

```

entry_call_alternative ::=
  procedure_or_entry_call [sequence_of_statements]

```

3.1/2

```

procedure_or_entry_call ::=
  procedure_call_statement | entry_call_statement

```

Legality Rules

3.2/2

If a `procedure_call_statement` is used for a `procedure_or_entry_call`, the `<procedure_>name` or `<procedure_>prefix` of the `procedure_call_statement` shall statically denote an entry re-named as a procedure or (a view of) a primitive subprogram of a limited interface whose first parameter is a controlling parameter (see Section 4.9.2 [3.9.2], page 145).

Static Semantics

3.3/2

If a `procedure_call_statement` is used for a `procedure_or_entry_call`, and the procedure is implemented by an entry, then the `<procedure_>name`, or `<procedure_>prefix` and possibly the first parameter of the `procedure_call_statement`, determine the target object of the call and the entry to be called.

Dynamic Semantics

4/2

For the execution of a `timed_entry_call`, the `<entry_>name`, `<procedure_>name`, or `<procedure_>prefix`, and any actual parameters are evaluated, as for a simple entry call (see Section 10.5.3 [9.5.3], page 352) or procedure call (see Section 7.4 [6.4], page 266). The expiration time (see Section 10.6 [9.6], page 358) for the call is determined by evaluating the `<delay_>expression` of the `delay_alternative`. If the call is an entry call or a call on a procedure implemented by an entry, the entry call is then issued. Otherwise, the call proceeds as described in Section 7.4 [6.4], page 266, for a procedure call, followed by the `sequence_of_statements` (see [S0130], page 240) of the `entry_call_alternative` (see [S0220], page 381); the `sequence_of_statements` (see [S0130], page 240) of the `delay_alternative` (see [S0217], page 379) is ignored.

5

If the call is queued (including due to a `requeue-with-abort`), and not selected before the expiration time is reached, an attempt to cancel the call is made. If the call completes due to the cancellation, the optional `sequence_of_statements` (see [S0130], page 240) of the `delay_alternative` (see [S0217], page 379) is executed; if the entry call completes normally, the optional `sequence_of_statements` (see [S0130], page 240) of the `entry_call_alternative` (see [S0220], page 381) is executed.

Examples

6

<Example of a timed entry call:>

7

```
select
    Controller.Request(Medium)(Some_Item);
or
    delay 45.0;
    --< controller too busy, try something else>
end select;
```

10.7.3 9.7.3 Conditional Entry Calls

1/2

A `conditional_entry_call` issues an entry call that is then cancelled if it is not selected immediately (or if a `requeue-with-abort` of the call is not selected immediately). A procedure

call may appear rather than an entry call for cases where the procedure might be implemented by an entry.

Syntax

2

```
conditional_entry_call ::=
  select
    entry_call_alternative
  else
    sequence_of_statements
  end select;
```

Dynamic Semantics

3

The execution of a `conditional_entry_call` is defined to be equivalent to the execution of a `timed_entry_call` (see [S0219], page 381) with a `delay_alternative` (see [S0217], page 379) specifying an immediate expiration time and the same `sequence_of_statements` (see [S0130], page 240) as given after the reserved word `else`.

NOTES

4

40 A `conditional_entry_call` may briefly increase the `Count` attribute of the entry, even if the conditional call is not selected.

Examples

5

<Example of a conditional entry call:>

6

```
procedure Spin(R : in Resource) is
begin
  loop
    select
      R.Seize;
    return;
    else
      null; --< busy waiting>
    end select;
  end loop;
end;
```

10.7.4 9.7.4 Asynchronous Transfer of Control

1

An asynchronous `select_statement` provides asynchronous transfer of control upon completion of an entry call or the expiration of a delay.

Syntax

2

```
asynchronous_select ::=  
  select  
    triggering_alternative  
  then abort  
    abortable_part  
  end select;
```

3

```
triggering_alternative ::= triggering_statement [sequence_of_statements]
```

4/2

```
triggering_statement ::= procedure_or_entry_call | delay_statement
```

5

```
abortable_part ::= sequence_of_statements  
Dynamic Semantics
```

6/2

For the execution of an `asynchronous_select` whose `triggering_statement` (see [S0225], page 384) is a `procedure_or_entry_call`, the `<entry_>name`, `<procedure_>name`, or `<procedure_>prefix`, and actual parameters are evaluated as for a simple entry call (see Section 10.5.3 [9.5.3], page 352) or procedure call (see Section 7.4 [6.4], page 266). If the call is an entry call or a call on a procedure implemented by an entry, the entry call is issued. If the entry call is queued (or requeued-with-abort), then the `abortable_part` is executed. If the entry call is selected immediately, and never requeued-with-abort, then the `abortable_part` is never started. If the call is on a procedure that is not implemented by an entry, the call proceeds as described in Section 7.4 [6.4], page 266, followed by the `sequence_of_statements` (see [S0130], page 240) of the `triggering_alternative` (see [S0224], page 384); the `abortable_part` is never started.

7

For the execution of an `asynchronous_select` whose `triggering_statement` (see [S0225], page 384) is a `delay_statement`, the `<delay_>expression` is evaluated and the expiration time is determined, as for a normal `delay_statement`. If the expiration time has not already passed, the `abortable_part` is executed.

8

If the `abortable_part` completes and is left prior to completion of the `triggering_statement` (see [S0225], page 384), an attempt to cancel the `triggering_statement` (see [S0225], page 384) is made. If the attempt to cancel succeeds (see Section 10.5.3 [9.5.3], page 352, and Section 10.6 [9.6], page 358), the `asynchronous_select` is complete.

9

If the `triggering_statement` (see [S0225], page 384) completes other than due to cancellation, the `abortable_part` is aborted (if started but not yet completed -- see Section 10.8 [9.8], page 385). If the `triggering_statement` (see [S0225], page 384) completes normally, the

optional `sequence_of_statements` (see [S0130], page 240) of the `triggering_alternative` (see [S0224], page 384) is executed after the `abortable_part` is left.

Examples

10

<Example of a main command loop for a command interpreter:>

11

```
loop
  select
    Terminal.Wait_For_Interrupt;
    Put_Line("Interrupted");
  then abort
    -- <This will be abandoned upon terminal interrupt>
    Put_Line("-> ");
    Get_Line(Command, Last);
    Process_Command(Command(1..Last));
  end select;
end loop;
```

12

<Example of a time-limited calculation:>

13

```
select
  delay 5.0;
  Put_Line("Calculation does not converge");
then abort
  -- <This calculation should finish in 5.0 seconds;>
  -- < if not, it is assumed to diverge.>
  Horribly_Complicated_Recursive_Function(X, Y);
end select;
```

10.8 9.8 Abort of a Task - Abort of a Sequence of Statements

1

An `abort_statement` causes one or more tasks to become abnormal, thus preventing any further interaction with such tasks. The completion of the `triggering_statement` (see [S0225], page 384) of an `asynchronous_select` causes a `sequence_of_statements` (see [S0130], page 240) to be aborted.

Syntax

2

```
abort_statement ::= abort <task_>name {, <task_>name};
```

Name Resolution Rules

3

Each <task_>name is expected to be of any task type; they need not all be of the same task type.

Dynamic Semantics

4

For the execution of an abort_statement, the given <task_>names are evaluated in an arbitrary order. Each named task is then <aborted>, which consists of making the task <abnormal> and aborting the execution of the corresponding task_body, unless it is already completed.

5

When the execution of a construct is <aborted> (including that of a task_body (see [S0192], page 330) or of a sequence_of_statements (see [S0130], page 240)), the execution of every construct included within the aborted execution is also aborted, except for executions included within the execution of an <abort-deferred> operation; the execution of an abort-deferred operation continues to completion without being affected by the abort; the following are the abort-deferred operations:

6

- a protected action;

7

- waiting for an entry call to complete (after having initiated the attempt to cancel it -- see below);

8

- waiting for the termination of dependent tasks;

9

- the execution of an Initialize procedure as the last step of the default initialization of a controlled object;

10

- the execution of a Finalize procedure as part of the finalization of a controlled object;

11

- an assignment operation to an object with a controlled part.

12

The last three of these are discussed further in Section 8.6 [7.6], page 295.

13

When a master is aborted, all tasks that depend on that master are aborted.

14

The order in which tasks become abnormal as the result of an `abort_statement` or the abort of a `sequence_of_statements` (see [S0130], page 240) is not specified by the language.

15

If the execution of an entry call is aborted, an immediate attempt is made to cancel the entry call (see Section 10.5.3 [9.5.3], page 352). If the execution of a construct is aborted at a time when the execution is blocked, other than for an entry call, at a point that is outside the execution of an abort–deferred operation, then the execution of the construct completes immediately. For an abort due to an `abort_statement`, these immediate effects occur before the execution of the `abort_statement` completes. Other than for these immediate cases, the execution of a construct that is aborted does not necessarily complete before the `abort_statement` completes. However, the execution of the aborted construct completes no later than its next <abort completion point> (if any) that occurs outside of an abort–deferred operation; the following are abort completion points for an execution:

16

- the point where the execution initiates the activation of another task;

17

- the end of the activation of a task;

18

- the start or end of the execution of an entry call, `accept_statement`, `delay_statement`, or `abort_statement`;

19

- the start of the execution of a `select_statement`, or of the `sequence_of_statements` (see [S0130], page 240) of an `exception_handler`.

Bounded (Run-Time) Errors

20

An attempt to execute an `asynchronous_select` as part of the execution of an abort–deferred operation is a bounded error. Similarly, an attempt to create a task that depends on a master that is included entirely within the execution of an abort–deferred operation is a bounded error. In both cases, `Program_Error` is raised if the error is detected by the implementation; otherwise the operations proceed as they would outside an abort–deferred operation, except that an abort of the `abortable_part` or the created task might or might not have an effect.

Erroneous Execution

21

If an assignment operation completes prematurely due to an abort, the assignment is said to be <disrupted>; the target of the assignment or its parts can become abnormal, and certain subsequent uses of the object can be erroneous, as explained in Section 14.9.1 [13.9.1], page 522.

NOTES

22

41 An `abort_statement` should be used only in situations requiring unconditional termination.

23

42 A task is allowed to abort any task it can name, including itself.

24

43 Additional requirements associated with `abort` are given in Section 18.6 [D.6], page 1000, "Section 18.6 [D.6], page 1000, Preemptive Abort".

10.9 9.9 Task and Entry Attributes

Dynamic Semantics

1

For a prefix `T` that is of a task type (after any implicit dereference), the following attributes are defined:

2

`T'Callable`

Yields the value `True` when the task denoted by `T` is `<callable>`, and `False` otherwise; a task is callable unless it is completed or abnormal. The value of this attribute is of the predefined type `Boolean`.

3

`T'Terminated`

Yields the value `True` if the task denoted by `T` is terminated, and `False` otherwise. The value of this attribute is of the predefined type `Boolean`.

4

For a prefix `E` that denotes an entry of a task or protected unit, the following attribute is defined. This attribute is only allowed within the body of the task or protected unit, but

excluding, in the case of an entry of a task unit, within any program unit that is, itself, inner to the body of the task unit.

5

E'Count

Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type <universal_integer>.

NOTES

6

44 For the Count attribute, the entry can be either a single entry or an entry of a family. The name of the entry or entry family can be either a `direct_name` or an expanded name.

7

45 Within task units, algorithms interrogating the attribute E'Count should take precautions to allow for the increase of the value of this attribute for incoming entry calls, and its decrease, for example with `timed_entry_calls`. Also, a `conditional_entry_call` may briefly increase this value, even if the conditional call is not accepted.

8

46 Within protected units, algorithms interrogating the attribute E'Count in the `entry_barrier` for the entry E should take precautions to allow for the evaluation of the condition of the barrier both before and after queuing a given caller.

10.10 9.10 Shared Variables

Static Semantics

1

If two different objects, including nonoverlapping parts of the same object, are <independently addressable>, they can be manipulated concurrently by two different tasks without synchronization. Normally, any two nonoverlapping objects are independently addressable. However, if `packing`, `record layout`, or `Component_Size` is specified for a given composite object, then it is implementation defined whether or not two nonoverlapping parts of that composite object are independently addressable.

Dynamic Semantics

2

Separate tasks normally proceed independently and concurrently with one another. How-

ever, task interactions can be used to synchronize the actions of two or more tasks to allow, for example, meaningful communication by the direct updating and reading of variables shared between the tasks. The actions of two different tasks are synchronized in this sense when an action of one task <signals> an action of the other task; an action A1 is defined to signal an action A2 under the following circumstances:

3

- If A1 and A2 are part of the execution of the same task, and the language rules require A1 to be performed before A2;

4

- If A1 is the action of an activator that initiates the activation of a task, and A2 is part of the execution of the task that is activated;

5

- If A1 is part of the activation of a task, and A2 is the action of waiting for completion of the activation;

6

- If A1 is part of the execution of a task, and A2 is the action of waiting for the termination of the task;

6.1/1

- If A1 is the termination of a task T, and A2 is either the evaluation of the expression T.Terminated or a call to `Ada.Task_Identification.Is_Terminated` with an actual parameter that identifies T (see Section 17.7.1 [C.7.1], page 965);

7

- If A1 is the action of issuing an entry call, and A2 is part of the corresponding execution of the appropriate `entry_body` or `accept_statement`.

8

- If A1 is part of the execution of an `accept_statement` or `entry_body`, and A2 is the action of returning from the corresponding entry call;

9

- If A1 is part of the execution of a protected procedure body or `entry_body` for a given protected object, and A2 is part of a later execution of an `entry_body` for the same protected object;

10

- If A1 signals some action that in turn signals A2.

Erroneous Execution

11

Given an action of assigning to an object, and an action of reading or updating a part of the same object (or of a neighboring object if the two are not independently addressable), then the execution of the actions is erroneous unless the actions are <sequential>. Two actions are sequential if one of the following is true:

12

- One action signals the other;

13

- Both actions occur as part of the execution of the same task;

14

- Both actions occur as part of protected actions on the same protected object, and at most one of the actions is part of a call on a protected function of the protected object.

15

A pragma Atomic or Atomic_Components may also be used to ensure that certain reads and updates are sequential -- see Section 17.6 [C.6], page 962.

10.11 9.11 Example of Tasking and Synchronization

Examples

1

The following example defines a buffer protected object to smooth variations between the speed of output of a producing task and the speed of input of some consuming task. For instance, the producing task might have the following structure:

2

```
task Producer;
```

3/2

```
task body Producer is
  Person : Person_Name; --< see Section 4.10.1 [3.10.1], page 160>
begin
  loop
    ... --< simulate arrival of the next customer>
    Buffer.Append_Wait(Person);
    exit when Person = null;
  end loop;
end Producer;
```

4

and the consuming task might have the following structure:

5

```
task Consumer;
```

6/2

```
task body Consumer is
  Person : Person_Name;
begin
  loop
    Buffer.Remove_First_Wait(Person);
    exit when Person = null;
    ... --< simulate serving a customer>
  end loop;
end Consumer;
```

7/2

The buffer object contains an internal array of person names managed in a round-robin fashion. The array has two indices, an `In_Index` denoting the index for the next input person name and an `Out_Index` denoting the index for the next output person name.

7.1/2

The Buffer is defined as an extension of the `Synchronized_Queue` interface (see Section 4.9.4 [3.9.4], page 152), and as such promises to implement the abstraction defined by that interface. By doing so, the Buffer can be passed to the Transfer class-wide operation defined for objects of a type covered by `Queue'Class`.

8/2

```
protected Buffer is new Synchronized_Queue with --< see Section 4.9.4
[3.9.4], page 152>
  entry Append_Wait(Person : in Person_Name);
  entry Remove_First_Wait(Person : out Person_Name);
  function Cur_Count return Natural;
  function Max_Count return Natural;
  procedure Append(Person : in Person_Name);
  procedure Remove_First(Person : out Person_Name);
private
  Pool      : Person_Name_Array(1 .. 100);
  Count     : Natural := 0;
  In_Index, Out_Index : Positive := 1;
end Buffer;
```

9/2

```
protected body Buffer is
  entry Append_Wait(Person : in Person_Name)
    when Count < Pool'Length is
  begin
    Append(Person);
```


9.1/2 end Append_Wait;

```
procedure Append(Person : in Person_Name) is
begin
    if Count = Pool'Length then
        raise Queue_Error with "Buffer Full"; --< see Section 12.3
[11.3], page 421>
    end if;
    Pool(In_Index) := Person;
    In_Index      := (In_Index mod Pool'Length) + 1;
    Count         := Count + 1;
end Append;
```

10/2

```
entry Remove_First_Wait(Person : out Person_Name)
when Count > 0 is
begin
    Remove_First(Person);
end Remove_First_Wait;
```

11/2

```
procedure Remove_First(Person : out Person_Name) is
begin
    if Count = 0 then
        raise Queue_Error with "Buffer Empty"; --< see Section 12.3
[11.3], page 421>
    end if;
    Person := Pool(Out_Index);
    Out_Index := (Out_Index mod Pool'Length) + 1;
    Count := Count - 1;
end Remove_First;
```

12/2

```
function Cur_Count return Natural is
begin
    return Buffer.Count;
end Cur_Count;
```

13/2

```
function Max_Count return Natural is
begin
    return Pool'Length;
end Max_Count;
end Buffer;
```

11 10 Program Structure and Compilation Issues

1

The overall structure of programs and the facilities for separate compilation are described in this section. A <program> is a set of <partitions>, each of which may execute in a separate address space, possibly on a separate computer.

2

As explained below, a partition is constructed from <library units>. Syntactically, the declaration of a library unit is a library_item, as is the body of a library unit. An implementation may support a concept of a <program library> (or simply, a "library"), which contains library_items and their subunits. Library units may be organized into a hierarchy of children, grandchildren, and so on.

3

This section has two clauses: Section 11.1 [10.1], page 394, "Section 11.1 [10.1], page 394, Separate Compilation" discusses compile-time issues related to separate compilation. Section 11.2 [10.2], page 409, "Section 11.2 [10.2], page 409, Program Execution" discusses issues related to what is traditionally known as "link time" and "run time" -- building and executing partitions.

11.1 10.1 Separate Compilation

1

A <program unit> is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units.

2

The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation_units. A compilation_unit contains either the declaration, the body, or a renaming of a program unit. The representation for a compilation is implementation-defined.

3

A library unit is a separately compiled program unit, and is always a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a <subsystem>.

Implementation Permissions

4

An implementation may impose implementation-defined restrictions on compilations that contain multiple compilation_units.

11.1.1 10.1.1 Compilation Units - Library Units

1

A library_item is a compilation unit that is the declaration, body, or renaming of a library unit. Each library unit (except Standard) has a <parent unit>, which is a library package

or generic library package. A library unit is a <child> of its parent unit. The <root> library units are the children of the predefined library package Standard.

Syntax

2

```
compilation ::= {compilation_unit}
```

3

```
compilation_unit ::=  
    context_clause library_item  
    | context_clause subunit
```

4

```
library_item ::= [private] library_unit_declaration  
    | library_unit_body  
    | [private] library_unit_renaming_declaration
```

5

```
library_unit_declaration ::=  
    subprogram_declaration | package_declaration  
    | generic_declaration | generic_instantiation
```

6

```
library_unit_renaming_declaration ::=  
    package_renaming_declaration  
    | generic_renaming_declaration  
    | subprogram_renaming_declaration
```

7

```
library_unit_body ::= subprogram_body | package_body
```

8

```
parent_unit_name ::= name
```

8.1/2

An `overriding_indicator` is not allowed in a `subprogram_declaration`, `generic_instantiation`, or `subprogram_renaming_declaration` that declares a library unit.

9

A <library unit> is a program unit that is declared by a `library_item`. When a program unit is a library unit, the prefix "library" is used to refer to it (or "generic library" if generic), as well as to its declaration and body, as in "library procedure", "library package_body", or "generic library package". The term <compilation unit> is used to refer to

a `compilation_unit`. When the meaning is clear from context, the term is also used to refer to the `library_item` of a `compilation_unit` or to the `proper_body` of a subunit (that is, the `compilation_unit` without the `context_clause` and the separate (`parent_unit_name`)).

10

The `<parent declaration>` of a `library_item` (and of the library unit) is the declaration denoted by the `parent_unit_name` (see [S0234], page 395), if any, of the `defining_program_unit_name` (see [S0154], page 256) of the `library_item`. If there is no `parent_unit_name` (see [S0234], page 395), the parent declaration is the declaration of Standard, the `library_item` is a `<root>` `library_item`, and the library unit (renaming) is a `<root>` library unit (renaming). The declaration and body of Standard itself have no parent declaration. The `<parent unit>` of a `library_item` or library unit is the library unit declared by its parent declaration.

11

The children of a library unit occur immediately within the declarative region of the declaration of the library unit. The `<ancestors>` of a library unit are itself, its parent, its parent's parent, and so on. (Standard is an ancestor of every library unit.) The `<descendant>` relation is the inverse of the ancestor relation.

12

A `library_unit_declaration` or a `library_unit_renaming_declaration` (see [S0232], page 395) is `<private>` if the declaration is immediately preceded by the reserved word `private`; it is otherwise `<public>`. A library unit is private or public according to its declaration. The `<public descendants>` of a library unit are the library unit itself, and the public descendants of its public children. Its other descendants are `<private descendants>`.

12.1/2

For each `library_package_declaration` in the environment, there is an implicit declaration of a `<limited view>` of that library package. The limited view of a package contains:

12.2/2

- For each nested `package_declaration`, a declaration of the limited view of that package, with the same `defining_program_unit_name`.

12.3/2

- For each `type_declaration` in the visible part, an incomplete view of the type; if the `type_declaration` is tagged, then the view is a tagged incomplete view.

12.4/2

The limited view of a `library_package_declaration` is private if that `library_package_declaration` is immediately preceded by the reserved word `private`.

12.5/2

There is no syntax for declaring limited views of packages, because they are always implicit. The implicit declaration of a limited view of a library package is not the declaration of a library unit (the `library_package_declaration` is); nonetheless, it is a `library_item`. The implicit declaration of the limited view of a library package forms an (implicit) `compilation_unit` whose `context_clause` is empty.

12.6/2

A `library_package_declaration` is the completion of the declaration of its limited view.

Legality Rules

13

The parent unit of a `library_item` shall be a library package or generic library package.

14

If a `defining_program_unit_name` of a given declaration or body has a `parent_unit_name`, then the given declaration or body shall be a `library_item`. The body of a program unit shall be a `library_item` if and only if the declaration of the program unit is a `library_item`. In a `library_unit_renaming_declaration` (see [S0232], page 395), the (old) name shall denote a `library_item`.

15/2

A `parent_unit_name` (which can be used within a `defining_program_unit_name` of a `library_item` and in the separate clause of a subunit), and each of its prefixes, shall not denote a `renaming_declaration`. On the other hand, a name that denotes a `library_unit_renaming_declaration` (see [S0232], page 395) is allowed in a `nonlimited_with_clause` and other places where the name of a library unit is allowed.

16

If a library package is an instance of a generic package, then every child of the library package shall either be itself an instance or be a renaming of a library unit.

17

A child of a generic library package shall either be itself a generic unit or be a renaming of some other child of the same generic unit. The renaming of a child of a generic package shall occur only within the declarative region of the generic package.

18

A child of a parent generic package shall be instantiated or renamed only within the declarative region of the parent generic.

19/2

For each child `<C>` of some parent generic package `<P>`, there is a corresponding declaration `<C>` nested immediately within each instance of `<P>`. For the purposes of this rule, if a child `<C>` itself has a child `<D>`, each corresponding declaration for `<C>` has a corresponding child `<D>`. The corresponding declaration for a child within an instance is visible only within the scope of a `with_clause` that mentions the (original) child generic unit.

20

A library subprogram shall not override a primitive subprogram.

21

The defining name of a function that is a compilation unit shall not be an `operator_symbol`.

Static Semantics

22

A `subprogram_renaming_declaration` that is a `library_unit_renaming_declaration` (see [S0232], page 395) is a `renaming-as-declaration`, not a `renaming-as-body`.

23

There are two kinds of dependences among compilation units:

24

- The `<semantic dependences>` (see below) are the ones needed to check the compile-time rules across compilation unit boundaries; a compilation unit depends semantically on

the other compilation units needed to determine its legality. The visibility rules are based on the semantic dependences.

25

- The <elaboration dependences> (see Section 11.2 [10.2], page 409) determine the order of elaboration of library_items.

26/2

A library_item depends semantically upon its parent declaration. A subunit depends semantically upon its parent body. A library_unit_body depends semantically upon the corresponding library_unit_declaration, if any. The declaration of the limited view of a library package depends semantically upon the declaration of the limited view of its parent. The declaration of a library package depends semantically upon the declaration of its limited view. A compilation unit depends semantically upon each library_item mentioned in a with_clause of the compilation unit. In addition, if a given compilation unit contains an attribute_reference of a type defined in another compilation unit, then the given compilation unit depends semantically upon the other compilation unit. The semantic dependence relationship is transitive.

Dynamic Semantics

26.1/2

The elaboration of the declaration of the limited view of a package has no effect.

NOTES

27

1 A simple program may consist of a single compilation unit. A compilation need not have any compilation units; for example, its text can consist of pragmas.

28

2 The designator of a library function cannot be an operator_symbol, but a nonlibrary renaming_declaration is allowed to rename a library function as an operator. Within a partition, two library subprograms are required to have distinct names and hence cannot overload each other. However, renaming_declarations are allowed to define overloaded names for such subprograms, and a locally declared subprogram is allowed to overload a library subprogram. The expanded name Standard.L can be used to denote a root library unit L (unless the declaration of Standard is hidden) since root library unit declarations occur immediately within the declarative region of package Standard.

Examples

29

<Examples of library units:>

30

```

package Rational_Numbers.IO is --< public child of Rational_Numbers, see Section
[7.1], page 279>
    procedure Put(R : in Rational);
    procedure Get(R : out Rational);
end Rational_Numbers.IO;
31

private procedure Rational_Numbers.Reduce(R : in out Rational);
--< private child of Rational_Numbers>■

32

with Rational_Numbers.Reduce; --< refer to a private child>
package body Rational_Numbers is
    ...
end Rational_Numbers;
33

with Rational_Numbers.IO; use Rational_Numbers;
with Ada.Text_io; --< see Section 15.10 [A.10], page 696>■
procedure Main is --< a root library procedure>
    R : Rational;
begin
    R := 5/3; --< construct a rational number, see Section 8.1
[7.1], page 279>
    Ada.Text_IO.Put("The answer is: ");
    IO.Put(R);
    Ada.Text_IO.New_Line;
end Main;
34

with Rational_Numbers.IO;
package Rational_IO renames Rational_Numbers.IO;
--< a library unit renaming declaration>■
35

```

Each of the above library_items can be submitted to the compiler separately.

11.1.2 10.1.2 Context Clauses - With Clauses

1

A context_clause is used to specify the library_items whose names are needed within a compilation unit.

Syntax

2

```
context_clause ::= {context_item}
```

3

context_item ::= with_clause | use_clause

4/2

with_clause ::= limited_with_clause | nonlimited_with_clause

4.1/2

limited_with_clause ::= limited [private] with <library_unit_>name {, <library_unit_>name};

4.2/2

nonlimited_with_clause ::= [private] with <library_unit_>name {, <library_unit_>name};
Name Resolution Rules

5

The <scope> of a with_clause that appears on a library_unit_declaration (see [S0231], page 395) or library_unit_renaming_declaration (see [S0232], page 395) consists of the entire declarative region of the declaration, which includes all children and subunits. The scope of a with_clause that appears on a body consists of the body, which includes all subunits.

6/2

A library_item (and the corresponding library unit) is <named> in a with_clause if it is denoted by a <library_unit_>name in the with_clause. A library_item (and the corresponding library unit) is <mentioned> in a with_clause if it is named in the with_clause or if it is denoted by a prefix in the with_clause.

7

Outside its own declarative region, the declaration or renaming of a library unit can be visible only within the scope of a with_clause that mentions it. The visibility of the declaration or renaming of a library unit otherwise follows from its placement in the environment.

Legality Rules

8/2

If a with_clause of a given compilation_unit mentions a private child of some library unit, then the given compilation_unit shall be one of:

9/2

- the declaration, body, or subunit of a private descendant of that library unit;

10/2

- the body or subunit of a public descendant of that library unit, but not a subprogram body acting as a subprogram declaration (see Section 11.1.4 [10.1.4], page 406); or

11/2

- the declaration of a public descendant of that library unit, in which case the with_clause shall include the reserved word private.

12/2

A name denoting a library item that is visible only due to being mentioned in one or more `with`-clauses that include the reserved word `private` shall appear only within:

13/2

- a private part;

14/2

- a body, but not within the `subprogram_specification` of a library subprogram body;

15/2

- a private descendant of the unit on which one of these `with`-clauses appear; or

16/2

- a pragma within a context clause.

17/2

A `library_item` mentioned in a `limited_with_clause` shall be the implicit declaration of the limited view of a library package, not the declaration of a subprogram, generic unit, generic instance, or a renaming.

18/2

A `limited_with_clause` shall not appear on a `library_unit_body`, subunit, or `library_unit_renaming_declaration` (see [S0232], page 395).

19/2

A `limited_with_clause` that names a library package shall not appear:

20/2

- in the `context_clause` for the explicit declaration of the named library package;

21/2

- in the same `context_clause` as, or within the scope of, a `nonlimited_with_clause` that mentions the same library package; or

22/2

- in the same `context_clause` as, or within the scope of, a `use_clause` that names an entity declared within the declarative region of the library package.

NOTES

23/2

3 A `library_item` mentioned in a `nonlimited_with_clause` of a compilation unit is visible within the compilation unit and hence acts just like an ordinary declaration. Thus, within a compilation unit that

mentions its declaration, the name of a library package can be given in use_clauses and can be used to form expanded names, a library subprogram can be called, and instances of a generic library unit can be declared. If a child of a parent generic package is mentioned in a nonlimited_with_clause, then the corresponding declaration nested within each visible instance is visible within the compilation unit. Similarly, a library_item mentioned in a limited_with_clause of a compilation unit is visible within the compilation unit and thus can be used to form expanded names.

Examples

24/2

```
package Office is
end Office;
```

25/2

```
with Ada.Strings.Unbounded;
package Office.Locations is
  type Location is new Ada.Strings.Unbounded.Unbounded_String;
end Office.Locations;
```

26/2

```
limited with Office.Departments;  --< types are incomplete>
private with Office.Locations;   --< only visible in private part>
package Office.Employees is
  type Employee is private;
```

27/2

```
function Dept_Of(Emp : Employee) return access Departments.Department;
procedure Assign_Dept(Emp : in out Employee;
                      Dept : access Departments.Department);
```

28/2

```
...
private
  type Employee is
    record
      Dept : access Departments.Department;
      Loc : Locations.Location;
      ...
    end record;
end Office.Employees;
```

29/2

```
limited with Office.Employees;
```

```

package Office.Departments is
    type Department is private;
30/2

    function Manager_Of(Dept : Department) return access Employees.Employee;
    procedure Assign_Manager(Dept : in out Department;
                             Mgr  : access Employees.Employee);
    ...
end Office.Departments;

```

31/2

The `limited_with_clause` may be used to support mutually dependent abstractions that are split across multiple packages. In this case, an employee is assigned to a department, and a department has a manager who is an employee. If a `with_clause` with the reserved word `private` appears on one library unit and mentions a second library unit, it provides visibility to the second library unit, but restricts that visibility to the private part and body of the first unit. The compiler checks that no use is made of the second unit in the visible part of the first unit.

11.1.3 10.1.3 Subunits of Compilation Units

1

Subunits are like child units, with these (important) differences: subunits support the separate compilation of bodies only (not declarations); the parent contains a `body_stub` to indicate the existence and place of each of its subunits; declarations appearing in the parent's body can be visible within the subunits.

Syntax

2

```
body_stub ::= subprogram_body_stub | package_body_stub | task_body_stub | protected_body_s
```

3/2

```
subprogram_body_stub ::=
    [overriding_indicator]
    subprogram_specification is separate;

```

4

```
package_body_stub ::= package body defining_identifier is separate;

```

5

```
task_body_stub ::= task body defining_identifier is separate;

```

6

```
protected_body_stub ::= protected body defining_identifier is separate;

```

7

subunit ::= separate (parent_unit_name) proper_body

Legality Rules

8/2

The <parent body> of a subunit is the body of the program unit denoted by its parent_unit_name. The term <subunit> is used to refer to a subunit and also to the proper_body of a subunit. The <subunits of a program unit> include any subunit that names that program unit as its parent, as well as any subunit that names such a subunit as its parent (recursively).

9

The parent body of a subunit shall be present in the current environment, and shall contain a corresponding body_stub with the same defining_identifier as the subunit.

10/2

A package_body_stub shall be the completion of a package_declaration (see [S0173], page 279) or generic_package_declaration (see [S0254], page 450); a task_body_stub (see [S0243], page 403) shall be the completion of a task declaration; a protected_body_stub (see [S0244], page 403) shall be the completion of a protected declaration.

11

In contrast, a subprogram_body_stub need not be the completion of a previous declaration, in which case the _stub declares the subprogram. If the _stub is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a subprogram_body_stub that completes a declaration shall conform fully to that of the declaration.

12

A subunit that corresponds to a body_stub shall be of the same kind (package_, subprogram_, task_, or protected_) as the body_stub. The profile of a subprogram_body subunit shall be fully conformant to that of the corresponding body_stub.

13

A body_stub shall appear immediately within the declarative_part of a compilation unit body. This rule does not apply within an instance of a generic unit.

14

The defining_identifiers of all body_stubs that appear immediately within a particular declarative_part shall be distinct.

Post-Compilation Rules

15

For each body_stub, there shall be a subunit containing the corresponding proper_body.

NOTES

16

4 The rules in Section 11.1.4 [10.1.4], page 406, "Section 11.1.4 [10.1.4], page 406, The Compilation Process" say that a body_stub is equivalent to the corresponding proper_body. This implies:

17

- Visibility within a subunit is the visibility that would be obtained at the place of the corresponding body_stub (within the

parent body) if the context_clause of the subunit were appended to that of the parent body.

18

- The effect of the elaboration of a body_stub is to elaborate the subunit.

Examples

19

The package Parent is first written without subunits:

20

```
package Parent is
  procedure Inner;
end Parent;
```

21

```
with Ada.Text_IO;
package body Parent is
  Variable : String := "Hello, there.";
  procedure Inner is
  begin
    Ada.Text_IO.Put_Line(Variable);
  end Inner;
end Parent;
```

22

The body of procedure Inner may be turned into a subunit by rewriting the package body as follows (with the declaration of Parent remaining the same):

23

```
package body Parent is
  Variable : String := "Hello, there.";
  procedure Inner is separate;
end Parent;
```

24

```
with Ada.Text_IO;
separate(Parent)
procedure Inner is
begin
  Ada.Text_IO.Put_Line(Variable);
end Inner;
```

11.1.4 10.1.4 The Compilation Process

1

Each compilation unit submitted to the compiler is compiled in the context of an <environment> declarative_part (or simply, an <environment>), which is a conceptual declarative_part that forms the outermost declarative region of the context of any compilation. At run time, an environment forms the declarative_part of the body of the environment task of a partition (see Section 11.2 [10.2], page 409, "Section 11.2 [10.2], page 409, Program Execution").

2

The declarative_items of the environment are library_items appearing in an order such that there are no forward semantic dependences. Each included subunit occurs in place of the corresponding stub. The visibility rules apply as if the environment were the outermost declarative region, except that with_clause (see [S0237], page 400)s are needed to make declarations of library units visible (see Section 11.1.2 [10.1.2], page 399).

3/2

The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined. The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment are implementation defined.

Name Resolution Rules

4/1

If a library_unit_body that is a subprogram_body is submitted to the compiler, it is interpreted only as a completion if a library_unit_declaration with the same defining_program_unit_name already exists in the environment for a subprogram other than an instance of a generic subprogram or for a generic subprogram (even if the profile of the body is not type conformant with that of the declaration); otherwise the subprogram_body is interpreted as both the declaration and body of a library subprogram.

Legality Rules

5

When a compilation unit is compiled, all compilation units upon which it depends semantically shall already exist in the environment; the set of these compilation units shall be <consistent> in the sense that the new compilation unit shall not semantically depend (directly or indirectly) on two different versions of the same compilation unit, nor on an earlier version of itself.

Implementation Permissions

6/2

The implementation may require that a compilation unit be legal before it can be mentioned in a limited_with_clause or it can be inserted into the environment.

7/2

When a compilation unit that declares or renames a library unit is added to the environment, the implementation may remove from the environment any preexisting library_item or subunit with the same full expanded name. When a compilation unit that is a subunit or the body of a library unit is added to the environment, the implementation may remove from the environment any preexisting version of the same compilation unit. When a compi-

lation unit that contains a `body_stub` is added to the environment, the implementation may remove any preexisting `library_item` or subunit with the same full expanded name as the `body_stub`. When a given compilation unit is removed from the environment, the implementation may also remove any compilation unit that depends semantically upon the given one. If the given compilation unit contains the body of a subprogram to which a `pragma Inline` applies, the implementation may also remove any compilation unit containing a call to that subprogram.

NOTES

8

5 The rules of the language are enforced across compilation and compilation unit boundaries, just as they are enforced within a single compilation unit.

9

6 An implementation may support a concept of a `<library>`, which contains `library_items`. If multiple libraries are supported, the implementation has to define how a single environment is constructed when a compilation unit is submitted to the compiler. Naming conflicts between different libraries might be resolved by treating each library as the root of a hierarchy of child library units.

10

7 A compilation unit containing an instantiation of a separately compiled generic unit does not semantically depend on the body of the generic unit. Therefore, replacing the generic body in the environment does not result in the removal of the compilation unit containing the instantiation.

11.1.5 10.1.5 Pragmas and Program Units

1

This subclause discusses pragmas related to program units, library units, and compilations.

Name Resolution Rules

2

Certain pragmas are defined to be `<program unit pragmas>`. A name given as the argument of a program unit pragma shall resolve to denote the declarations or renamings of one or more program units that occur immediately within the declarative region or compilation in which the pragma immediately occurs, or it shall resolve to denote the declaration of the immediately enclosing program unit (if any); the pragma applies to the denoted program unit(s). If there are no names given as arguments, the pragma applies to the immediately enclosing program unit.

Legality Rules

3

A program unit pragma shall appear in one of these places:

4

- At the place of a `compilation_unit`, in which case the pragma shall immediately follow in the same compilation (except for other pragmas) a `library_unit_declaration` (see [S0231], page 395) that is a `subprogram_declaration` (see [S0148], page 255), `generic_subprogram_declaration` (see [S0253], page 450), or `generic_instantiation` (see [S0257], page 455), and the pragma shall have an argument that is a name denoting that declaration.

5/1

- Immediately within the visible part of a program unit and before any nested declaration (but not within a generic formal part), in which case the argument, if any, shall be a `direct_name` that denotes the immediately enclosing program unit declaration.

6

- At the place of a declaration other than the first, of a `declarative_part` or program unit declaration, in which case the pragma shall have an argument, which shall be a `direct_name` that denotes one or more of the following (and nothing else): a `subprogram_declaration` (see [S0148], page 255), a `generic_subprogram_declaration` (see [S0253], page 450), or a `generic_instantiation` (see [S0257], page 455), of the same `declarative_part` (see [S0086], page 175) or program unit declaration.

7

Certain program unit pragmas are defined to be <library unit pragmas>. The name, if any, in a library unit pragma shall denote the declaration of a library unit.

Static Semantics

7.1/1

A library unit pragma that applies to a generic unit does not apply to its instances, unless a specific rule for the pragma specifies the contrary.

Post-Compilation Rules

8

Certain pragmas are defined to be <configuration pragmas>; they shall appear before the first `compilation_unit` of a compilation. They are generally used to select a partition-wide or system-wide option. The pragma applies to all `compilation_units` appearing in the compilation, unless there are none, in which case it applies to all future `compilation_units` compiled into the same environment.

Implementation Permissions

9/2

An implementation may require that configuration pragmas that select partition-wide or system-wide options be compiled when the environment contains no `library_items` other than those of the predefined environment. In this case, the implementation shall still accept configuration pragmas in individual compilations that confirm the initially selected partition-wide or system-wide options.

Implementation Advice

10/1

When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance.

11.1.6 10.1.6 Environment-Level Visibility Rules

1

The normal visibility rules do not apply within a `parent_unit_name` or a `context_clause`, nor within a pragma that appears at the place of a compilation unit. The special visibility rules for those contexts are given here.

Static Semantics

2/2

Within the `parent_unit_name` at the beginning of an explicit `library_item`, and within a `nonlimited_with_clause`, the only declarations that are visible are those that are explicit `library_items` of the environment, and the only declarations that are directly visible are those that are explicit root `library_items` of the environment. Within a `limited_with_clause`, the only declarations that are visible are those that are the implicit declaration of the limited view of a library package of the environment, and the only declarations that are directly visible are those that are the implicit declaration of the limited view of a root library package.

3

Within a `use_clause` or pragma that is within a `context_clause`, each `library_item` mentioned in a previous `with_clause` of the same `context_clause` is visible, and each root `library_item` so mentioned is directly visible. In addition, within such a `use_clause`, if a given declaration is visible or directly visible, each declaration that occurs immediately within the given declaration's visible part is also visible. No other declarations are visible or directly visible.

4

Within the `parent_unit_name` of a subunit, `library_items` are visible as they are in the `parent_unit_name` of a `library_item`; in addition, the declaration corresponding to each `body_stub` in the environment is also visible.

5

Within a pragma that appears at the place of a compilation unit, the immediately preceding `library_item` and each of its ancestors is visible. The ancestor root `library_item` is directly visible.

6/2

Notwithstanding the rules of Section 5.1.3 [4.1.3], page 183, an expanded name in a `with_clause`, a pragma in a `context_clause`, or a pragma that appears at the place of a compilation unit may consist of a prefix that denotes a generic package and a `selector_name` that denotes a child of that generic package. (The child is necessarily a generic unit; see Section 11.1.1 [10.1.1], page 394.)

11.2 10.2 Program Execution

1

An Ada <program> consists of a set of <partitions>, which can execute in parallel with one another, possibly in a separate address space, and possibly on a separate computer.

Post-Compilation Rules

2

A partition is a program or part of a program that can be invoked from outside the Ada implementation. For example, on many systems, a partition might be an executable file generated by the system linker. The user can <explicitly assign> library units to a partition. The assignment is done in an implementation–defined manner. The compilation units included in a partition are those of the explicitly assigned library units, as well as other compilation units <needed by> those library units. The compilation units needed by a given compilation unit are determined as follows (unless specified otherwise via an implementation–defined pragma, or by some other implementation–defined means):

3

- A compilation unit needs itself;

4

- If a compilation unit is needed, then so are any compilation units upon which it depends semantically;

5

- If a `library_unit_declaration` is needed, then so is any corresponding `library_unit_body`;

6/2

- If a compilation unit with stubs is needed, then so are any corresponding subunits;

6.1/2

- If the (implicit) declaration of the limited view of a library package is needed, then so is the explicit declaration of the library package.

7

The user can optionally designate (in an implementation–defined manner) one subprogram as the <main subprogram> for the partition. A main subprogram, if specified, shall be a subprogram.

8

Each partition has an anonymous <environment task>, which is an implicit outermost task whose execution elaborates the `library_items` of the `environment_declarative_part`, and then calls the main subprogram, if there is one. A partition’s execution is that of its tasks.

9

The order of elaboration of library units is determined primarily by the <elaboration dependences>. There is an elaboration dependence of a given `library_item` upon another if the given `library_item` or any of its subunits depends semantically on the other `library_item`. In addition, if a given `library_item` or any of its subunits has a pragma `Elaborate` or `Elaborate_All` that names another library unit, then there is an elaboration dependence of the given `library_item` upon the body of the other library unit, and, for `Elaborate_All` only, upon each `library_item` needed by the declaration of the other library unit.

10

The environment task for a partition has the following structure:

11

```
task <Environment_Task>;
```

12/2

```
task body <Environment_Task> is
    ... (1) --< The environment >declarative_part
           --< (that is, the sequence of >library_item<s) goes here.>■
begin
    ... (2) --< Call the main subprogram, if there is one.>
end <Environment_Task>;
```

13

The environment declarative_part at (1) is a sequence of declarative_items consisting of copies of the library_items included in the partition. The order of elaboration of library_items is the order in which they appear in the environment declarative_part:

14

- The order of all included library_items is such that there are no forward elaboration dependences.

15

- Any included library_unit_declaration to which a pragma Elaborate_Body applies is immediately followed by its library_unit_body, if included.

16

- All library_items declared pure occur before any that are not declared pure.

17

- All preelaborated library_items occur before any that are not preelaborated.

18

There shall be a total order of the library_items that obeys the above rules. The order is otherwise implementation defined.

19

The full expanded names of the library units and subunits included in a given partition shall be distinct.

20

The sequence_of_statements of the environment task (see (2) above) consists of either:

21

- A call to the main subprogram, if the partition has one. If the main subprogram has parameters, they are passed; where the actuals come from is implementation defined. What happens to the result of a main function is also implementation defined.

22

or:

23

- A `null_statement`, if there is no main subprogram.

24

The mechanisms for building and running partitions are implementation defined. These might be combined into one operation, as, for example, in dynamic linking, or "load-and-go" systems.

Dynamic Semantics

25

The execution of a program consists of the execution of a set of partitions. Further details are implementation defined. The execution of a partition starts with the execution of its environment task, ends when the environment task terminates, and includes the executions of all tasks of the partition. The execution of the (implicit) `task_body` of the environment task acts as a master for all other tasks created as part of the execution of the partition. When the environment task completes (normally or abnormally), it waits for the termination of all such tasks, and then finalizes any remaining objects of the partition.

Bounded (Run-Time) Errors

26

Once the environment task has awaited the termination of all other tasks of the partition, any further attempt to create a task (during finalization) is a bounded error, and may result in the raising of `Program_Error` either upon creation or activation of the task. If such a task is activated, it is not specified whether the task is awaited prior to termination of the environment task.

Implementation Requirements

27

The implementation shall ensure that all compilation units included in a partition are consistent with one another, and are legal according to the rules of the language.

Implementation Permissions

28

The kind of partition described in this clause is known as an `<active>` partition. An implementation is allowed to support other kinds of partitions, with implementation-defined semantics.

29

An implementation may restrict the kinds of subprograms it supports as main subprograms. However, an implementation is required to support all main subprograms that are public parameterless library procedures.

30

If the environment task completes abnormally, the implementation may abort any dependent tasks.

NOTES

31

8 An implementation may provide inter-partition communication mechanism(s) via special packages and pragmas. Standard pragmas for distribution and methods for specifying inter-partition communication are defined in Chapter 19 [Annex E], page 1034, "Chapter 19 [Annex E], page 1034, Distributed Systems". If no such mechanisms are provided, then each partition is isolated from all others, and behaves as a program in and of itself.

32

9 Partitions are not required to run in separate address spaces. For example, an implementation might support dynamic linking via the partition concept.

33

10 An order of elaboration of `library_items` that is consistent with the partial ordering defined above does not always ensure that each `library_unit_body` is elaborated before any other compilation unit whose elaboration necessitates that the `library_unit_body` be already elaborated. (In particular, there is no requirement that the body of a library unit be elaborated as soon as possible after the `library_unit_declaration` is elaborated, unless the pragmas in subclause Section 11.2.1 [10.2.1], page 413, are used.)

34

11 A partition (active or otherwise) need not have a main subprogram. In such a case, all the work done by the partition would be done by elaboration of various `library_items`, and by tasks created by that elaboration. Passive partitions, which cannot have main subprograms, are defined in Chapter 19 [Annex E], page 1034, "Chapter 19 [Annex E], page 1034, Distributed Systems".

11.2.1 10.2.1 Elaboration Control

1

This subclause defines pragmas that help control the elaboration order of `library_items`.

Syntax

2

The form of a pragma `Preelaborate` is as follows:

3

```
pragma Preelaborate[(<library_unit_>name)];
```

4

A pragma `Preelaborate` is a library unit pragma.

4.1/2

The form of a pragma `Preelaborable_Initialization` is as follows:

4.2/2

```
pragma Preelaborable_Initialization(direct_name);  
Legality Rules
```

5

An elaborable construct is preelaborable unless its elaboration performs any of the following actions:

6

- The execution of a statement other than a `null_statement`.

7

- A call to a subprogram other than a static function.

8

- The evaluation of a primary that is a name of an object, unless the name is a static expression, or statically denotes a discriminant of an enclosing type.

9/2

- The creation of an object (including a component) of a type that does not have preelaborable initialization. Similarly, the evaluation of an `extension_aggregate` (see [S0109], page 194) with an `ancestor_subtype_mark` (see [S0028], page 56) denoting a subtype of such a type.

10/2

A generic body is preelaborable only if elaboration of a corresponding instance body would not perform any such actions, presuming that:

10.1/2

- the actual for each formal private type (or extension) declared within the formal part of the generic unit is a private type (or extension) that does not have preelaborable initialization;

10.2/2

- the actual for each formal type is nonstatic;

10.3/2

- the actual for each formal object is nonstatic; and

10.4/2

- the actual for each formal subprogram is a user–defined subprogram.

11/1

If a pragma Preelaborate (or pragma Pure — see below) applies to a library unit, then it is <preelaborated>. If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all non–preelaborated library_items of the partition. The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit. In addition, all compilation units of a preelaborated library unit shall depend semantically only on compilation units of other preelaborated library units.

11.1/2

The following rules specify which entities have <preelaborable initialization>:

11.2/2

- The partial view of a private type or private extension, a protected type without entry_declarations, a generic formal private type, or a generic formal derived type, have preelaborable initialization if and only if the pragma Preelaborable_Initialization has been applied to them. A protected type with entry_declarations or a task type never has preelaborable initialization.

11.3/2

- A component (including a discriminant) of a record or protected type has preelaborable initialization if its declaration includes a default_expression whose execution does not perform any actions prohibited in preelaborable constructs as described above, or if its declaration does not include a default expression and its type has preelaborable initialization.

11.4/2

- A derived type has preelaborable initialization if its parent type has preelaborable initialization and (in the case of a derived record extension) if the non–inherited components all have preelaborable initialization. However, a user–defined controlled type with an overriding Initialize procedure does not have preelaborable initialization.

11.5/2

- A view of a type has preelaborable initialization if it is an elementary type, an array type whose component type has preelaborable initialization, a record type whose components all have preelaborable initialization, or an interface type.

11.6/2

A pragma `Preelaborable_Initialization` specifies that a type has preelaborable initialization. This pragma shall appear in the visible part of a package or generic package.

11.7/2

If the pragma appears in the first list of `basic_declarative_items` of a `package_specification`, then the `direct_name` shall denote the first subtype of a private type, private extension, or protected type that is not an interface type and is without `entry_declarations`, and the type shall be declared immediately within the same package as the pragma. If the pragma is applied to a private type or a private extension, the full view of the type shall have preelaborable initialization. If the pragma is applied to a protected type, each component of the protected type shall have preelaborable initialization. In addition to the places where Legality Rules normally apply, these rules apply also in the private part of an instance of a generic unit.

11.8/2

If the pragma appears in a `generic_formal_part`, then the `direct_name` shall denote a generic formal private type or a generic formal derived type declared in the same `generic_formal_part` as the pragma. In a `generic_instantiation` the corresponding actual type shall have preelaborable initialization.

Implementation Advice

12

In an implementation, a type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package, whether the elaborations occur in distinct executions of the same program, or in executions of distinct programs or partitions that include the given version.

Syntax

13

The form of a pragma `Pure` is as follows:

14

```
pragma Pure[(<library_unit_name>)];
```

15

A pragma `Pure` is a library unit pragma.

Static Semantics

15.1/2

A `<pure>` `library_item` is a preelaborable `library_item` whose elaboration does not perform any of the following actions:

15.2/2

- the elaboration of a variable declaration;

15.3/2

- the evaluation of an allocator of an access-to-variable type; for the purposes of this rule, the partial view of a type is presumed to have non-visible components whose default initialization evaluates such an allocator;

15.4/2

- the elaboration of the declaration of a named access-to-variable type unless the `Storage_Size` of the type has been specified by a static expression with value zero or is defined by the language to be zero;

15.5/2

- the elaboration of the declaration of a named access-to-constant type for which the `Storage_Size` has been specified by an expression other than a static expression with value zero.

15.6/2

The `Storage_Size` for an anonymous access-to-variable type declared at library level in a library unit that is declared pure is defined to be zero.

Legality Rules

16/2

<This paragraph was deleted.>

17/2

A pragma `Pure` is used to declare that a library unit is pure. If a pragma `Pure` applies to a library unit, then its compilation units shall be pure, and they shall depend semantically only on compilation units of other library units that are declared pure. Furthermore, the full view of any partial view declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see Section 14.13.2 [13.13.2], page 540).

Implementation Permissions

18/2

If a library unit is declared pure, then the implementation is permitted to omit a call on a library-level subprogram of the library unit if the results are not needed after the call. In addition, the implementation may omit a call on such a subprogram and simply reuse the results produced by an earlier call on the same subprogram, provided that none of the parameters nor any object accessible via access values from the parameters are of a limited type, and the addresses and values of all by-reference actual parameters, the values of all by-copy-in actual parameters, and the values of all objects accessible via access values from the parameters, are the same as they were at the earlier call. This permission applies even if the subprogram produces other side effects when called.

Syntax

19

The form of a pragma `Elaborate`, `Elaborate_All`, or `Elaborate_Body` is as follows:

20

21 pragma Elaborate(<library_unit_>name{, <library_unit_>name});

21

 pragma Elaborate_All(<library_unit_>name{, <li-
brary_unit_>name});

22

 pragma Elaborate_Body[(<library_unit_>name)];

23

A pragma Elaborate or Elaborate_All is only allowed within a con-
text_clause.

24

A pragma Elaborate_Body is a library unit pragma.

Legality Rules

25

If a pragma Elaborate_Body applies to a declaration, then the declaration requires a completion (a body).

25.1/2

The <library_unit_>name of a pragma Elaborate or Elaborate_All shall denote a nonlimited view of a library unit.

Static Semantics

26

A pragma Elaborate specifies that the body of the named library unit is elaborated before the current library_item. A pragma Elaborate_All specifies that each library_item that is needed by the named library unit declaration is elaborated before the current library_item. A pragma Elaborate_Body specifies that the body of the library unit is elaborated immediately after its declaration.

NOTES

27

12 A preelaborated library unit is allowed to have non-preelaborable children.

28

13 A library unit that is declared pure is allowed to have impure children.

12 11 Exceptions

1

This section defines the facilities for dealing with errors or other exceptional situations that arise during program execution. An `<exception>` represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an `<exception occurrence>`. To `<raise>` an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called `<handling>` the exception.

2

An `exception_declaration` declares a name for an exception. An exception is raised initially either by a `raise_statement` or by the failure of a language-defined check. When an exception arises, control can be transferred to a user-provided `exception_handler` at the end of a `handled_sequence_of_statements` (see [S0247], page 420), or it can be propagated to a dynamically enclosing execution.

12.1 11.1 Exception Declarations

1

An `exception_declaration` declares a name for an exception.

Syntax

2

`exception_declaration ::= defining_identifier_list : exception;`

Static Semantics

3

Each single `exception_declaration` declares a name for a different exception. If a generic unit includes an `exception_declaration`, the `exception_declarations` implicitly generated by different instantiations of the generic unit refer to distinct exceptions (but all have the same `defining_identifier`). The particular exception denoted by an exception name is determined at compilation time and is the same regardless of how many times the `exception_declaration` is elaborated.

4

The `<predefined>` exceptions are the ones declared in the declaration of package `Standard`: `Constraint_Error`, `Program_Error`, `Storage_Error`, and `Tasking_Error`; one of them is raised when a language-defined check fails.

Dynamic Semantics

5

The elaboration of an `exception_declaration` has no effect.

6

The execution of any construct raises `Storage_Error` if there is insufficient storage for that execution. The amount of storage needed for the execution of constructs is unspecified.

Examples

7

`<Examples of user-defined exception declarations:>`

8

```
Singular : exception;  
Error    : exception;  
Overflow, Underflow : exception;
```

12.2 11.2 Exception Handlers

1

The response to one or more exceptions is specified by an `exception_handler`.

Syntax

2

```
handled_sequence_of_statements ::=  
    sequence_of_statements  
    [exception  
     exception_handler  
     {exception_handler}]
```

3

```
exception_handler ::=  
    when [choice_parameter_specification:] exception_choice { | exception_choice } =>  
  
    sequence_of_statements
```

4

```
choice_parameter_specification ::= defining_identifier
```

5

```
exception_choice ::= <exception_>name | others  
Legality Rules
```

6

A choice with an `<exception_>name` `<covers>` the named exception. A choice with `others` covers all exceptions not named by previous choices of the same `handled_sequence_of_statements` (see [S0247], page 420). Two choices in different `exception_handlers` of the same `handled_sequence_of_statements` (see [S0247], page 420) shall not cover the same exception.

7

A choice with `others` is allowed only for the last handler of a `handled_sequence_of_statements` and as the only choice of that handler.

8

An `<exception_>name` of a choice shall not denote an exception declared in a generic formal package.

Static Semantics

9

A `choice_parameter_specification` declares a `<choice parameter>`, which is a constant object

of type `Exception_Occurrence` (see Section 12.4.1 [11.4.1], page 423). During the handling of an exception occurrence, the choice parameter, if any, of the handler represents the exception occurrence that is being handled.

Dynamic Semantics

10

The execution of a `handled_sequence_of_statements` consists of the execution of the `sequence_of_statements` (see [S0130], page 240). The optional handlers are used to handle any exceptions that are propagated by the `sequence_of_statements` (see [S0130], page 240).

Examples

11

<Example of an exception handler:>

12

```
begin
  Open(File, In_File, "input.txt");  --< see Section 15.8.2 [A.8.2],
page 685>
exception
  when E : Name_Error =>
    Put("Cannot open input file : ");
    Put_Line(Exception_Message(E));  --< see Section 12.4.1 [11.4.1],
page 423>
    raise;
end;
```

12.3 11.3 Raise Statements

1

A `raise_statement` raises an exception.

Syntax

2/2

```
raise_statement ::= raise;
| raise <exception_>name [with <string_>expression];
```

Legality Rules

3

The name, if any, in a `raise_statement` shall denote an exception. A `raise_statement` with no `<exception_>name` (that is, a `<re-raise statement>`) shall be within a handler, but not within a body enclosed by that handler.

Name Resolution Rules

3.1/2

The expression, if any, in a `raise_statement`, is expected to be of type `String`.

Dynamic Semantics

4/2

To `<raise an exception>` is to raise a new occurrence of that exception, as explained in

Section 12.4 [11.4], page 422. For the execution of a `raise_statement` with an `<exception_name>`, the named exception is raised. If a `<string_expression>` is present, the expression is evaluated and its value is associated with the exception occurrence. For the execution of a `re-raise` statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again.

Examples

5

`<Examples of raise statements:>`

6/2

```
raise Ada.IO_Exceptions.Name_Error;    --< see Section 15.13 [A.13],
page 752>
raise Queue_Error with "Buffer Full";  --< see Section 10.11 [9.11],
page 391>
```

7

```
raise;                                --< re-raise the current exception>■
```

12.4 11.4 Exception Handling

1

When an exception occurrence is raised, normal program execution is abandoned and control is transferred to an applicable exception_handler, if any. To `<handle>` an exception occurrence is to respond to the exceptional event. To `<propagate>` an exception occurrence is to raise it again in another context; that is, to fail to respond to the exceptional event in the present context.

Dynamic Semantics

2

Within a given task, if the execution of construct `<a>` is defined by this International Standard to consist (in part) of the execution of construct ``, then while `` is executing, the execution of `<a>` is said to `<dynamically enclose>` the execution of ``. The `<innermost dynamically enclosing>` execution of a given execution is the dynamically enclosing execution that started most recently.

3

When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is `<abandoned>`; that is, any portions of the execution that have not yet taken place are not performed. The construct is first completed, and then left, as explained in Section 8.6.1 [7.6.1], page 299. Then:

4

- If the construct is a `task_body`, the exception does not propagate further;

5

- If the construct is the `sequence_of_statements` of a `handled_sequence_of_statements` that has a handler with a choice covering the exception, the occurrence is handled by that handler;

6

- Otherwise, the occurrence is `<propagated>` to the innermost dynamically enclosing execution, which means that the occurrence is raised again in that context.

7

When an occurrence is `<handled>` by a given handler, the `choice_parameter_specification`, if any, is first elaborated, which creates the choice parameter and initializes it to the occurrence. Then, the `sequence_of_statements` of the handler is executed; this execution replaces the abandoned portion of the execution of the `sequence_of_statements`.

NOTES

8

1 Note that exceptions raised in a `declarative_part` of a body are not handled by the handlers of the `handled_sequence_of_statements` (see [S0247], page 420) of that body.

12.4.1 11.4.1 The Package Exceptions

Static Semantics

1

The following language-defined library package exists:

2/2

```
with Ada.Streams;
package Ada.Exceptions is
  pragma Preelaborate(Exceptions);
  type
  Exception_Id is private;
    pragma Preelaborable_Initialization(Exception_Id);

  Null_Id : constant Exception_Id;
  function
  Exception_Name(Id : Exception_Id) return String;
  function
  Wide_Exception_Name(Id : Exception_Id) return Wide_String;
  function
  Wide_Wide_Exception_Name(Id : Exception_Id)
    return Wide_Wide_String;
```

3/2

```
type
```

```

Exception_Occurrence is limited private;
  pragma Preelaborable_Initialization(Exception_Occurrence);
  type
Exception_Occurrence_Access is access all Exception_Occurrence;

Null_Occurrence : constant Exception_Occurrence;

```

4/2

```

  procedure
Raise_Exception(E : in Exception_Id;
                Message : in String := "");
  pragma No_Return(Raise_Exception);
  function
Exception_Message(X : Exception_Occurrence) return String;
  procedure
Reraise_Occurrence(X : in Exception_Occurrence);

```

5/2

```

  function
Exception_Identity(X : Exception_Occurrence)
  return Exception_Id;
  function
Exception_Name(X : Exception_Occurrence) return String;
  --< Same as Exception_Name(Exception_Identity(X)).>
  function
Wide_Exception_Name(X : Exception_Occurrence)
  return Wide_String;
  --< Same as Wide_Exception_Name(Exception_Identity(X)).>
  function
Wide_Wide_Exception_Name(X : Exception_Occurrence)
  return Wide_Wide_String;
  --< Same as Wide_Wide_Exception_Name(Exception_Identity(X)).>
  function
Exception_Information(X : Exception_Occurrence) return String;

```

6/2

```

  procedure Save_Occurrence(Target : out Exception_Occurrence;
                           Source : in Exception_Occurrence);
  function
Save_Occurrence(Source : Exception_Occurrence)
  return Exception_Occurrence_Access;

```

6.1/2

```

  procedure Read_Exception_Occurrence
  (Stream : not null access Ada.Streams.Root_Stream_Type'Class;

```



```

        Item : out Exception_Occurrence);
procedure Write_Exception_Occurrence
  (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in Exception_Occurrence);

```

6.2/2

```

for Exception_Occurrence'Read use Read_Exception_Occurrence;
for Exception_Occurrence'Write use Write_Exception_Occurrence;

```

6.3/2

```

private
  ... --< not specified by the language>
end Ada.Exceptions;

```

7

Each distinct exception is represented by a distinct value of type `Exception_Id`. `Null_Id` does not represent any exception, and is the default initial value of type `Exception_Id`. Each occurrence of an exception is represented by a value of type `Exception_Occurrence`. `Null_Occurrence` does not represent any exception occurrence, and is the default initial value of type `Exception_Occurrence`.

8/1

For a prefix `E` that denotes an exception, the following attribute is defined:

9

`E'Identity`

`E'Identity` returns the unique identity of the exception. The type of this attribute is `Exception_Id`.

10/2

`Raise_Exception` raises a new occurrence of the identified exception.

10.1/2

`Exception_Message` returns the message associated with the given `Exception_Occurrence`. For an occurrence raised by a call to `Raise_Exception`, the message is the `Message` parameter passed to `Raise_Exception`. For the occurrence raised by a `raise_statement` with an `<exception_>name` and a `<string_>expression`, the message is the `<string_>expression`. For the occurrence raised by a `raise_statement` with an `<exception_>name` but without a `<string_>expression`, the message is a string giving implementation-defined information about the exception occurrence. In all cases, `Exception_Message` returns a string with lower bound 1.

10.2/2

`Reraise_Occurrence` reraises the specified exception occurrence.

11

`Exception_Identity` returns the identity of the exception of the occurrence.

12/2

The `Wide_Wide_Exception_Name` functions return the full expanded name of the exception, in upper case, starting with a root library unit. For an exception declared immediately within package `Standard`, the `defining_identifier` (see [S0022], page 49) is returned. The result is implementation defined if the exception is declared within an unnamed `block_statement`.

12.1/2

The `Exception_Name` functions (respectively, `Wide_Exception_Name`) return the same sequence of graphic characters as that defined for `Wide_Wide_Exception_Name`, if all the graphic characters are defined in `Character` (respectively, `Wide_Character`); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by `Wide_Wide_Exception_Name` for the same value of the argument.

12.2/2

The string returned by the `Exception_Name`, `Wide_Exception_Name`, and `Wide_Wide_Exception_Name` functions has lower bound 1.

13/2

`Exception_Information` returns implementation-defined information about the exception occurrence. The returned string has lower bound 1.

14/2

`Reraise_Occurrence` has no effect in the case of `Null_Occurrence`. `Raise_Exception` and `Exception_Name` raise `Constraint_Error` for a `Null_Id`. `Exception_Message`, `Exception_Name`, and `Exception_Information` raise `Constraint_Error` for a `Null_Occurrence`. `Exception_Identity` applied to `Null_Occurrence` returns `Null_Id`.

15

The `Save_Occurrence` procedure copies the `Source` to the `Target`. The `Save_Occurrence` function uses an allocator of type `Exception_Occurrence_Access` to create a new object, copies the `Source` to this new object, and returns an access value designating this new object; the result may be deallocated using an instance of `Unchecked_Deallocation`.

15.1/2

`Write_Exception_Occurrence` writes a representation of an exception occurrence to a stream; `Read_Exception_Occurrence` reconstructs an exception occurrence from a stream (including one written in a different partition).

Implementation Requirements

16/2

<This paragraph was deleted.>

Implementation Permissions

17

An implementation of `Exception_Name` in a space-constrained environment may return the `defining_identifier` (see [S0022], page 49) instead of the full expanded name.

18

The string returned by `Exception_Message` may be truncated (to no less than 200 characters) by the `Save_Occurrence` procedure (not the function), the `Reraise_Occurrence` procedure, and the re-raise statement.

Implementation Advice

19

Exception_Message (by default) and Exception_Information should produce information useful for debugging. Exception_Message should be short (about one line), whereas Exception_Information can be long. Exception_Message should not include the Exception_Name. Exception_Information should include both the Exception_Name and the Exception_Message.

12.4.2 11.4.2 Pragmas Assert and Assertion_Policy

1/2

Pragma Assert is used to assert the truth of a Boolean expression at any point within a sequence of declarations or statements. Pragma Assertion_Policy is used to control whether such assertions are to be ignored by the implementation, checked at run-time, or handled in some implementation-defined manner.

Syntax

2/2

The form of a pragma Assert is as follows:

3/2

```
pragma Assert([Check ==>] <boolean->expression[, [Message ==>]
<string->expression]);
```

4/2

A pragma Assert is allowed at the place where a declarative_item or a statement is allowed.

5/2

The form of a pragma Assertion_Policy is as follows:

6/2

```
pragma Assertion_Policy(<policy->identifier);
```

7/2

A pragma Assertion_Policy is a configuration pragma.

Name Resolution Rules

8/2

The expected type for the <boolean->expression of a pragma Assert is any boolean type. The expected type for the <string->expression of a pragma Assert is type String.

Legality Rules

9/2

The <policy->identifier of a pragma Assertion_Policy shall be either Check, Ignore, or an implementation-defined identifier.

Static Semantics

10/2

A pragma `Assertion.Policy` is a configuration pragma that specifies the assertion policy in effect for the compilation units to which it applies. Different policies may apply to different compilation units within the same partition. The default assertion policy is implementation-defined.

11/2

The following language-defined library package exists:

12/2

```
package Ada.Assertions is
  pragma Pure(Assertions);
```

13/2

```
  Assertion_Error : exception;
```

14/2

```
  procedure Assert(Check : in Boolean);
  procedure Assert(Check : in Boolean; Message : in String);
```

15/2

```
end Ada.Assertions;
```

16/2

A compilation unit containing a pragma `Assert` has a semantic dependence on the `Assertions` library unit.

17/2

The assertion policy that applies to a generic unit also applies to all its instances.

Dynamic Semantics

18/2

An assertion policy specifies how a pragma `Assert` is interpreted by the implementation. If the assertion policy is `Ignore` at the point of a pragma `Assert`, the pragma is ignored. If the assertion policy is `Check` at the point of a pragma `Assert`, the elaboration of the pragma consists of evaluating the boolean expression, and if the result is `False`, evaluating the `Message` argument, if any, and raising the exception `Assertions.Assertion_Error`, with a message if the `Message` argument is provided.

19/2

Calling the procedure `Assertions.Assert` without a `Message` parameter is equivalent to:

20/2

```
if Check = False then
  raise Ada.Assertions.Assertion_Error;
end if;
```

21/2

Calling the procedure `Assertions.Assert` with a `Message` parameter is equivalent to:

22/2

```
if Check = False then
  raise Ada.Assertions.Assertion_Error with Message;
end if;
```

23/2

The procedures `Assertions.Assert` have these effects independently of the assertion policy in effect.

Implementation Permissions

24/2

`Assertion_Error` may be declared by renaming an implementation–defined exception from another package.

25/2

Implementations may define their own assertion policies.

NOTES

26/2

2 Normally, the boolean expression in a pragma `Assert` should not call functions that have significant side–effects when the result of the expression is `True`, so that the particular assertion policy in effect will not affect normal operation of the program.

12.4.3 11.4.3 Example of Exception Handling

Examples

1

Exception handling may be used to separate the detection of an error from the response to that error:

2/2

```
package File_System is
  type File_Handle is limited private;

  File_Not_Found : exception;
  procedure Open(F : in out File_Handle; Name : String);
  --< raises File_Not_Found if named file does not exist>

  End_Of_File : exception;
  procedure Read(F : in out File_Handle; Data : out Data_Type);
  --< raises End_Of_File if the file is not open>
```

4

5

```
    ...  
end File_System;
```

6/2

```
package body File_System is  
  procedure Open(F : in out File_Handle; Name : String) is  
  begin  
    if File_Exists(Name) then  
      ...  
    else  
      raise File_Not_Found with "File not found: " & Name & ".";■  
    end if;  
  end Open;
```

7

```
  procedure Read(F : in out File_Handle; Data : out Data_Type) is  
  begin  
    if F.Current_Position <= F.Last_Position then  
      ...  
    else  
      raise End_Of_File;  
    end if;  
  end Read;
```

8

```
    ...
```

9

```
end File_System;
```

10

```
with Ada.Text_IO;  
with Ada.Exceptions;  
with File_System; use File_System;  
use Ada;  
procedure Main is  
begin  
  ... --< call operations in File_System>  
exception  
  when End_Of_File =>  
    Close(Some_File);  
  when Not_Found_Error : File_Not_Found =>
```

```

        Text_IO.Put_Line(Exceptions.Exception_Message(Not_Found_Error));
when The_Error : others =>
    Text_IO.Put_Line("Unknown error:");
    if Verbosity_Desired then
        Text_IO.Put_Line(Exceptions.Exception_Information(The_Error));
    else
        Text_IO.Put_Line(Exceptions.Exception_Name(The_Error));
        Text_IO.Put_Line(Exceptions.Exception_Message(The_Error));
    end if;
    raise;
end Main;

```

11

In the above example, the `File_System` package contains information about detecting certain exceptional situations, but it does not specify how to handle those situations. Procedure `Main` specifies how to handle them; other clients of `File_System` might have different handlers, even though the exceptional situations arise from the same basic causes.

12.5 11.5 Suppressing Checks

1/2

<Checking pragmas> give instructions to an implementation on handling language-defined checks. A pragma `Suppress` gives permission to an implementation to omit certain language-defined checks, while a pragma `Unsuppress` revokes the permission to omit checks..

2

A <language-defined check> (or simply, a "check") is one of the situations defined by this International Standard that requires a check to be made at run time to determine whether some condition is true. A check <fails> when the condition being checked is false, causing an exception to be raised.

Syntax

3/2

The forms of checking pragmas are as follows:

4/2

```
pragma Suppress(identifier);
```

4.1/2

```
pragma Unsuppress(identifier);
```

5/2

A checking pragma is allowed only immediately within a declarative-part, immediately within a package-specification (see [S0174], page 279), or as a configuration pragma.

Legality Rules

6/2

The identifier shall be the name of a check.

7/2

<This paragraph was deleted.>

Static Semantics

7.1/2

A checking pragma applies to the named check in a specific region, and applies to all entities in that region. A checking pragma given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a checking pragma given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies.

7.2/2

If a checking pragma applies to a generic instantiation, then the checking pragma also applies to the instance. If a checking pragma applies to a call to a subprogram that has a pragma Inline applied to it, then the checking pragma also applies to the inlined subprogram body.

8/2

A pragma Suppress gives permission to an implementation to omit the named check (or every check in the case of All_Checks) for any entities to which it applies. If permission has been given to suppress a given check, the check is said to be <suppressed>.

8.1/2

A pragma Unsuppress revokes the permission to omit the named check (or every check in the case of All_Checks) given by any pragma Suppress that applies at the point of the pragma Unsuppress. The permission is revoked for the region to which the pragma Unsuppress applies. If there is no such permission at the point of a pragma Unsuppress, then the pragma has no effect. A later pragma Suppress can renew the permission.

9

The following are the language-defined checks:

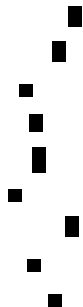
10

- The following checks correspond to situations in which the exception Constraint_Error is raised upon failure.

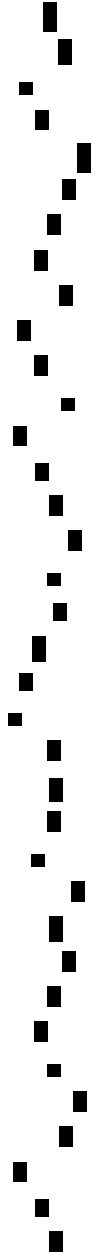
11/2

Access_Check

When
eval-
u-
at-
ing
a
deref-
er-
ence

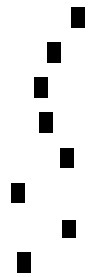


(explicit or implicit), check that the value of the name is not null. When converting to a subtype that excludes null, check that the converted value is not null.

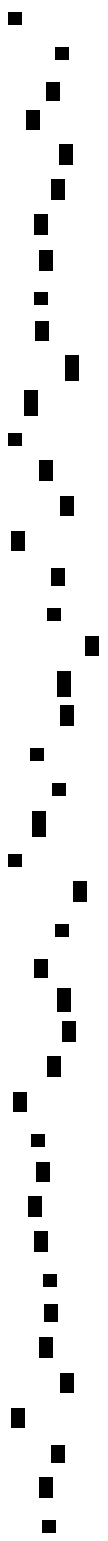


12
Discriminant_Check

Check that the discriminants of

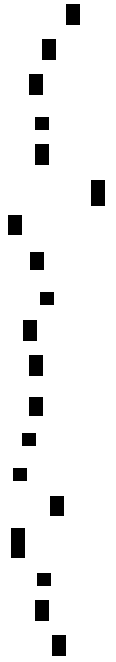


a composite value have the values imposed by a discriminant constraint. Also, when accessing a record component, check that it exists for the current discriminant values.

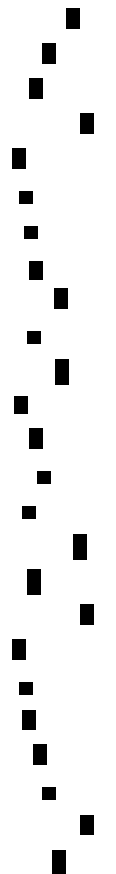


14
Index_Check

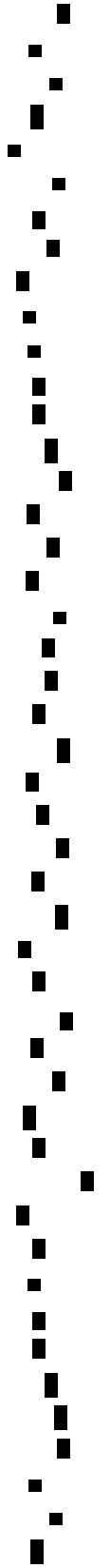
Check
that
the
sec-
ond
operand
is
not
zero
for
the
op-
er-
a-
tions
/,
rem
and
mod.



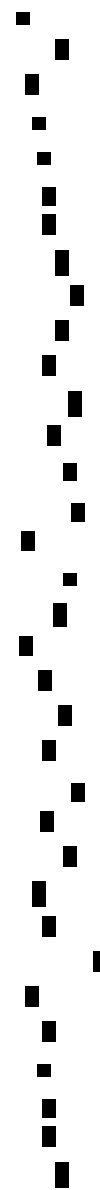
Check
that
the
bounds
of
an
ar-
ray
value
are
equal
to
the
cor-
re-
spond-
ing
bounds
of
an
in-
dex
con-
straint.
Also,



when
ac-
cess-
ing
a
com-
po-
nent
of
an
ar-
ray
ob-
ject,
check
for
each
di-
men-
sion
that
the
given
in-
dex
value
be-
longs
to
the
range
de-
fined
by
the
bounds
of
the
ar-
ray
ob-
ject.
Also,
when
ac-
cess-
ing

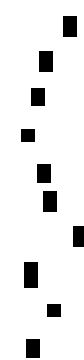


a slice of an array object, check that the given discrete range is compatible with the range defined by the bounds of the array object.

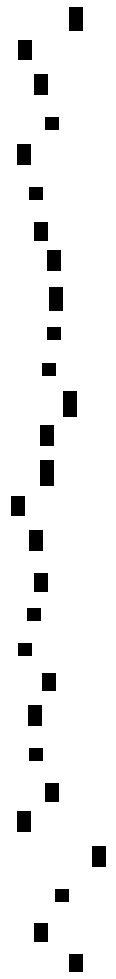


15
Length_Check

Check that two arrays have matching components.

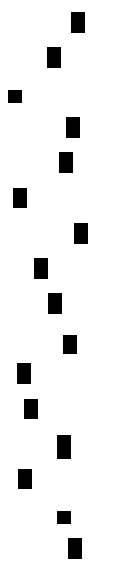


nents,
in
the
case
of
ar-
ray
sub-
type
con-
ver-
sions,
and
log-
i-
cal
op-
er-
a-
tors
for
ar-
rays
of
boolean
com-
po-
nents.

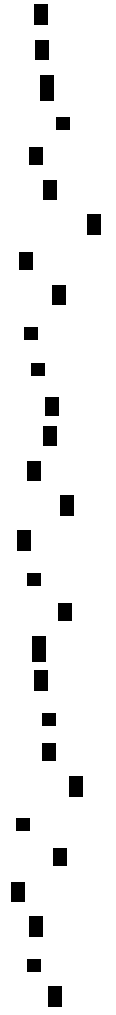


16
Overflow_Check

Check
that
a
scalar
value
is
within
the
base
range
of
its
type,
in
cases
where

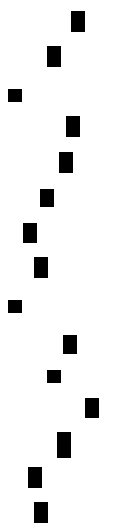


the
im-
ple-
men-
ta-
tion
chooses
to
raise
an
ex-
cep-
tion
in-
stead
of
re-
turn-
ing
the
cor-
rect
math-
e-
mat-
i-
cal
re-
sult.

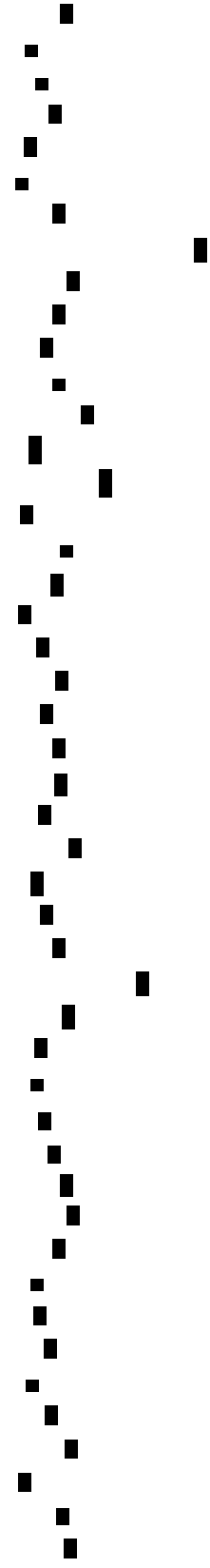


17
Range_Check

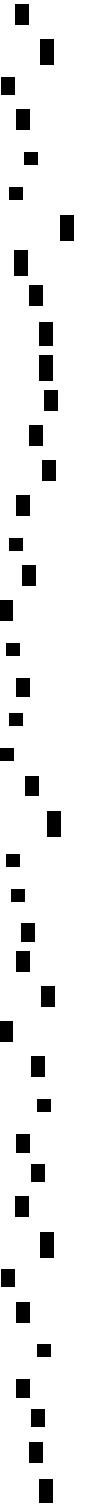
Check
that
a
scalar
value
sat-
is-
fies
a
range
con-
straint.
Also,
for
the



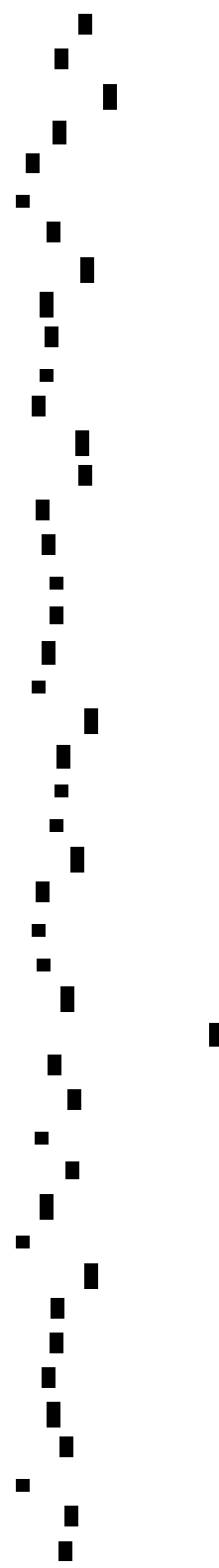
elab-
o-
ra-
tion
of
a
sub-
type_indication,
check
that
the
con-
straint
(if
present)
is
com-
pat-
i-
ble
with
the
sub-
type
de-
noted
by
the
sub-
type_mark.
Also,
for
an
ag-
gre-
gate,
check
that
an
in-
dex
or
dis-
crim-
i-
nant
value



be-
long-
to
the
cor-
re-
spond-
ing
sub-
type.
Also,
check
that
when
the
re-
sult
of
an
op-
er-
a-
tion
yields
an
ar-
ray,
the
value
of
each
com-
po-
nent
be-
long-
to
the
com-
po-
nent
sub-
type.



Check
that
operand
tags
in
a
dis-
patch-
ing
call
are
all
equal.
Check
for
the
cor-
rect
tag
on
tagged
type
con-
ver-
sions,
for
an
as-
sign-
ment_statement,
and
when
re-
turn-
ing
a
tagged
lim-
ited
ob-
ject
from
a
func-
tion.



- The following checks correspond to situations in which the exception `Program_Error` is raised upon failure.

19.1/2

`Accessibility_Check`

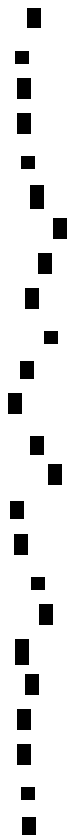
Check
the
ac-
ces-
si-
bil-
ity
level
of
an
en-
tity
or
view.



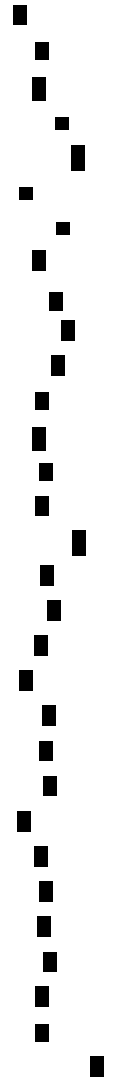
19.2/2

`Allocation_Check`

For
an
al-
lo-
ca-
tor,
check
that
the
mas-
ter
of
any
tasks
to
be
cre-
ated
by
the
al-
lo-
ca-
tor

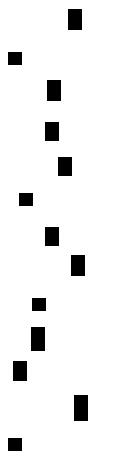


is
not
yet
com-
pleted
or
some
de-
pen-
dents
have
not
yet
ter-
mi-
nated,
and
that
the
fi-
nal-
iza-
tion
of
the
col-
lec-
tion
has
not
started.

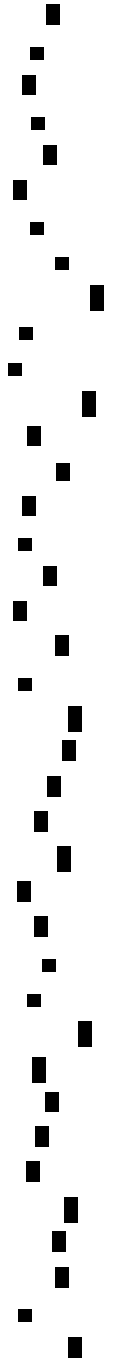


20
Elaboration_Check

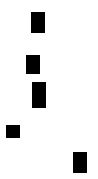
When
a
sub-
pro-
gram
or
pro-
tected
en-
try
is
called,
a



task
ac-
ti-
va-
tion
is
ac-
com-
plished,
or
a
generic
in-
stan-
ti-
a-
tion
is
elab-
o-
rated,
check
that
the
body
of
the
cor-
re-
spond-
ing
unit
has
al-
ready
been
elab-
o-
rated.



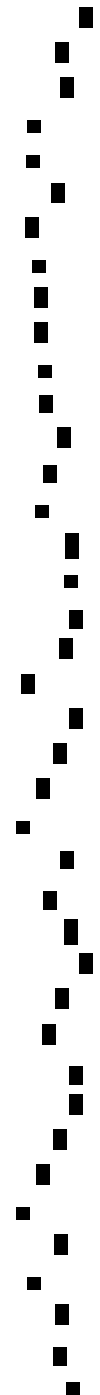
<This
para-
graph
was
deleted.>



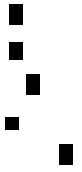
- The following check corresponds to situations in which the exception `Storage_Error` is raised upon failure.

`Storage_Check`

Check
that
eval-
u-
a-
tion
of
an
al-
lo-
ca-
tor
does
not
re-
quire
more
space
than
is
avail-
able
for
a
stor-
age
pool.
Check
that
the
space
avail-
able
for
a
task
or
sub-
pro-
gram



has
not
been
ex-
ceeded.



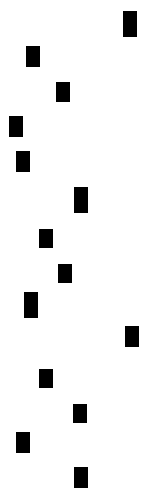
24

- The following check corresponds to all situations in which any predefined exception is raised.

25

All_Checks

Represents
the
union
of
all
checks;
sup-
press-
ing
All_Checks
sup-
presses
all
checks.



Erroneous Execution

26

If a given check has been suppressed, and the corresponding error situation occurs, the execution of the program is erroneous.

Implementation Permissions

27/2

An implementation is allowed to place restrictions on checking pragmas, subject only to the requirement that pragma Unsuppress shall allow any check names supported by pragma Suppress. An implementation is allowed to add additional check names, with implementation-defined semantics. When Overflow_Check has been suppressed, an implementation may also suppress an unspecified subset of the Range_Checks.

27.1/2

An implementation may support an additional parameter on pragma Unsuppress similar to the one allowed for pragma Suppress (see Section 23.10 [J.10], page 1175). The meaning of such a parameter is implementation-defined.

Implementation Advice

28

The implementation should minimize the code executed for checks that have been suppressed.

NOTES

29

3 There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress should be used only for efficiency reasons.

29.1/2

4 It is possible to give both a pragma Suppress and Unsuppress for the same check immediately within the same declarative_part. In that case, the last pragma given determines whether or not the check is suppressed. Similarly, it is possible to resuppress a check which has been unsuppressed by giving a pragma Suppress in an inner declarative region.

Examples

30/2

<Examples of suppressing and unsuppressing checks:>

31/2

```
pragma Suppress(Index_Check);  
pragma Unsuppress(Overflow_Check);
```

12.6 11.6 Exceptions and Optimization

1

This clause gives permission to the implementation to perform certain "optimizations" that do not necessarily preserve the canonical semantics.

Dynamic Semantics

2

The rest of this International Standard (outside this clause) defines the <canonical semantics> of the language. The canonical semantics of a given (legal) program determines a set of possible external effects that can result from the execution of the program with given inputs.

3

As explained in Section 2.1.3 [1.1.3], page 23, "Section 2.1.3 [1.1.3], page 23, Conformity of an Implementation with the Standard", the external effect of a program is defined in terms of its interactions with its external environment. Hence, the implementation can perform any internal actions whatsoever, in any order or in parallel, so long as the external effect of the execution of the program is one that is allowed by the canonical semantics, or by the rules of this clause.

Implementation Permissions

4

The following additional permissions are granted to the implementation:

5

- An implementation need not always raise an exception when a language-defined check fails. Instead, the operation that failed the check can simply yield an <undefined

result>. The exception need be raised by the implementation only if, in the absence of raising it, the value of this undefined result would have some effect on the external interactions of the program. In determining this, the implementation shall not presume that an undefined result has a value that belongs to its subtype, nor even to the base range of its type, if scalar. Having removed the raise of the exception, the canonical semantics will in general allow the implementation to omit the code for the check, and some or all of the operation itself.

6

- If an exception is raised due to the failure of a language–defined check, then upon reaching the corresponding `exception_handler` (or the termination of the task, if none), the external interactions that have occurred need reflect only that the exception was raised somewhere within the execution of the `sequence_of_statements` with the handler (or the `task_body`), possibly earlier (or later if the interactions are independent of the result of the checked operation) than that defined by the canonical semantics, but not within the execution of some abort–deferred operation or <independent> subprogram that does not dynamically enclose the execution of the construct whose check failed. An independent subprogram is one that is defined outside the library unit containing the construct whose check failed, and has no `Inline` pragma applied to it. Any assignment that occurred outside of such abort–deferred operations or independent subprograms can be disrupted by the raising of the exception, causing the object or its parts to become abnormal, and certain subsequent uses of the object to be erroneous, as explained in Section 14.9.1 [13.9.1], page 522.

NOTES

7

5 The permissions granted by this clause can have an effect on the semantics of a program only if the program fails a language–defined check.

13 12 Generic Units

1

A <generic unit> is a program unit that is either a generic subprogram or a generic package. A generic unit is a <template>, which can be parameterized, and from which corresponding (nongeneric) subprograms or packages can be obtained. The resulting program units are said to be <instances> of the original generic unit.

2

A generic unit is declared by a `generic_declaration`. This form of declaration has a `generic_formal_part` (see [S0255], page 450) declaring any generic formal parameters. An instance of a generic unit is obtained as the result of a `generic_instantiation` with appropriate generic actual parameters for the generic formal parameters. An instance of a generic subprogram is a subprogram. An instance of a generic package is a package.

3

Generic units are templates. As templates they do not have the properties that are specific to their nongeneric counterparts. For example, a generic subprogram can be instantiated but it cannot be called. In contrast, an instance of a generic subprogram is a (nongeneric) subprogram; hence, this instance can be called but it cannot be used to produce further instances.

13.1 12.1 Generic Declarations

1

A `generic_declaration` declares a generic unit, which is either a generic subprogram or a generic package. A `generic_declaration` includes a `generic_formal_part` declaring any generic formal parameters. A generic formal parameter can be an object; alternatively (unlike a parameter of a subprogram), it can be a type, a subprogram, or a package.

Syntax

2

```
generic_declaration ::= generic_subprogram_declaration | generic_package_declaration
```

3

```
generic_subprogram_declaration ::=  
    generic_formal_part subprogram_specification;
```

4

```
generic_package_declaration ::=  
    generic_formal_part package_specification;
```

5

```
generic_formal_part ::= generic {generic_formal_parameter_declaration | use_clause}
```

6

```
generic_formal_parameter_declaration ::=
    formal_object_declaration
  | formal_type_declaration
  | formal_subprogram_declaration
  | formal_package_declaration
```

7

The only form of `subtype_indication` allowed within a `generic_formal_part` is a `subtype_mark` (that is, the `subtype_indication` shall not include an explicit constraint). The defining name of a generic subprogram shall be an identifier (not an `operator_symbol`).

Static Semantics

8/2

A `generic_declaration` declares a generic unit — a generic package, generic procedure, or generic function, as appropriate.

9

An entity is a `<generic formal>` entity if it is declared by a `generic_formal_parameter_declaration`. "Generic formal," or simply "formal," is used as a prefix in referring to objects, subtypes (and types), functions, procedures and packages, that are generic formal entities, as well as to their respective declarations. Examples: "generic formal procedure" or a "formal integer type declaration."

Dynamic Semantics

10

The elaboration of a `generic_declaration` has no effect.

NOTES

11

1 Outside a generic unit a name that denotes the `generic_declaration` denotes the generic unit. In contrast, within the declarative region of the generic unit, a name that denotes the `generic_declaration` denotes the current instance.

12

2 Within a `generic_subprogram_body`, the name of this program unit acts as the name of a subprogram. Hence this name can be overloaded, and it can appear in a recursive call of the current instance. For the same reason, this name cannot appear after the reserved word `new` in a (recursive) `generic_instantiation`.

13

3 A `default_expression` or `default_name` appearing in a `generic_formal_part` is not evaluated during elaboration of the `generic_formal_part`; instead, it is evaluated when used. (The

usual visibility rules apply to any name used in a default: the denoted declaration therefore has to be visible at the place of the expression.)

Examples

14

<Examples of generic formal parts:>

15

```
generic      --< parameterless >
```

16

```
generic
  Size : Natural;  --< formal object >
```

17

```
generic
  Length : Integer := 200;          --< formal object with a default expression>
```

18

```
  Area   : Integer := Length*Length; --< formal object with a default expression>
```

19

```
generic
  type Item is private;           --< formal type>
  type Index is (<>);              --< formal type>
  type Row   is array(Index range <>) of Item; --< formal type>
  with function "<"(X, Y : Item) return Boolean;  --< formal subprogram >■
```

20

<Examples of generic declarations declaring generic subprograms Exchange and Squaring:>

21

```
generic
  type Elem is private;
  procedure Exchange(U, V : in out Elem);
```

22

```
generic
  type Item is private;
  with function "*" (U, V : Item) return Item is <>;
  function Squaring(X : Item) return Item;
```

23

<Example of a generic declaration declaring a generic package:>

```

generic
  type Item  is private;
  type Vector is array (Positive range <>) of Item;
  with function Sum(X, Y : Item) return Item;
package On_Vectors is
  function Sum (A, B : Vector) return Vector;
  function Sigma(A : Vector) return Item;
  Length_Error : exception;
end On_Vectors;

```

13.2 12.2 Generic Bodies

1

The body of a generic unit (a <generic body>) is a template for the instance bodies. The syntax of a generic body is identical to that of a nongeneric body.

Dynamic Semantics

2

The elaboration of a generic body has no other effect than to establish that the generic unit can from then on be instantiated without failing the Elaboration_Check. If the generic body is a child of a generic package, then its elaboration establishes that each corresponding declaration nested in an instance of the parent (see Section 11.1.1 [10.1.1], page 394) can from then on be instantiated without failing the Elaboration_Check.

NOTES

3

4 The syntax of generic subprograms implies that a generic subprogram body is always the completion of a declaration.

Examples

4

<Example of a generic procedure body:>

5

```

procedure Exchange(U, V : in out Elem) is --< see Section 13.1 [12.1],
page 450>
  T : Elem; --< the generic formal type>
begin
  T := U;
  U := V;
  V := T;
end Exchange;

```

6

<Example of a generic function body:>

7

```
function Squaring(X : Item) return Item is --< see Section 13.1
[12.1], page 450>
begin
  return X*X; --< the formal operator "*">
end Squaring;
```

8

<Example of a generic package body:>

9

```
package body On_Vectors is --< see Section 13.1 [12.1], page 450>
```

10

```
function Sum(A, B : Vector) return Vector is
  Result : Vector(A'Range); --< the formal type Vector>
  Bias   : constant Integer := B'First - A'First;
begin
  if A'Length /= B'Length then
    raise Length_Error;
  end if;
```

11

```
  for N in A'Range loop
    Result(N) := Sum(A(N), B(N + Bias)); --< the formal function Sum>
  end loop;
  return Result;
end Sum;
```

12

```
function Sigma(A : Vector) return Item is
  Total : Item := A(A'First); --< the formal type Item>
begin
  for N in A'First + 1 .. A'Last loop
    Total := Sum(Total, A(N)); --< the formal function Sum>
  end loop;
  return Total;
end Sigma;
end On_Vectors;
```

13.3 12.3 Generic Instantiation

1

An instance of a generic unit is declared by a generic_instantiation.

Syntax

2/2

```
generic_instantiation ::=  
    package defining_program_unit_name is  
        new <generic_package_>name [generic_actual_part];  
    | [overriding_indicator]  
    procedure defining_program_unit_name is  
        new <generic_procedure_>name [generic_actual_part];  
    | [overriding_indicator]  
    function defining_designator is  
        new <generic_function_>name [generic_actual_part];
```

3

```
generic_actual_part ::=  
    (generic_association {, generic_association})
```

4

```
generic_association ::=  
    [<generic_formal_parameter_>selector_name =>] explicit_generic_actual_parameter
```

5

```
explicit_generic_actual_parameter ::= expression | <variable_>name  
    | <subprogram_>name | <entry_>name | subtype_mark  
    | <package_instance_>name
```

6

A generic_association is <named> or <positional> according to whether or not the <generic_formal_parameter_>selector_name (see [S0099], page 184) is specified. Any positional associations shall precede any named associations.

7/2

The <generic actual parameter> is either the explicit_generic_actual_parameter given in a generic_association (see [S0259], page 455) for each formal, or the corresponding default_expression (see [S0063], page 123) or default_name (see [S0279], page 471) if no generic_association (see [S0259], page 455) is given for the formal. When the meaning is clear from context, the term "generic actual," or simply "actual," is used as a synonym for "generic actual parameter" and also for the view denoted by one, or the value of one.

Legality Rules

8

In a generic_instantiation for a particular kind of program unit (package, procedure, or function), the name shall denote a generic unit of the corresponding kind (generic package, generic procedure, or generic function, respectively).

9

The <generic_formal_parameter_>selector_name of a generic_association shall denote a

generic_formal_parameter_declaration of the generic unit being instantiated. If two or more formal subprograms have the same defining name, then named associations are not allowed for the corresponding actuals.

10

A generic_instantiation shall contain at most one generic_association for each formal. Each formal without an association shall have a default_expression or subprogram_default.

11

In a generic unit Legality Rules are enforced at compile time of the generic_declaration and generic body, given the properties of the formals. In the visible part and formal part of an instance, Legality Rules are enforced at compile time of the generic_instantiation, given the properties of the actuals. In other parts of an instance, Legality Rules are not enforced; this rule does not apply when a given rule explicitly specifies otherwise.

Static Semantics

12

A generic_instantiation declares an instance; it is equivalent to the instance declaration (a package_declaration (see [S0173], page 279) or subprogram_declaration (see [S0148], page 255)) immediately followed by the instance body, both at the place of the instantiation.

13

The instance is a copy of the text of the template. Each use of a formal parameter becomes (in the copy) a use of the actual, as explained below. An instance of a generic package is a package, that of a generic procedure is a procedure, and that of a generic function is a function.

14

The interpretation of each construct within a generic declaration or body is determined using the overloading rules when that generic declaration or body is compiled. In an instance, the interpretation of each (copied) construct is the same, except in the case of a name that denotes the generic_declaration or some declaration within the generic unit; the corresponding name in the instance then denotes the corresponding copy of the denoted declaration. The overloading rules do not apply in the instance.

15

In an instance, a generic_formal_parameter_declaration declares a view whose properties are identical to those of the actual, except as specified in Section 13.4 [12.4], page 458, "Section 13.4 [12.4], page 458, Formal Objects" and Section 13.6 [12.6], page 470, "Section 13.6 [12.6], page 470, Formal Subprograms". Similarly, for a declaration within a generic_formal_parameter_declaration, the corresponding declaration in an instance declares a view whose properties are identical to the corresponding declaration within the declaration of the actual.

16

Implicit declarations are also copied, and a name that denotes an implicit declaration in the generic denotes the corresponding copy in the instance. However, for a type declared within the visible part of the generic, a whole new set of primitive subprograms is implicitly declared for use outside the instance, and may differ from the copied set if the properties of the type in some way depend on the properties of some actual type specified in the

instantiation. For example, if the type in the generic is derived from a formal private type, then in the instance the type will inherit subprograms from the corresponding actual type.

17

These new implicit declarations occur immediately after the type declaration in the instance, and override the copied ones. The copied ones can be called only from within the instance; the new ones can be called only from outside the instance, although for tagged types, the body of a new one can be executed by a call to an old one.

18

In the visible part of an instance, an explicit declaration overrides an implicit declaration if they are homographs, as described in Section 9.3 [8.3], page 308. On the other hand, an explicit declaration in the private part of an instance overrides an implicit declaration in the instance, only if the corresponding explicit declaration in the generic overrides a corresponding implicit declaration in the generic. Corresponding rules apply to the other kinds of overriding described in Section 9.3 [8.3], page 308.

Post-Compilation Rules

19

Recursive generic instantiation is not allowed in the following sense: if a given generic unit includes an instantiation of a second generic unit, then the instance generated by this instantiation shall not include an instance of the first generic unit (whether this instance is generated directly, or indirectly by intermediate instantiations).

Dynamic Semantics

20

For the elaboration of a generic_instantiation, each generic_association is first evaluated. If a default is used, an implicit generic_association is assumed for this rule. These evaluations are done in an arbitrary order, except that the evaluation for a default actual takes place after the evaluation for another actual if the default includes a name that denotes the other one. Finally, the instance declaration and body are elaborated.

21

For the evaluation of a generic_association the generic actual parameter is evaluated. Additional actions are performed in the case of a formal object of mode in (see Section 13.4 [12.4], page 458).

NOTES

22

5 If a formal type is not tagged, then the type is treated as an untagged type within the generic body. Deriving from such a type in a generic body is permitted; the new type does not get a new tag value, even if the actual is tagged. Overriding operations for such a derived type cannot be dispatched to from outside the instance.

Examples

23

<Examples of generic instantiations (see Section 13.1 [12.1], page 450):>

24

```
procedure Swap is new Exchange(Elem => Integer);
```

```

procedure Swap is new Exchange(Character);    --< Swap is overloaded >
function Square is new Squaring(Integer);    --< "*" of Integer used by default>
function Square is new Squaring(Item => Matrix, "*" => Matrix_Product);
function Square is new Squaring(Matrix, Matrix_Product); --< same as previous

```

25

```

package Int_Vectors is new On_Vectors(Integer, Table, "+");

```

26

<Examples of uses of instantiated units:>

27

```

Swap(A, B);
A := Square(A);

```

28

```

T : Table(1 .. 5) := (10, 20, 30, 40, 50);
N : Integer := Int_Vectors.Sigma(T); --< 150 (see Section 13.2 [12.2],
page 453, "Section 13.2 [12.2], page 453, Generic Bodies" for the body of Sigma)>

```

29

```

use Int_Vectors;
M : Integer := Sigma(T); --< 150>

```

13.4 12.4 Formal Objects

1

A generic formal object can be used to pass a value or variable to a generic unit.

Syntax

2/2

```

formal_object_declaration ::=
  defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression];

  defining_identifier_list : mode access_definition [:= default_expression];

```

Name Resolution Rules

3

The expected type for the default_expression, if any, of a formal object is the type of the formal object.

4

For a generic formal object of mode in, the expected type for the actual is the type of the formal.

5/2

For a generic formal object of mode in out, the type of the actual shall resolve to the

type determined by the `subtype_mark`, or for a `formal_object_declaration` with an `access_definition`, to a specific anonymous access type. If the anonymous access type is an `access-to-object` type, the type of the actual shall have the same designated type as that of the `access_definition`. If the anonymous access type is an `access-to-subprogram` type, the type of the actual shall have a designated profile which is type conformant with that of the `access_definition`. .

Legality Rules

6

If a generic formal object has a `default_expression`, then the mode shall be in (either explicitly or by default); otherwise, its mode shall be either in or in out.

7

For a generic formal object of mode in, the actual shall be an expression. For a generic formal object of mode in out, the actual shall be a name that denotes a variable for which renaming is allowed (see Section 9.5.1 [8.5.1], page 317).

8/2

In the case where the type of the formal is defined by an `access_definition`, the type of the actual and the type of the formal:

8.1/2

- shall both be `access-to-object` types with statically matching designated subtypes and with both or neither being `access-to-constant` types; or

8.2/2

- shall both be `access-to-subprogram` types with subtype conformant designated profiles.

8.3/2

For a `formal_object_declaration` with a `null_exclusion` or an `access_definition` that has a `null_exclusion`:

8.4/2

- if the actual matching the `formal_object_declaration` denotes the generic formal object of another generic unit `<G>`, and the instantiation containing the actual occurs within the body of `<G>` or within the body of a generic unit declared within the declarative region of `<G>`, then the declaration of the formal object of `<G>` shall have a `null_exclusion`;

8.5/2

- otherwise, the subtype of the actual matching the `formal_object_declaration` shall exclude null. In addition to the places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit.

Static Semantics

9/2

A `formal_object_declaration` declares a generic formal object. The default mode is `in`. For a formal object of mode `in`, the nominal subtype is the one denoted by the `subtype_mark` or `access_definition` in the declaration of the formal. For a formal object of mode `in out`, its type is determined by the `subtype_mark` or `access_definition` in the declaration; its nominal subtype is nonstatic, even if the `subtype_mark` denotes a static subtype; for a composite type, its nominal subtype is unconstrained if the first subtype of the type is unconstrained, even if the `subtype_mark` denotes a constrained subtype.

10/2

In an instance, a `formal_object_declaration` of mode `in` is a `<full constant declaration>` and declares a new stand-alone constant object whose initialization expression is the `actual`, whereas a `formal_object_declaration` of mode `in out` declares a view whose properties are identical to those of the `actual`.

Dynamic Semantics

11

For the evaluation of a generic association for a formal object of mode `in`, a constant object is created, the value of the `actual` parameter is converted to the nominal subtype of the formal object, and assigned to the object, including any value adjustment — see Section 8.6 [7.6], page 295.

NOTES

12

6 The constraints that apply to a generic formal object of mode `in out` are those of the corresponding generic `actual` parameter (not those implied by the `subtype_mark` that appears in the `formal_object_declaration`). Therefore, to avoid confusion, it is recommended that the name of a first subtype be used for the declaration of such a formal object.

13.5 12.5 Formal Types

1/2

A generic formal subtype can be used to pass to a generic unit a subtype whose type is in a certain category of types.

Syntax

2

```
formal_type_declaration ::=  
  type defining_identifier[discriminant_part] is formal_type_definition;
```

3/2

```
formal_type_definition ::=  
  formal_private_type_definition  
  | formal_derived_type_definition  
  | formal_discrete_type_definition
```

- | formal_signed_integer_type_definition
- | formal_modular_type_definition
- | formal_floating_point_definition
- | formal_ordinary_fixed_point_definition
- | formal_decimal_fixed_point_definition
- | formal_array_type_definition
- | formal_access_type_definition
- | formal_interface_type_definition

Legality Rules

4

For a generic formal subtype, the actual shall be a subtype_mark; it denotes the <(generic) actual subtype>.

Static Semantics

5

A formal_type_declaration declares a <(generic) formal type>, and its first subtype, the <(generic) formal subtype>.

6/2

The form of a formal_type_definition <determines a category (of types)> to which the formal type belongs. For a formal_private_type_definition the reserved words tagged and limited indicate the category of types (see Section 13.5.1 [12.5.1], page 462). For a formal_derived_type_definition the category of types is the derivation class rooted at the ancestor type. For other formal types, the name of the syntactic category indicates the category of types; a formal_discrete_type_definition defines a discrete type, and so on.

Legality Rules

7/2

The actual type shall be in the category determined for the formal.

Static Semantics

8/2

The formal type also belongs to each category that contains the determined category. The primitive subprograms of the type are as for any type in the determined category. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later immediately within the declarative region in which the type is declared according to the rules of Section 8.3.1 [7.3.1], page 287. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in Section 13.5.1 [12.5.1], page 462.

NOTES

9

7 Generic formal types, like all types, are not named. Instead, a name can denote a generic formal subtype. Within a generic unit,

a generic formal type is considered as being distinct from all other (formal or nonformal) types.

10

8 A `discriminant_part` is allowed only for certain kinds of types, and therefore only for certain kinds of generic formal types. See Section 4.7 [3.7], page 123.

Examples

11

<Examples of generic formal types:>

12

```
type Item is private;
type Buffer(Length : Natural) is limited private;
```

13

```
type Enum is (<>);
type Int is range <>;
type Angle is delta <>;
type Mass is digits <>;
```

14

```
type Table is array (Enum) of Item;
```

15

<Example of a generic formal part declaring a formal integer type:>

16

```
generic
  type Rank is range <>;
  First : Rank := Rank'First;
  Second : Rank := First + 1; --< the operator "+" of the type Rank >■
```

13.5.1 12.5.1 Formal Private and Derived Types

1/2

In its most general form, the category determined for a formal private type is all types, but it can be restricted to only nonlimited types or to only tagged types. The category determined for a formal derived type is the derivation class rooted at the ancestor type.

Syntax

2

```
formal_private_type_definition ::= [[abstract] tagged] [limited] private ■
```

3/2

formal_derived_type_definition ::=
[abstract] [limited | synchronized] new subtype_mark [[and interface_list]with private]
Legality Rules

4

If a generic formal type declaration has a known_discriminant_part, then it shall not include a default_expression for a discriminant.

5/2

The <ancestor subtype> of a formal derived type is the subtype denoted by the subtype_mark of the formal_derived_type_definition. For a formal derived type declaration, the reserved words with private shall appear if and only if the ancestor type is a tagged type; in this case the formal derived type is a private extension of the ancestor type and the ancestor shall not be a class-wide type. Similarly, an interface_list or the optional reserved words abstract or synchronized shall appear only if the ancestor type is a tagged type. The reserved word limited or synchronized shall appear only if the ancestor type and any progenitor types are limited types. The reserved word synchronized shall appear (rather than limited) if the ancestor type or any of the progenitor types are synchronized interfaces.

5.1/2

The actual type for a formal derived type shall be a descendant of the ancestor type and every progenitor of the formal type. If the reserved word synchronized appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

6

If the formal subtype is definite, then the actual subtype shall also be definite.

7

For a generic formal derived type with no discriminant_part:

8

- If the ancestor subtype is constrained, the actual subtype shall be constrained, and shall be statically compatible with the ancestor;

9

- If the ancestor subtype is an unconstrained access or composite subtype, the actual subtype shall be unconstrained.

10

- If the ancestor subtype is an unconstrained discriminated subtype, then the actual shall have the same number of discriminants, and each discriminant of the actual shall correspond to a discriminant of the ancestor, in the sense of Section 4.7 [3.7], page 123.

10.1/2

- If the ancestor subtype is an access subtype, the actual subtype shall exclude null if and only if the ancestor subtype excludes null.

11

The declaration of a formal derived type shall not have a `known_discriminant_part`. For a generic formal private type with a `known_discriminant_part`:

12

- The actual type shall be a type with the same number of discriminants.

13

- The actual subtype shall be unconstrained.

14

- The subtype of each discriminant of the actual type shall statically match the subtype of the corresponding discriminant of the formal type.

15

For a generic formal type with an `unknown_discriminant_part`, the actual may, but need not, have discriminants, and may be definite or indefinite.

Static Semantics

16/2

The category determined for a formal private type is as follows:

17/2

<Type Definition> <Determined Category>

limited private the category of all types

private the category of all nonlimited types

tagged limited private the category of all tagged types

tagged private the category of all nonlimited tagged types

18

The presence of the reserved word `abstract` determines whether the actual type may be abstract.

19

A formal private or derived type is a private or derived type, respectively. A formal derived tagged type is a private extension. A formal private or derived type is abstract if the reserved word `abstract` appears in its declaration.

20/2

If the ancestor type is a composite type that is not an array type, the formal type inherits components from the ancestor type (including discriminants if a new `discriminant_part` is not specified), as for a derived type defined by a `derived_type_definition` (see Section 4.4 [3.4], page 66, and Section 8.3.1 [7.3.1], page 287).

21/2

For a formal derived type, the predefined operators and inherited user-defined subprograms are determined by the ancestor type and any progenitor types, and are implicitly declared at the earliest place, if any, immediately within the declarative region in which the

formal type is declared, where the corresponding primitive subprogram of the ancestor or progenitor is visible (see Section 8.3.1 [7.3.1], page 287). In an instance, the copy of such an implicit declaration declares a view of the corresponding primitive subprogram of the ancestor or progenitor of the formal derived type, even if this primitive has been overridden for the actual type. When the ancestor or progenitor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor or progenitor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type.

22/1

For a prefix S that denotes a formal indefinite subtype, the following attribute is defined:

23

S'Definite

S'Definite yields True if the actual subtype corresponding to S is definite; otherwise it yields False. The value of this attribute is of the predefined type Boolean.

Dynamic Semantics

23.1/2

In the case where a formal type is tagged with unknown discriminants, and the actual type is a class-wide type <T>'Class:

23.2/2

- For the purposes of defining the primitive operations of the formal type, each of the primitive operations of the actual type is considered to be a subprogram (with an intrinsic calling convention -- see Section 7.3.1 [6.3.1], page 263) whose body consists of a dispatching call upon the corresponding operation of <T>, with its formal parameters as the actual parameters. If it is a function, the result of the dispatching call is returned.

23.3/2

- If the corresponding operation of <T> has no controlling formal parameters, then the controlling tag value is determined by the context of the call, according to the rules for tag-indeterminate calls (see Section 4.9.2 [3.9.2], page 145, and Section 6.2 [5.2], page 242). In the case where the tag would be statically determined to be that of the formal type, the call raises Program_Error. If such a function is renamed, any call on the renaming raises Program_Error.

NOTES

24/2

9 In accordance with the general rule that the actual type shall belong to the category determined for the formal (see Section 13.5 [12.5], page 460, "Section 13.5 [12.5], page 460, Formal Types"):

25

- If the formal type is nonlimited, then so shall be the actual;

26

- For a formal derived type, the actual shall be in the class rooted at the ancestor subtype.

27

10 The actual type can be abstract only if the formal type is abstract (see Section 4.9.3 [3.9.3], page 149).

28

11 If the formal has a `discriminant_part`, the actual can be either definite or indefinite. Otherwise, the actual has to be definite.

13.5.2 12.5.2 Formal Scalar Types

1/2

A `<formal scalar type>` is one defined by any of the `formal_type_definitions` in this subclause. The category determined for a formal scalar type is the category of all discrete, signed integer, modular, floating point, ordinary fixed point, or decimal types.

Syntax

2

`formal_discrete_type_definition ::= (<>)`

3

`formal_signed_integer_type_definition ::= range <>`

4

`formal_modular_type_definition ::= mod <>`

5

`formal_floating_point_definition ::= digits <>`

6

`formal_ordinary_fixed_point_definition ::= delta <>`

7

formal_decimal_fixed_point_definition ::= delta <> digits <>
Legality Rules

8

The actual type for a formal scalar type shall not be a nonstandard numeric type.

NOTES

9

12 The actual type shall be in the class of types implied by the syntactic category of the formal type definition (see Section 13.5 [12.5], page 460, "Section 13.5 [12.5], page 460, Formal Types"). For example, the actual for a formal_modular_type_definition shall be a modular type.

13.5.3 12.5.3 Formal Array Types

1/2

The category determined for a formal array type is the category of all array types.

Syntax

2

formal_array_type_definition ::= array_type_definition
Legality Rules

3

The only form of discrete_subtype_definition that is allowed within the declaration of a generic formal (constrained) array subtype is a subtype_mark.

4

For a formal array subtype, the actual subtype shall satisfy the following conditions:

5

- The formal array type and the actual array type shall have the same dimensionality; the formal subtype and the actual subtype shall be either both constrained or both unconstrained.

6

- For each index position, the index types shall be the same, and the index subtypes (if unconstrained), or the index ranges (if constrained), shall statically match (see Section 5.9.1 [4.9.1], page 238).

7

- The component subtypes of the formal and actual array types shall statically match.

8

- If the formal type has aliased components, then so shall the actual.

Examples

9

<Example of formal array types:>

10

```
--< given the generic package >
```

11

```
generic
  type Item   is private;
  type Index  is (<>);
  type Vector is array (Index range <>) of Item;
  type Table  is array (Index) of Item;
package P is
  ...
end P;
```

12

```
--< and the types >
```

13

```
type Mix   is array (Color range <>) of Boolean;
type Option is array (Color) of Boolean;
```

14

```
--< then Mix can match Vector and Option can match Table >
```

15

```
package R is new P(Item   => Boolean, Index => Color,
                  Vector => Mix,      Table => Option);
```

16

```
--< Note that Mix cannot match Table and Option cannot match Vector>■
```

13.5.4 12.5.4 Formal Access Types

1/2

The category determined for a formal access type is the category of all access types.

Syntax

2

formal-access_type_definition ::= access_type_definition

Legality Rules

3

For a formal access-to-object type, the designated subtypes of the formal and actual types shall statically match.

4/2

If and only if the `general_access_modifier` constant applies to the formal, the actual shall be an access-to-constant type. If the `general_access_modifier` all applies to the formal, then the actual shall be a general access-to-variable type (see Section 4.10 [3.10], page 156). If and only if the formal subtype excludes null, the actual subtype shall exclude null.

5

For a formal access-to-subprogram subtype, the designated profiles of the formal and the actual shall be mode-conformant, and the calling convention of the actual shall be `<protected>` if and only if that of the formal is `<protected>`.

Examples

6

`<Example of formal access types:>`

7

```
--< the formal types of the generic package >
```

8

```
generic
  type Node is private;
  type Link is access Node;
package P is
  ...
end P;
```

9

```
--< can be matched by the actual types >
```

10

```
type Car;
type Car_Name is access Car;
```

11

```
type Car is
  record
    Pred, Succ : Car_Name;
    Number      : License_Number;
    Owner       : Person;
  end record;
```

12

```
--< in the following generic instantiation >
```

13

```
package R is new P(Node => Car, Link => Car_Name);
```

13.5.5 12.5.5 Formal Interface Types

1/2

The category determined for a formal interface type is the category of all interface types.

Syntax

2/2

```
formal_interface_type_definition ::= interface_type_definition
```

Legality Rules

3/2

The actual type shall be a descendant of every progenitor of the formal type.

4/2

The actual type shall be a limited, task, protected, or synchronized interface if and only if the formal type is also, respectively, a limited, task, protected, or synchronized interface.

Examples

5/2

```
type Root_Work_Item is tagged private;
```

6/2

```
generic
  type Managed_Task is task interface;
  type Work_Item(<>) is new Root_Work_Item with private;
package Server_Manager is
  task type Server is new Managed_Task with
    entry Start(Data : in out Work_Item);
  end Server;
end Server_Manager;
```

7/2

This generic allows an application to establish a standard interface that all tasks need to implement so they can be managed appropriately by an application-specific scheduler.

13.6 12.6 Formal Subprograms

1

Formal subprograms can be used to pass callable entities to a generic unit.

Syntax

2/2

```
formal_subprogram_declaration ::= formal_concrete_subprogram_declaration
```

```
| formal_abstract_subprogram_declaration
```



2.1/2

formal_concrete_subprogram_declaration ::=
with subprogram_specification [is subprogram_default];

2.2/2

formal_abstract_subprogram_declaration ::=
with subprogram_specification is abstract [subprogram_default];

3/2

subprogram_default ::= default_name | <> | null

4

default_name ::= name

4.1/2

A subprogram_default of null shall not be specified for a formal function or for a formal_abstract_subprogram_declaration.

Name Resolution Rules

5

The expected profile for the default_name, if any, is that of the formal subprogram.

6

For a generic formal subprogram, the expected profile for the actual is that of the formal subprogram.

Legality Rules

7

The profiles of the formal and any named default shall be mode-conformant.

8

The profiles of the formal and actual shall be mode-conformant.

8.1/2

For a parameter or result subtype of a formal_subprogram_declaration that has an explicit null_exclusion:

8.2/2

- if the actual matching the formal_subprogram_declaration denotes a generic formal object of another generic unit <G>, and the instantiation containing the actual that occurs within the body of a generic unit <G> or within the body of a generic unit declared within the declarative region of the generic unit <G>, then the corresponding parameter or result type of the formal subprogram of <G> shall have a null_exclusion;

8.3/2

- otherwise, the subtype of the corresponding parameter or result type of the actual matching the formal_subprogram_declaration shall exclude null. In addition to the

places where Legality Rules normally apply (see Section 13.3 [12.3], page 454), this rule applies also in the private part of an instance of a generic unit.

8.4/2

If a formal parameter of a `formal_abstract_subprogram_declaration` (see [S0277], page 471) is of a specific tagged type `<T>` or of an anonymous access type designating a specific tagged type `<T>`, `<T>` is called a `<controlling type>` of the `formal_abstract_subprogram_declaration` (see [S0277], page 471). Similarly, if the result of a `formal_abstract_subprogram_declaration` (see [S0277], page 471) for a function is of a specific tagged type `<T>` or of an anonymous access type designating a specific tagged type `<T>`, `<T>` is called a `controlling type` of the `formal_abstract_subprogram_declaration` (see [S0277], page 471). A `formal_abstract_subprogram_declaration` (see [S0277], page 471) shall have exactly one `controlling type`.

8.5/2

The actual subprogram for a `formal_abstract_subprogram_declaration` (see [S0277], page 471) shall be a dispatching operation of the `controlling type` or of the actual type corresponding to the `controlling type`.

Static Semantics

9

A `formal_subprogram_declaration` declares a generic formal subprogram. The types of the formal parameters and result, if any, of the formal subprogram are those determined by the `subtype_marks` given in the `formal_subprogram_declaration`; however, independent of the particular subtypes that are denoted by the `subtype_marks`, the nominal subtypes of the formal parameters and result, if any, are defined to be `nonstatic`, and `unconstrained` if of an array type (no applicable index constraint is provided in a call on a formal subprogram). In an instance, a `formal_subprogram_declaration` declares a view of the actual. The profile of this view takes its subtypes and calling convention from the original profile of the actual entity, while taking the formal parameter names and `default_expression` (see [S0063], page 123)s from the profile given in the `formal_subprogram_declaration` (see [S0275], page 470). The view is a function or procedure, never an entry.

10

If a generic unit has a `subprogram_default` specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal.

10.1/2

If a generic unit has a `subprogram_default` specified by the reserved word `null`, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a null procedure having the profile given in the `formal_subprogram_declaration` (see [S0275], page 470).

10.2/2

The subprogram declared by a `formal_abstract_subprogram_declaration` (see [S0277], page 471) with a `controlling type` `<T>` is a dispatching operation of type `<T>`.

NOTES

11

13 The matching rules for formal subprograms state requirements that are similar to those applying to subprogram_renaming_declarations (see Section 9.5.4 [8.5.4], page 319). In particular, the name of a parameter of the formal subprogram need not be the same as that of the corresponding parameter of the actual subprogram; similarly, for these parameters, default_expressions need not correspond.

12

14 The constraints that apply to a parameter of a formal subprogram are those of the corresponding formal parameter of the matching actual subprogram (not those implied by the corresponding subtype_mark in the _specification of the formal subprogram). A similar remark applies to the result of a function. Therefore, to avoid confusion, it is recommended that the name of a first subtype be used in any declaration of a formal subprogram.

13

15 The subtype specified for a formal parameter of a generic formal subprogram can be any visible subtype, including a generic formal subtype of the same generic_formal_part.

14

16 A formal subprogram is matched by an attribute of a type if the attribute is a function with a matching specification. An enumeration literal of a given type matches a parameterless formal function whose result type is the given type.

15

17 A default_name denotes an entity that is visible or directly visible at the place of the generic_declaration; a box used as a default is equivalent to a name that denotes an entity that is directly visible at the place of the _instantiation.

16/2

18 The actual subprogram cannot be abstract unless the formal subprogram is a formal_abstract_subprogram_declaration (see [S0277], page 471) (see Section 4.9.3 [3.9.3], page 149).

16.1/2

19 The subprogram declared by a formal_abstract_subprogram_declaration (see [S0277], page 471) is an abstract subprogram. All calls on a subprogram declared by a formal_abstract_subprogram_declaration (see [S0277], page 471) must be dispatching calls. See Section 4.9.3 [3.9.3], page 149.

16.2/2

20 A null procedure as a subprogram default has convention Intrinsic (see Section 7.3.1 [6.3.1], page 263).

Examples

17

<Examples of generic formal subprograms:>

18/2

```
with function "+"(X, Y : Item) return Item is <>;
with function Image(X : Enum) return String is Enum'Image;
with procedure Update is Default_Update;
with procedure Pre_Action(X : in Item) is null; --< defaults to no action>
with procedure Write(S      : not null access Root_Stream_Type'Class;
                    Desc : Descriptor)
                    is abstract Descriptor'Write; --< see Section 14.13.2
[13.13.2], page 540>
--< Dispatching operation on Descriptor with default>
```

19

```
--< given the generic procedure declaration >
```

20

```
generic
  with procedure Action (X : in Item);
  procedure Iterate(Seq : in Item_Sequence);
```

21

```
--< and the procedure >
```

22

```
procedure Put_Item(X : in Item);
```

23

```
--< the following instantiation is possible >
```

24

```
procedure Put_List is new Iterate(Action => Put_Item);
```

13.7 12.7 Formal Packages

1

Formal packages can be used to pass packages to a generic unit. The formal-package-declaration declares that the formal package is an instance of a given generic

package. Upon instantiation, the actual package has to be an instance of that generic package.

Syntax

2

```
formal_package_declaration ::=  
    with package defining_identifier is new <generic_package->name formal_package_actual_part;
```

3/2

```
formal_package_actual_part ::=  
    ([others =>] <>)  
    | [generic_actual_part]  
    | (formal_package_association {, formal_package_association} [, others => <>])
```

3.1/2

```
formal_package_association ::=  
    generic_association  
    | <generic_formal_parameter->selector_name => <>
```

3.2/2

Any positional formal_package_associations shall precede any named formal_package_associations.

Legality Rules

4

The <generic_package->name shall denote a generic package (the <template> for the formal package); the formal package is an instance of the template.

4.1/2

A formal_package_actual_part shall contain at most one formal_package_association for each formal parameter. If the formal_package_actual_part does not include "others => <>", each formal parameter without an association shall have a default_expression or subprogram_default.

5/2

The actual shall be an instance of the template. If the formal_package_actual_part is (<>) or (others => <>), then the actual may be any instance of the template; otherwise, certain of the actual parameters of the actual instance shall match the corresponding actual parameters of the formal package, determined as follows:

5.1/2

- If the formal_package_actual_part (see [S0281], page 475) includes generic_associations as well as associations with <>, then only the actual parameters specified explicitly with generic_associations are required to match;

5.2/2

- Otherwise, all actual parameters shall match, whether any actual parameter is given explicitly or by default.

5.3/2

The rules for matching of actual parameters between the actual instance and the formal package are as follows:

6/2

- For a formal object of mode in, the actuals match if they are static expressions with the same value, or if they statically denote the same constant, or if they are both the literal null.

7

- For a formal subtype, the actuals match if they denote statically matching subtypes.

8

- For other kinds of formals, the actuals match if they statically denote the same entity.

8.1/1

For the purposes of matching, any actual parameter that is the name of a formal object of mode in is replaced by the formal object's actual expression (recursively).

Static Semantics

9

A formal_package_declaration declares a generic formal package.

10/2

The visible part of a formal package includes the first list of basic_declarative_items of the package_specification (see [S0174], page 279). In addition, for each actual parameter that is not required to match, a copy of the declaration of the corresponding formal parameter of the template is included in the visible part of the formal package. If the copied declaration is for a formal type, copies of the implicit declarations of the primitive subprograms of the formal type are also included in the visible part of the formal package.

11/2

For the purposes of matching, if the actual instance <A> is itself a formal package, then the actual parameters of <A> are those specified explicitly or implicitly in the formal_package_actual_part for <A>, plus, for those not specified, the copies of the formal parameters of the template included in the visible part of <A>.

Examples

12/2

<Example of a generic package with formal package parameters:>

13/2

```
with Ada.Containers.Ordered_Maps;  --< see Section 15.18.6 [A.18.6],
page 846>
generic
  with package Mapping_1 is new Ada.Containers.Ordered_Maps(<>);
```


24/2

```
Apple_Info : constant Symbol_Info := String_Info.Lookup("Apple");■
```

25/2

```
end Symbol_Package;
```

13.8 12.8 Example of a Generic Package

1

The following example provides a possible formulation of stacks by means of a generic package. The size of each stack and the type of the stack elements are provided as generic formal parameters.

Examples

2/1

<This paragraph was deleted.>

3

```
generic
  Size : Positive;
  type Item is private;
package Stack is
  procedure Push(E : in Item);
  procedure Pop (E : out Item);
  Overflow, Underflow : exception;
end Stack;
```

4

```
package body Stack is
```

5

```
  type Table is array (Positive range <>) of Item;
  Space : Table(1 .. Size);
  Index : Natural := 0;
```

6

```
  procedure Push(E : in Item) is
  begin
    if Index >= Size then
      raise Overflow;
    end if;
    Index := Index + 1;
    Space(Index) := E;
  end Push;
```

7

```
procedure Pop(E : out Item) is
begin
  if Index = 0 then
    raise Underflow;
  end if;
  E := Space(Index);
  Index := Index - 1;
end Pop;
```

8

```
end Stack;
```

9

Instances of this generic package can be obtained as follows:

10

```
package Stack_Int is new Stack(Size => 200, Item => Integer);
package Stack_Bool is new Stack(100, Boolean);
```

11

Thereafter, the procedures of the instantiated packages can be called as follows:

12

```
Stack_Int.Push(N);
Stack_Bool.Push(True);
```

13

Alternatively, a generic formulation of the type Stack can be given as follows (package body omitted):

14

```
generic
  type Item is private;
package On_Stacks is
  type Stack(Size : Positive) is limited private;
  procedure Push(S : in out Stack; E : in Item);
  procedure Pop (S : in out Stack; E : out Item);
  Overflow, Underflow : exception;
private
  type Table is array (Positive range <>) of Item;
  type Stack(Size : Positive) is
    record
      Space : Table(1 .. Size);
      Index : Natural := 0;
    end record;
```

```
end On_Stacks;
```

15

In order to use such a package, an instance has to be created and thereafter stacks of the corresponding type can be declared:

16

```
declare
  package Stack_Real is new On_Stacks(Real); use Stack_Real;
  S : Stack(100);
begin
  ...
  Push(S, 2.54);
  ...
end;
```


14 13 Representation Issues

1/1

This section describes features for querying and controlling certain aspects of entities and for interfacing to hardware.

14.1 13.1 Operational and Representation Items

0.1/1

Representation and operational items can be used to specify aspects of entities. Two kinds of aspects of entities can be specified: aspects of representation and operational aspects. Representation items specify how the types and other entities of the language are to be mapped onto the underlying machine. Operational items specify other properties of entities.

1/1

There are six kinds of <representation items>: `attribute_definition_clause` (see [S0286], page 487)s for representation attributes, `enumeration_representation_clause` (see [S0287], page 500)s, `record_representation_clause` (see [S0289], page 502)s, `at_clauses`, `component_clauses`, and <representation pragmas>. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware).

1.1/1

An <operational item> is an `attribute_definition_clause` for an operational attribute.

1.2/1

An operational item or a representation item applies to an entity identified by a `local_name`, which denotes an entity declared local to the current declarative region, or a library unit declared immediately preceding a representation pragma in a compilation.

Syntax

2/1

```
aspect_clause ::= attribute_definition_clause
                | enumeration_representation_clause
                | record_representation_clause
                | at_clause
```

3

```
local_name ::= direct_name
             | direct_name'attribute_designator
             | <library_unit_>name
```

4/1

A representation pragma is allowed only at places where an `aspect_clause` or `compilation_unit` is allowed.

Name Resolution Rules

5/1

In an operational item or representation item, if the `local_name` is a `direct_name`, then it

shall resolve to denote a declaration (or, in the case of a pragma, one or more declarations) that occurs immediately within the same declarative region as the item. If the `local_name` has an `attribute_designator`, then it shall resolve to denote an implementation-defined component (see Section 14.5.1 [13.5.1], page 501) or a class-wide type implicitly declared immediately within the same declarative region as the item. A `local_name` that is a `<library_unit>name` (only permitted in a representation pragma) shall resolve to denote the `library_item` that immediately precedes (except for other pragmas) the representation pragma.

Legality Rules

6/1

The `local_name` of an `aspect_clause` or representation pragma shall statically denote an entity (or, in the case of a pragma, one or more entities) declared immediately preceding it in a compilation, or within the same `declarative_part` (see [S0086], page 175), `package_specification` (see [S0174], page 279), `task_definition` (see [S0190], page 329), `protected_definition` (see [S0195], page 338), or `record_definition` (see [S0067], page 130) as the representation or operational item. If a `local_name` denotes a local callable entity, it may do so through a `local_subprogram_renaming_declaration` (see [S0186], page 320) (as a way to resolve ambiguity in the presence of overloading); otherwise, the `local_name` shall not denote a `renaming_declaration` (see [S0182], page 316).

7/2

The `<representation>` of an object consists of a certain number of bits (the `<size>` of the object). For an object of an elementary type, these are the bits that are normally read or updated by the machine code when loading, storing, or operating-on the value of the object. For an object of a composite type, these are the bits reserved for this object, and include bits occupied by subcomponents of the object. If the size of an object is greater than that of its subtype, the additional bits are padding bits. For an elementary object, these padding bits are normally read and updated along with the others. For a composite object, padding bits might not be read or updated in any given composite operation, depending on the implementation.

8

A representation item `<directly specifies>` an `<aspect of representation>` of the entity denoted by the `local_name`, except in the case of a type-related representation item, whose `local_name` shall denote a first subtype, and which directly specifies an aspect of the subtype's type. A representation item that names a subtype is either `<subtype-specific>` (Size and Alignment clauses) or `<type-related>` (all others). Subtype-specific aspects may differ for different subtypes of the same type.

8.1/1

An operational item `<directly specifies>` an `<operational aspect>` of the type of the subtype denoted by the `local_name`. The `local_name` of an operational item shall denote a first subtype. An operational item that names a subtype is `type-related`.

9

A representation item that directly specifies an aspect of a subtype or type shall appear after the type is completely defined (see Section 4.11.1 [3.11.1], page 177), and before the subtype or type is frozen (see Section 14.14 [13.14], page 550). If a representation item is given that directly specifies an aspect of an entity, then it is illegal to give another representation item that directly specifies the same aspect of the entity.

9.1/1

An operational item that directly specifies an aspect of a type shall appear before the type is frozen (see Section 14.14 [13.14], page 550). If an operational item is given that directly specifies an aspect of a type, then it is illegal to give another operational item that directly specifies the same aspect of the type.

10

For an untagged derived type, no type-related representation items are allowed if the parent type is a by-reference type, or has any user-defined primitive subprograms.

11/2

Operational and representation aspects of a generic formal parameter are the same as those of the actual. Operational and representation aspects are the same for all views of a type. A type-related representation item is not allowed for a descendant of a generic formal untagged type.

12

A representation item that specifies the Size for a given subtype, or the size or storage place for an object (including a component) of a given subtype, shall allow for enough storage space to accommodate any value of the subtype.

13/1

A representation or operational item that is not supported by the implementation is illegal, or raises an exception at run time.

13.1/2

A `type_declaration` is illegal if it has one or more progenitors, and a representation item applies to an ancestor, and this representation item conflicts with the representation of some other ancestor. The cases that cause conflicts are implementation defined.

Static Semantics

14

If two subtypes statically match, then their subtype-specific aspects (Size and Alignment) are the same.

15/1

A derived type inherits each type-related aspect of representation of its parent type that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent type from the grandparent type. A derived subtype inherits each subtype-specific aspect of representation of its parent subtype that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent subtype from the grandparent subtype, but only if the parent subtype statically matches the first subtype of the parent type. An inherited aspect of representation is overridden by a subsequent representation item that specifies the same aspect of the type or subtype.

15.1/2

In contrast, whether operational aspects are inherited by an untagged derived type depends on each specific aspect. Operational aspects are never inherited for a tagged type. When operational aspects are inherited by an untagged derived type, aspects that were directly specified by operational items that are visible at the point of the derived type declaration, or (in the case where the parent is derived) that were inherited by the parent type from

the grandparent type are inherited. An inherited operational aspect is overridden by a subsequent operational item that specifies the same aspect of the type.

15.2/2

When an aspect that is a subprogram is inherited, the derived type inherits the aspect in the same way that a derived type inherits a user-defined primitive subprogram from its parent (see Section 4.4 [3.4], page 66).

16

Each aspect of representation of an entity is as follows:

17

- If the aspect is <specified> for the entity, meaning that it is either directly specified or inherited, then that aspect of the entity is as specified, except in the case of `Storage-Size`, which specifies a minimum.

18

- If an aspect of representation of an entity is not specified, it is chosen by default in an unspecified manner.

18.1/1

If an operational aspect is <specified> for an entity (meaning that it is either directly specified or inherited), then that aspect of the entity is as specified. Otherwise, the aspect of the entity has the default value for that aspect.

18.2/2

A representation item that specifies an aspect of representation that would have been chosen in the absence of the representation item is said to be <confirming>.

Dynamic Semantics

19/1

For the elaboration of an `aspect_clause`, any evaluable constructs within it are evaluated.

Implementation Permissions

20

An implementation may interpret aspects of representation in an implementation-defined manner. An implementation may place implementation-defined restrictions on representation items. A <recommended level of support> is specified for representation items and related features in each subclause. These recommendations are changed to requirements for implementations that support the Systems Programming Annex (see Section 17.2 [C.2], page 950, "Section 17.2 [C.2], page 950, Required Representation Support").

Implementation Advice

21

The recommended level of support for all representation items is qualified as follows:

21.1/2

- A confirming representation item should be supported.

22

- An implementation need not support representation items containing nonstatic expressions, except that an implementation should support a representation item for a given entity if each nonstatic expression in the representation item is a name that statically denotes a constant declared before the entity.

23

- An implementation need not support a specification for the Size for a given composite subtype, nor the size or storage place for an object (including a component) of a given composite subtype, unless the constraints on the subtype and its composite subcomponents (if any) are all static constraints.

24/2

- An implementation need not support a nonconfirming representation item if it could cause an aliased object or an object of a by-reference type to be allocated at a non-addressable location or, when the alignment attribute of the subtype of such an object is nonzero, at an address that is not an integral multiple of that alignment.

25/2

- An implementation need not support a nonconfirming representation item if it could cause an aliased object of an elementary type to have a size other than that which would have been chosen by default.

26/2

- An implementation need not support a nonconfirming representation item if it could cause an aliased object of a composite type, or an object whose type is by-reference, to have a size smaller than that which would have been chosen by default.

27/2

- An implementation need not support a nonconfirming subtype-specific representation item specifying an aspect of representation of an indefinite or abstract subtype.

28/2

For purposes of these rules, the determination of whether a representation item applied to a type <could cause> an object to have some property is based solely on the properties of the type itself, not on any available information about how the type is used. In particular, it presumes that minimally aligned objects of this type might be declared at some point.

14.2 13.2 Pragma Pack

1

A pragma Pack specifies that storage minimization should be the main criterion when selecting the representation of a composite type.

Syntax

2

The form of a pragma Pack is as follows:

3

```
pragma Pack(<first_subtype_>local_name);  
                Legality Rules
```

4

The <first_subtype_>local_name of a pragma Pack shall denote a composite subtype.

Static Semantics

5

A pragma Pack specifies the <packing> aspect of representation; the type (or the extension part) is said to be <packed>. For a type extension, the parent part is packed as for the parent type, and a pragma Pack causes packing only of the extension part.

Implementation Advice

6

If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations.

6.1/2

If a packed type has a component that is not of a by-reference type and has no aliased part, then such a component need not be aligned according to the Alignment of its subtype; in particular it need not be allocated on a storage element boundary.

7

The recommended level of support for pragma Pack is:

8

- For a packed record type, the components should be packed as tightly as possible subject to the Sizes of the component subtypes, and subject to any record_representation_clause that applies to the type; the implementation may, but need not, reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words.

9

- For a packed array type, if the component subtype's Size is less than or equal to the word size, and Component_Size is not specified for the type, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size.

14.3 13.3 Operational and Representation Attributes

1/1

The values of certain implementation-dependent characteristics can be obtained by interrogating appropriate operational or representation attributes. Some of these attributes are specifiable via an attribute_definition_clause.

Syntax

2

```
attribute_definition_clause ::=  
    for local_name'attribute_designator use expression;  
    | for local_name'attribute_designator use name;
```

Name Resolution Rules

3

For an `attribute_definition_clause` that specifies an attribute that denotes a value, the form with an expression shall be used. Otherwise, the form with a name shall be used.

4

For an `attribute_definition_clause` that specifies an attribute that denotes a value or an object, the expected type for the expression or name is that of the attribute. For an `attribute_definition_clause` that specifies an attribute that denotes a subprogram, the expected profile for the name is the profile required for the attribute. For an `attribute_definition_clause` that specifies an attribute that denotes some other kind of entity, the name shall resolve to denote an entity of the appropriate kind.

Legality Rules

5/1

An `attribute_designator` is allowed in an `attribute_definition_clause` only if this International Standard explicitly allows it, or for an implementation-defined attribute if the implementation allows it. Each specifiable attribute constitutes an operational aspect or aspect of representation.

6

For an `attribute_definition_clause` that specifies an attribute that denotes a subprogram, the profile shall be mode conformant with the one required for the attribute, and the convention shall be Ada. Additional requirements are defined for particular attributes.

Static Semantics

7/2

A `<Size clause>` is an `attribute_definition_clause` whose `attribute_designator` is `Size`. Similar definitions apply to the other specifiable attributes.

8

A `<storage element>` is an addressable element of storage in the machine. A `<word>` is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of an integral number of storage elements.

8.1/2

A `<machine scalar>` is an amount of storage that can be conveniently and efficiently loaded, stored, or operated upon by the hardware. Machine scalars consist of an integral number of storage elements. The set of machine scalars is implementation defined, but must include at least the storage element and the word. Machine scalars are used to interpret component_clauses when the nondefault bit ordering applies.

9/1

The following representation attributes are defined: `Address`, `Alignment`, `Size`, `Storage_Size`, and `Component_Size`.

10/1

For a prefix X that denotes an object, program unit, or label:

11

X'Address

Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address.

12

Address may be specified for stand-alone objects and for program units via an attribute_definition_clause.

Erroneous Execution ■

13

If an Address is specified, it is the programmer's responsibility to ensure that the address is valid; otherwise, program execution is erroneous.

Implementation Advice

14

For an array X, X'Address should point at the first component of the array, and not at the array bounds.

15

The recommended level of support for the Address attribute is:

16

- X'Address should produce a useful result if X is an object that is aliased or of a by-reference type, or is an entity whose Address has been specified.

17

- An implementation should support Address clauses for imported subprograms.

18/2

- <This paragraph was deleted.>

19

- If the Address of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases.

NOTES

20

1 The specification of a link name in a pragma Export (see Section 16.1 [B.1], page 894) for a subprogram or object is an alternative to explicit specification of its link-time address, allowing a link-time directive to place the subprogram or object within memory.

21

2 The rules for the Size attribute imply, for an aliased object X, that if $X'Size = Storage_Unit$, then $X'Address$ points at a storage element containing all of the bits of X, and only the bits of X.

Static Semantics

22/2

For a prefix X that denotes an object:

23/2

X'Alignment

The value of this attribute is of type <universal_integer>, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If X'Alignment is not zero, then X is aligned on a storage unit boundary and X'Address is an integral multiple of X'Alignment (that is, the Address modulo the Alignment is zero).

24/2

<This paragraph was deleted.>

25/2

Alignment may be specified for stand-alone objects via an attribute-definition_clause (see [S0286], page 487); the expression of such a clause shall be static, and its value nonnegative.

26/2

<This paragraph was deleted.>

26.1/2

For every subtype S:

26.2/2

S'Alignment

The value of this attribute is of type <universal_integer>, and nonnegative.

26.3/2

For an object X of subtype S, if S'Alignment is not zero, then X'Alignment is a nonzero integral multiple of S'Alignment unless specified otherwise by a representation item.

26.4/2

Alignment may be specified for first subtypes via an attribute-definition_clause (see [S0286],

page 487); the expression of such a clause shall be static, and its value nonnegative.

Erroneous Execution

27

Program execution is erroneous if an Address clause is given that conflicts with the Alignment.

28/2

For an object that is not allocated under control of the implementation, execution is erroneous if the object is not aligned according to its Alignment.

Implementation Advice

29

The recommended level of support for the Alignment attribute for subtypes is:

30/2

- An implementation should support an Alignment clause for a discrete type, fixed point type, record type, or array type, specifying an Alignment value that is zero or a power of two, subject to the following:

31/2

- An implementation need not support an Alignment clause for a signed integer type specifying an Alignment greater than the largest Alignment value that is ever chosen by default by the implementation for any signed integer type. A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types.

32/2

- An implementation need not support a nonconfirming Alignment clause which could enable the creation of an object of an elementary type which cannot be easily loaded and stored by available machine instructions.

32.1/2

- An implementation need not support an Alignment specified for a derived tagged type which is not a multiple of the Alignment of the parent type. An implementation need not support a nonconfirming Alignment specified for a derived untagged by-reference type.

33

The recommended level of support for the Alignment attribute for objects is:

34/2

- <This paragraph was deleted.>

35

- For stand-alone library-level objects of statically constrained subtypes, the implementation should support all Alignments supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.

35.1/2

- For other objects, an implementation should at least support the alignments supported for their subtype, subject to the following:

35.2/2

- An implementation need not support Alignments specified for objects of a by-reference type or for objects of types containing aliased subcomponents if the specified Alignment is not a multiple of the Alignment of the subtype of the object.

NOTES

36

3 Alignment is a subtype-specific attribute.

37/2

<This paragraph was deleted.>

38

4 A `component_clause`, `Component_Size` clause, or a `pragma Pack` can override a specified Alignment.

Static Semantics

39/1

For a prefix `X` that denotes an object:

40

`X'Size`

Denotes the size in bits of the representation of the object. The value of this attribute is of the type `<universal_integer>`.

41

Size may be specified for stand-alone objects via an `attribute_definition_clause`; ■

the expression of such a clause shall be static and its value nonnegative.

Implementation Advice

41.1/2

The size of an array object should not include its bounds.

42/2

The recommended level of support for the Size attribute of objects is the same as for subtypes (see below), except that only a confirming Size clause need be supported for an aliased elementary object.

43/2

- <This paragraph was deleted.>

Static Semantics

44

For every subtype S:

45

S'Size

If S is definite, denotes the size (in bits) that the implementation would choose for the following objects of subtype S:

46

- A record component of subtype S when the record type is packed.

47

- The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype. ■

If `S` is indefinite, the meaning is implementation defined. The value of this attribute is of the type `<universal_integer>`. The `Size` of an object is at least as large as that of its subtype, unless the object's `Size` is determined by a `Size` clause, a `component_clause`, or a `Component_Size` clause. `Size` may be specified for first subtypes via an `attribute_definition_clause` (see [S0286], page 487); the expression of such a clause shall be static and its value nonnegative.

Implementation Requirements

49

In an implementation, `Boolean'Size` shall be 1.

Implementation Advice

50/2

If the `Size` of a subtype allows for efficient independent addressability (see Section 10.10 [9.10], page 389) on the target architecture, then the `Size` of the following objects of the subtype should equal the `Size` of the subtype:

51

- Aliased objects (including components).

52

- Unaliased components, unless the `Size` of the component is determined by a `component_clause` or `Component_Size` clause.

53

A `Size` clause on a composite subtype should not affect the internal layout of components.

54

The recommended level of support for the Size attribute of subtypes is:

55

- The Size (if not specified) of a static discrete or fixed point subtype should be the number of bits needed to represent each value belonging to the subtype using an unbiased representation, leaving space for a sign bit only if the subtype contains negative values. If such a subtype is a first subtype, then an implementation should support a specified Size for it that reflects this representation.

56

- For a subtype implemented with levels of indirection, the Size should include the size of the pointers, but not the size of what they point at.

56.1/2

- An implementation should support a Size clause for a discrete type, fixed point type, record type, or array type, subject to the following:

56.2/2

- An implementation need not support a Size clause for a signed integer type specifying a Size greater than that of the largest signed integer type supported by the implementation in the absence of a size clause (that is, when the size is chosen by default). A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types.

56.3/2

- A nonconfirming size clause for the first subtype of a derived untagged by-reference type need not be supported.

NOTES

57

- 5 Size is a subtype-specific attribute.

58

6 A `component_clause` or `Component_Size` clause can override a specified `Size`. A `pragma Pack` cannot.

Static Semantics

59/1

For a prefix `T` that denotes a task object (after any implicit dereference):

60

`T'Storage_Size`

Denotes the number of storage elements reserved for the task. The value of this attribute is of the type `<universal_integer>`.

The `Storage_Size` includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the "task control block" used by some implementations.) If a `pragma Storage_Size` is given, the value of the `Storage_Size` attribute is at least the value specified in the pragma.

61

A `pragma Storage_Size` specifies the amount of storage to be reserved for the execution of a task.

Syntax

62

The form of a `pragma Storage_Size` is as follows:

63

```
pragma Storage_Size(expression);
```

64

A pragma `Storage_Size` is allowed only immediately within a `task_definition`.

Name Resolution Rules

65

The expression of a pragma `Storage_Size` is expected to be of any integer type.

Dynamic Semantics

66

A pragma `Storage_Size` is elaborated when an object of the type defined by the immediately enclosing `task_definition` is created. For the elaboration of a pragma `Storage_Size`, the expression is evaluated; the `Storage_Size` attribute of the newly created task object is at least the value of the expression.

67

At the point of task object creation, or upon task activation, `Storage_Error` is raised if there is insufficient free storage to accommodate the requested `Storage_Size`.

Static Semantics

68/1

For a prefix `X` that denotes an array subtype or array object (after any implicit dereference):

69

`X'Component_Size`

Denotes the size in bits of components of the type of `X`. The value of this attribute is of type `<universal_integer>`.

70

`Component_Size` may be specified for array types via an `attribute_definition_clause` (see [S0286], page 487); the expression of such a clause shall be static, and its value nonnegative.

Implementation Advice

71

The recommended level of support for the `Component_Size` attribute is:

72

- An implementation need not support specified `Component_Sizes` that are less than the Size of the component subtype.

- An implementation should support specified `Component_Sizes` that are factors and multiples of the word size. For such `Component_Sizes`, the array should contain no gaps between components. For other `Component_Sizes` (if supported), the array should contain no gaps between components when packing is also specified; the implementation should forbid this combination in cases where it cannot support a no-gaps representation.

Static Semantics

73.1/1

The following operational attribute is defined: `External_Tag`.

74/1

For every subtype `S` of a tagged type `<T>` (specific or class-wide):

75/1

`S'External_Tag`

`S'External_Tag`
denotes an external string representation for `S'Tag`; it is of the predefined type `String`. `External_Tag` may be specified for a specific tagged type via an `attribute_definition_clause`; the expression of such a clause shall be static. The default external tag representation is implementation defined. See Section 4.9.2 [3.9.2], page 145, and Section 14.13.2 [13.13.2], page 540. The value of `External_Tag` is never inherited; the default value is always used unless a new value is directly specified for a type.

Implementation Requirements

76

In an implementation, the default external tag for each specific tagged type declared in a

partition shall be distinct, so long as the type is declared outside an instance of a generic body. If the compilation unit in which a given tagged type is declared, and all compilation units on which it semantically depends, are the same in two different partitions, then the external tag for the type shall be the same in the two partitions. What it means for a compilation unit to be the same in two different partitions is implementation defined. At a minimum, if the compilation unit is not recompiled between building the two different partitions that include it, the compilation unit is considered the same in the two partitions.

NOTES

77/2

7 The following language-defined attributes are specifiable, at least for some of the kinds of entities to which they apply: Address, Alignment, Bit_Order, Component_Size, External_Tag, Input, Machine_Radix, Output, Read, Size, Small, Storage_Pool, Storage_Size, Stream_Size, and Write.

78

8 It follows from the general rules in Section 14.1 [13.1], page 481, that if one writes "for X'Size use Y;" then the X'Size attribute_reference will return Y (assuming the implementation allows the Size clause). The same is true for all of the specifiable attributes except Storage_Size.

Examples

79

<Examples of attribute definition clauses:>

80

```
Byte : constant := 8;
Page : constant := 2**12;
```

81

```
type Medium is range 0 .. 65_000;
for Medium'Size use 2*Byte;
for Medium'Alignment use 2;
Device_Register : Medium;
for Device_Register'Size use Medium'Size;
for Device_Register'Address use System.Storage_Elements.To_Address(16#FFFF_0020#);
```

82

```
type Short is delta 0.01 range -100.0 .. 100.0;
for Short'Size use 15;
```

83

```
for Car_Name'Storage_Size use --< specify access type's storage pool size>■
```

```
2000*((Car'Size/System.Storage_Unit) +1); --< approximately 2000 cars>■
```

84/2

```
function My_Input(Stream : not null access Ada.Streams.Root_Stream_Type'Class)■  
    return T;  
for T'Input use My_Input; --< see Section 14.13.2 [13.13.2], page 540>■
```

NOTES

85

9 <Notes on the examples:> In the Size clause for Short, fifteen bits is the minimum necessary, since the type definition requires Short'Small $\leq 2^{*(-7)}$.

14.4 13.4 Enumeration Representation Clauses

1

An enumeration_representation_clause specifies the internal codes for enumeration literals.

Syntax

2

```
enumeration_representation_clause ::=  
    for <first_subtype_>local_name use enumeration_aggregate;
```

3

```
enumeration_aggregate ::= array_aggregate  
Name Resolution Rules
```

4

The enumeration_aggregate shall be written as a one-dimensional array_aggregate, for which the index subtype is the unconstrained subtype of the enumeration type, and each component expression is expected to be of any integer type.

Legality Rules

5

The <first_subtype_>local_name of an enumeration_representation_clause shall denote an enumeration subtype.

6/2

Each component of the array_aggregate shall be given by an expression rather than a <>. The expressions given in the array_aggregate shall be static, and shall specify distinct integer codes for each value of the enumeration type; the associated integer codes shall satisfy the predefined ordering relation of the type.

Static Semantics

7

An enumeration_representation_clause specifies the <coding> aspect of representation. The coding consists of the <internal code> for each enumeration literal, that is, the integral value used internally to represent each literal.

Implementation Requirements

8

For nonboolean enumeration types, if the coding is not specified for the type, then for each value of the type, the internal code shall be equal to its position number.

Implementation Advice

9

The recommended level of support for `enumeration_representation_clauses` is:

10

- An implementation should support at least the internal codes in the range `System.Min_Int..System.Max_Int`. An implementation need not support `enumeration_representation_clause` (see [S0287], page 500)s for boolean types.

NOTES

11/1

10 `Unchecked_Conversion` may be used to query the internal codes used for an enumeration type. The attributes of the type, such as `Succ`, `Pred`, and `Pos`, are unaffected by the `enumeration_representation_clause`. For example, `Pos` always returns the position number, <not> the internal integer code that might have been specified in an `enumeration_representation_clause`}.
Examples

Examples

12

<Example of an enumeration representation clause:>

13

```
type Mix_Code is (ADD, SUB, MUL, LDA, STA, STZ);
```

14

```
for Mix_Code use  
  (ADD => 1, SUB => 2, MUL => 3, LDA => 8, STA => 24, STZ =>33);
```

14.5 13.5 Record Layout

1

The <(record) layout> aspect of representation consists of the <storage places> for some or all components, that is, storage place attributes of the components. The layout can be specified with a `record_representation_clause` (see [S0289], page 502).

14.5.1 13.5.1 Record Representation Clauses

1

A `record_representation_clause` specifies the storage representation of records and record extensions, that is, the order, position, and size of components (including discriminants, if any).

Syntax

2

```
record_representation_clause ::=  
  for <first_subtype->local_name use  
  record [mod_clause]  
    {component_clause}  
  end record;
```

3

```
component_clause ::=  
  <component->local_name at position range first_bit .. last_bit;
```

4

```
position ::= <static->expression
```

5

```
first_bit ::= <static->simple_expression
```

6

```
last_bit ::= <static->simple_expression
```

Name Resolution Rules

7

Each position, first_bit, and last_bit is expected to be of any integer type.

Legality Rules

8/2

The <first_subtype->local_name of a record_representation_clause shall denote a specific record or record extension subtype.

9

If the <component->local_name is a direct_name, the local_name shall denote a component of the type. For a record extension, the component shall not be inherited, and shall not be a discriminant that corresponds to a discriminant of the parent type. If the <component->-local_name (see [S0285], page 481) has an attribute_designator (see [S0101], page 187), the direct_name (see [S0092], page 179) of the local_name (see [S0285], page 481) shall denote either the declaration of the type or a component of the type, and the attribute_designator (see [S0101], page 187) shall denote an implementation-defined implicit component of the type.

10

The position, first_bit, and last_bit shall be static expressions. The value of position and first_bit shall be nonnegative. The value of last_bit shall be no less than first_bit - 1.

10.1/2

If the nondefault bit ordering applies to the type, then either:

10.2/2

- the value of `last_bit` shall be less than the size of the largest machine scalar; or

10.3/2

- the value of `first_bit` shall be zero and the value of `last_bit + 1` shall be a multiple of `System.Storage_Unit`.

11

At most one `component_clause` is allowed for each component of the type, including for each discriminant (`component_clauses` may be given for some, all, or none of the components). Storage places within a `component_list` shall not overlap, unless they are for components in distinct variants of the same `variant_part`.

12

A name that denotes a component of a type is not allowed within a `record_representation_clause` for the type, except as the `<component_>local_name` of a `component_clause`.

Static Semantics

13/2

A `record_representation_clause` (without the `mod_clause`) specifies the layout.

13.1/2

If the default bit ordering applies to the type, the position, `first_bit`, and `last_bit` of each `component_clause` directly specify the position and size of the corresponding component.

13.2/2

If the nondefault bit ordering applies to the type then the layout is determined as follows:

13.3/2

- the `component_clauses` for which the value of `last_bit` is greater than or equal to the size of the largest machine scalar directly specify the position and size of the corresponding component;

13.4/2

- for other `component_clauses`, all of the components having the same value of position are considered to be part of a single machine scalar, located at that position; this machine scalar has a size which is the smallest machine scalar size larger than the largest `last_bit` for all `component_clauses` at that position; the `first_bit` and `last_bit` of each `component_clause` are then interpreted as bit offsets in this machine scalar.

14

A `record_representation_clause` for a record extension does not override the layout of the parent part; if the layout was specified for the parent type, it is inherited by the record extension.

Implementation Permissions

15

An implementation may generate implementation-defined components (for example,

one containing the offset of another component). An implementation may generate names that denote such implementation-defined components; such names shall be implementation-defined `attribute_references`. An implementation may allow such implementation-defined names to be used in `record_representation_clause` (see [S0289], page 502)s. An implementation can restrict such `component_clause` (see [S0290], page 502)s in any manner it sees fit.

16

If a `record_representation_clause` is given for an untagged derived type, the storage place attributes for all of the components of the derived type may differ from those of the corresponding components of the parent type, even for components whose storage place is not specified explicitly in the `record_representation_clause` (see [S0289], page 502).

Implementation Advice

17

The recommended level of support for `record_representation_clauses` is:

17.1/2

- An implementation should support machine scalars that correspond to all of the integer, floating point, and address formats supported by the machine.

18

- An implementation should support storage places that can be extracted with a load, mask, shift sequence of machine code, and set with a load, shift, mask, store sequence, given the available machine instructions and run-time model.

19

- A storage place should be supported if its size is equal to the Size of the component subtype, and it starts and ends on a boundary that obeys the Alignment of the component subtype.

20/2

- For a component with a subtype whose Size is less than the word size, any storage place that does not cross an aligned word boundary should be supported.

21

- An implementation may reserve a storage place for the tag field of a tagged type, and disallow other components from overlapping that place.

22

- An implementation need not support a `component_clause` for a component of an extension part if the storage place is not after the storage places of all components of the parent type, whether or not those storage places had been specified.

NOTES

23

11 If no `component_clause` is given for a component, then the choice of the storage place for the component is left to the implementation. If `component_clauses` are given for all components, the `record_representation_clause` completely specifies the representation of the type and will be obeyed exactly by the implementation.

Examples

24

<Example of specifying the layout of a record type:>

25

```
Word : constant := 4; --< storage element is byte, 4 bytes per word>■
```

26

```
type State      is (A,M,W,P);
type Mode       is (Fix, Dec, Exp, Signif);
```

27

```
type Byte_Mask  is array (0..7) of Boolean;
type State_Mask is array (State) of Boolean;
type Mode_Mask  is array (Mode) of Boolean;
```

28

```
type Program_Status_Word is
  record
    System_Mask      : Byte_Mask;
    Protection_Key   : Integer range 0 .. 3;
    Machine_State    : State_Mask;
    Interrupt_Cause  : Interruption_Code;
    Ilc               : Integer range 0 .. 3;
    Cc                : Integer range 0 .. 3;
    Program_Mask     : Mode_Mask;
    Inst_Address     : Address;
  end record;
```

29

```
for Program_Status_Word use
  record
    System_Mask      at 0*Word range 0 .. 7;
    Protection_Key   at 0*Word range 10 .. 11; --< bits 8,9 unused>■
    Machine_State    at 0*Word range 12 .. 15;
```

```

        Interrupt_Cause at 0*Word range 16 .. 31;
        Ilc              at 1*Word range 0  .. 1;  --< second word>
        Cc               at 1*Word range 2  .. 3;
        Program_Mask    at 1*Word range 4  .. 7;
        Inst_Address     at 1*Word range 8  .. 31;
    end record;

```

30

```

for Program_Status_Word'Size use 8*System.Storage_Unit;
for Program_Status_Word'Alignment use 8;

```

NOTES

31

12 <Note on the example:> The record_representation_clause defines the record layout. The Size clause guarantees that (at least) eight storage elements are used for objects of the type. The Alignment clause guarantees that aliased, imported, or exported objects of the type will have addresses divisible by eight.

14.5.2 13.5.2 Storage Place Attributes

Static Semantics

1

For a component C of a composite, non-array object R, the <storage place attributes> are defined:

2/2

R.C'Position

If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the position of the component_clause; otherwise, denotes the same value as R.C'Address - R'Address. The value of this attribute is of the type <universal_integer>.

3/2

R.C'First_Bit

If the nondefault bit ordering applies to the composite type, and if a `component_clause` specifies the placement of `C`, denotes the value given for the `first_bit` of the `component_clause`; otherwise, denotes the offset, from the start of the first of the storage elements occupied by `C`, of the first bit occupied by `C`. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type `<universal_integer>`.

4/2

R.C'Last_Bit

If the nondefault bit ordering applies to the composite type, and if a `component_clause` specifies the placement of `C`, denotes the value given for the `last_bit` of the `component_clause`; otherwise, denotes the offset, from the start of the first of the storage elements occupied by `C`, of the last bit occupied by `C`. This offset is measured in bits. The value of this attribute

is of the type
<universal_integer>.

Implementation Advice

5

If a component is represented using some form of pointer (such as an offset) to the actual data of the component, and this data is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data, not the pointer. If a component is allocated discontinuously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes.

14.5.3 13.5.3 Bit Ordering

1

The Bit_Order attribute specifies the interpretation of the storage place attributes.

Static Semantics

2

A bit ordering is a method of interpreting the meaning of the storage place attributes. High_Order_First (known in the vernacular as "big endian") means that the first bit of a storage element (bit 0) is the most significant bit (interpreting the sequence of bits that represent a component as an unsigned integer value). Low_Order_First (known in the vernacular as "little endian") means the opposite: the first bit is the least significant.

3

For every specific record subtype S, the following attribute is defined:

4

S'Bit_Order

Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order.

Bit_Order may be specified for specific record types via an attribute_definition_clause; the expression of such a clause shall be static. ■

5

If Word_Size = Storage_Unit, the default bit ordering is implementation defined. If Word_Size > Storage_Unit, the default bit ordering is the same as the ordering of storage elements in a word, when interpreted as an integer.

6

The storage place attributes of a component of a type are interpreted according to the bit ordering of the type.

Implementation Advice

7

The recommended level of support for the nondefault bit ordering is:

8/2

- The implementation should support the nondefault bit ordering in addition to the default bit ordering.

NOTES

9/2

13 Bit_Order clauses make it possible to write record_representation_clauses that can be ported between machines having different bit ordering. They do not guarantee transparent exchange of data between such machines.

14.6 13.6 Change of Representation

1

A type_conversion (see Section 5.6 [4.6], page 219) can be used to convert between two different representations of the same array or record. To convert an array from one representation to another, two array types need to be declared with matching component subtypes, and convertible index types. If one type has packing specified and the other does not, then explicit conversion can be used to pack or unpack an array.

2

To convert a record from one representation to another, two record types with a common ancestor type need to be declared, with no inherited subprograms. Distinct representations can then be specified for the record types, and explicit conversion between the types can be used to effect a change in representation.

Examples

3

<Example of change of representation:>

4

```
--< Packed_Descriptor and Descriptor are two different types>  
--< with identical characteristics, apart from their>  
--< representation>
```

5

```
type Descriptor is  
  record  
    --< components of a descriptor>  
  end record;
```

6

```
type Packed_Descriptor is new Descriptor;
```

7

```
for Packed_Descriptor use
  record
    --< component clauses for some or for all components>
  end record;
```

8

```
<-- Change of representation can now be accomplished by explicit type conversions
```

9

```
D : Descriptor;
P : Packed_Descriptor;
```

10

```
P := Packed_Descriptor(D); --< pack D>
D := Descriptor(P);      --< unpack P>
```

14.7 13.7 The Package System

1

For each implementation there is a library package called System which includes the definitions of certain configuration-dependent characteristics.

Static Semantics

2

The following language-defined library package exists:

3/2

```
package System is
  pragma Pure(System);
```

4

```
  type
    Name is <implementation-defined-enumeration-type>;

  System_Name : constant Name := <implementation-defined>;
```

5

```
  --< System-Dependent Named Numbers:>
```

6

```
  Min_Int          : constant := <root_integer>'First;
```

```

7      Max_Int          : constant := <root_integer>'Last;

      Max_Binary_Modulus : constant := <implementation-defined>;
      Max_Nonbinary_Modulus : constant := <implementation-defined>;
8

      Max_Base_Digits    : constant := <root_real>'Digits;
      Max_Digits         : constant := <implementation-defined>;
9

      Max_Mantissa       : constant := <implementation-defined>;
      Fine_Delta         : constant := <implementation-defined>;
10

      Tick               : constant := <implementation-defined>;
11

      --< Storage-related Declarations:>
12

      type
      Address is <implementation-defined>;

      Null_Address : constant Address;
13

      Storage_Unit : constant := <implementation-defined>;

      Word_Size    : constant := <implementation-defined> * Storage_Unit;

      Memory_Size  : constant := <implementation-defined>;
14

      --<

```

```

Address Comparison:>
  function "<" (Left, Right : Address) return Boolean;
  function "<=" (Left, Right : Address) return Boolean;
  function ">" (Left, Right : Address) return Boolean;
  function ">=" (Left, Right : Address) return Boolean;
  function "=" (Left, Right : Address) return Boolean;
-- function "/=" (Left, Right : Address) return Boolean;
--< "/=" is implicitly defined>
pragma Convention(Intrinsic, "<");
... --< and so on for all language-defined subprograms in this package>■

```

15/2

```

--< Other System-Dependent Declarations:>
  type
  Bit_Order is (
  High_Order_First,
  Low_Order_First);

Default_Bit_Order : constant Bit_Order := <implementation-defined>;

```

16

```

--< Priority-related declarations (see Section 18.1 [D.1], page 975):>■
  subtype
  Any_Priority is Integer range <implementation-defined>;
  subtype
  Priority is Any_Priority range Any_Priority'First ..
  <implementation-defined>;
  subtype
  Interrupt_Priority is Any_Priority range Priority'Last+1 ..
  Any_Priority'Last;

```

17

```

Default_Priority : constant Priority :=
  (Priority'First + Priority'Last)/2;

```

18

```

private
  ... -- <not specified by the language>
end System;

```

19

Name is an enumeration subtype. Values of type Name are the names of alternative machine configurations handled by the implementation. System_Name represents the current machine configuration.

20

The named numbers `Fine_Delta` and `Tick` are of the type `<universal_real>`; the others are of the type `<universal_integer>`.

21

The meanings of the named numbers are:

22

`Min_Int`

The smallest (most negative) value allowed for the expressions of a `signed_integer_-type_definition` (see [S0042], page 95).

23

`Max_Int`

The largest (most positive) value allowed for the expressions of a `signed_integer_-type_definition` (see [S0042], page 95).

24

`Max_Binary_Modulus`

A power of two such that it, and all lesser positive powers of two, are allowed as the modulus of a `modular_type_definition`.

25

`Max_Nonbinary_Modulus`

A value such that it, and all lesser positive integers, are allowed as the modulus of a `modular_type_definition`.

26

`Max_Base_Digits`

The largest value allowed for the

requested decimal precision in a floating_point_definition (see [S0045], page 103).

27

Max_Digits

The largest value allowed for the requested decimal precision in a floating_point_definition (see [S0045], page 103) that has no real_range_specification (see [S0046], page 103). Max_Digits is less than or equal to Max_Base_Digits.

28

Max_Mantissa

The largest possible number of binary digits in the mantissa of machine numbers of a user-defined ordinary fixed point type. (The mantissa is defined in Chapter 21 [Annex G], page 1083.)

29

Fine_Delta

The smallest delta allowed in an ordinary_fixed_point_definition that has the real_range_specification (see [S0046], page 103) range -1.0 .. 1.0 .

30

Tick

A period in seconds approximating the real time interval during which the value of `Calendar.Clock` remains constant.

31
`Storage_Unit`

The number of bits per storage element.

32
`Word_Size`

The number of bits per word.

33
`Memory_Size`

An implementation–defined value that is intended to reflect the memory size of the configuration in storage elements.

34/2
Address is a definite, nonlimited type with prelaborable initialization (see Section 11.2.1 [10.2.1], page 413). Address represents machine addresses capable of addressing individual storage elements. `Null_Address` is an address that is distinct from the address of any object or program unit.

35/2
`Default_Bit_Order` shall be a static constant. See Section 14.5.3 [13.5.3], page 508, for an explanation of `Bit_Order` and `Default_Bit_Order`.

Implementation Permissions

36/2
An implementation may add additional implementation–defined declarations to package `System` and its children. However, it is usually better for the implementation to provide additional functionality via implementation–defined children of `System`.

Implementation Advice

37
Address should be a private type.

NOTES

38

14 There are also some language–defined child packages of `System` defined elsewhere.

14.7.1 13.7.1 The Package System.Storage_Elements

Static Semantics

1

The following language-defined library package exists:

2/2

```
package System.Storage_Elements is
  pragma Pure(Storage_Elements);

  type
    Storage_Offset is range <implementation-defined>;

  subtype
    Storage_Count is Storage_Offset range 0..Storage_Offset'Last;

  type
    Storage_Element is mod <implementation-defined>;
    for Storage_Element'Size use Storage_Unit;
    type
    Storage_Array is array
      (Storage_Offset range <>) of aliased Storage_Element;
    for Storage_Array'Component_Size use Storage_Unit;

  --<
  Address Arithmetic:>

  function "+"(Left : Address; Right : Storage_Offset)
    return Address;
  function "+"(Left : Storage_Offset; Right : Address)
    return Address;
  function "-"(Left : Address; Right : Storage_Offset)
    return Address;
  function "-"(Left, Right : Address)
    return Storage_Offset;

  function "mod"(Left : Address; Right : Storage_Offset)
```

3

4

5

6

7

8

```

    return Storage_Offset;
9
    --< Conversion to/from integers:>
10
    type
    Integer_Address is <implementation-defined>;
    function
    To_Address(Value : Integer_Address) return Address;
    function
    To_Integer(Value : Address) return Integer_Address;
11
    pragma Convention(Intrinsic, "+");
    <-- ...and so on for all language-defined subprograms declared in this pac
end System.Storage_Elements;

```

12
Storage_Element represents a storage element. Storage_Offset represents an offset in storage elements. Storage_Count represents a number of storage elements. Storage_Array represents a contiguous sequence of storage elements.

13
Integer_Address is a (signed or modular) integer subtype. To_Address and To_Integer convert back and forth between this type and Address.

Implementation Requirements

14
Storage_Offset'Last shall be greater than or equal to Integer'Last or the largest possible storage offset, whichever is smaller. Storage_Offset'First shall be <= (-Storage_Offset'Last).

Implementation Permissions

15/2
<This paragraph was deleted.>

Implementation Advice

16
Operations in System and its children should reflect the target environment semantics as closely as is reasonable. For example, on most machines, it makes sense for address arithmetic to "wrap around." Operations that do not make sense should raise Program_Error.

14.7.2 13.7.2 The Package System.Address_To_Access_Conversions

Static Semantics

1
The following language-defined generic library package exists:

2

```

generic
    type Object(<>) is limited private;
package System.Address_To_Access_Conversions is
    pragma Preelaborate(Address_To_Access_Conversions);

```

3

```

        type Object_Pointer is access all Object;
        function
To_Pointer(Value : Address) return Object_Pointer;
        function
To_Address(Value : Object_Pointer) return Address;

```

4

```

        pragma Convention(Intrinsic, To_Pointer);
        pragma Convention(Intrinsic, To_Address);
end System.Address_To_Access_Conversions;

```

5/2

The `To_Pointer` and `To_Address` subprograms convert back and forth between values of types `Object_Pointer` and `Address`. `To_Pointer(X'Address)` is equal to `X'Unchecked_Access` for any `X` that allows `Unchecked_Access`. `To_Pointer(Null_Address)` returns `null`. For other addresses, the behavior is unspecified. `To_Address(null)` returns `Null_Address`. `To_Address(Y)`, where `Y /= null`, returns `Y.all'Address`.

Implementation Permissions

6

An implementation may place restrictions on instantiations of `Address_To_Access_Conversions`. ■

14.8 13.8 Machine Code Insertions

1

A machine code insertion can be achieved by a call to a subprogram whose `sequence_of_statements` contains `code_statements`.

Syntax

2

```
code_statement ::= qualified_expression;
```

3

A `code_statement` is only allowed in the `handled_sequence_of_statements` (see [S0247], page 420) of a `subprogram_body` (see [S0162], page 261). If a `subprogram_body` (see [S0162], page 261) contains any `code_statement` (see [S0294], page 518)s, then within this `subprogram_body` (see [S0162], page 261) the only allowed form of statement is a `code_statement` (see [S0294], page 518) (labeled or not), the only allowed `declarative_item` (see [S0087], page 175)s are

use_clause (see [S0179], page 314)s, and no exception_handler (see [S0248], page 420) is allowed (comments and pragmas are allowed as usual).

Name Resolution Rules

4

The qualified_expression is expected to be of any type.

Legality Rules

5

The qualified_expression shall be of a type declared in package System.Machine_Code.

6

A code_statement shall appear only within the scope of a with_clause that mentions package System.Machine_Code.

Static Semantics

7

The contents of the library package System.Machine_Code (if provided) are implementation defined. The meaning of code_statements is implementation defined. Typically, each qualified_expression represents a machine instruction or assembly directive.

Implementation Permissions

8

An implementation may place restrictions on code_statements. An implementation is not required to provide package System.Machine_Code.

NOTES

9

15 An implementation may provide implementation-defined pragmas specifying register conventions and calling conventions.

10/2

16 Machine code functions are exempt from the rule that a return statement is required. In fact, return statements are forbidden, since only code_statements are allowed.

11

17 Intrinsic subprograms (see Section 7.3.1 [6.3.1], page 263, "Section 7.3.1 [6.3.1], page 263, Conformance Rules") can also be used to achieve machine code insertions. Interface to assembly language can be achieved using the features in Chapter 16 [Annex B], page 894, "Chapter 16 [Annex B], page 894, Interface to Other Languages".

Examples

12

<Example of a code statement:>

13

```
M : Mask;  
procedure Set_Mask; pragma Inline(Set_Mask);
```

14

```
procedure Set_Mask is  
  use System.Machine_Code; --< assume "with System.Machine_Code;" appears somewhere  
begin  
  SI_Format'(Code => SSM, B => M'Base_Reg, D => M'Disp);  
  --< Base_Reg and Disp are implementation-defined attributes>  
end Set_Mask;
```

14.9 13.9 Unchecked Type Conversions

1

An unchecked type conversion can be achieved by a call to an instance of the generic function `Unchecked_Conversion`.

Static Semantics

2

The following language-defined generic library function exists:

3

```
generic  
  type Source(<>) is limited private;  
  type Target(<>) is limited private;  
  
function Ada.Unchecked_Conversion(S : Source) return Target;  
pragma Convention(Intrinsic, Ada.Unchecked_Conversion);  
pragma Pure(Ada.Unchecked_Conversion);
```

Dynamic Semantics

4

The size of the formal parameter `S` in an instance of `Unchecked_Conversion` is that of its subtype. This is the actual subtype passed to `Source`, except when the actual is an unconstrained composite subtype, in which case the subtype is constrained by the bounds or discriminants of the value of the actual expression passed to `S`.

5

If all of the following are true, the effect of an unchecked conversion is to return the value of an object of the target subtype whose representation is the same as that of the source object `S`:

6

- $S'Size = Target'Size$.

7

- $S'Alignment = Target'Alignment$.

8

- The target subtype is not an unconstrained composite subtype.

9

- S and the target subtype both have a contiguous representation.

10

- The representation of S is a representation of an object of the target subtype.

11/2

Otherwise, if the result type is scalar, the result of the function is implementation defined, and can have an invalid representation (see Section 14.9.1 [13.9.1], page 522). If the result type is nonscalar, the effect is implementation defined; in particular, the result can be abnormal (see Section 14.9.1 [13.9.1], page 522).

Implementation Permissions

12

An implementation may return the result of an unchecked conversion by reference, if the Source type is not a by-copy type. In this case, the result of the unchecked conversion represents simply a different (read-only) view of the operand of the conversion.

13

An implementation may place restrictions on Unchecked_Conversion.

Implementation Advice

14/2

Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data.

15

The implementation should not generate unnecessary run-time checks to ensure that the representation of S is a representation of the target type. It should take advantage of the permission to return by reference when possible. Restrictions on unchecked conversions should be avoided unless required by the target environment.

16

The recommended level of support for unchecked conversions is:

17

- Unchecked conversions should be supported and should be reversible in the cases where this clause defines the result. To enable meaningful use of unchecked conversion, a contiguous representation should be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in this paragraph, and for record subtypes without discriminants whose component subtypes are described in this paragraph.

14.9.1 13.9.1 Data Validity

1

Certain actions that can potentially lead to erroneous execution are not directly erroneous, but instead can cause objects to become `<abnormal>`. Subsequent uses of abnormal objects can be erroneous.

2

A scalar object can have an `<invalid representation>`, which means that the object's representation does not represent any value of the object's subtype. The primary cause of invalid representations is uninitialized variables.

3

Abnormal objects and invalid representations are explained in this subclause.

Dynamic Semantics

4

When an object is first created, and any explicit or default initializations have been performed, the object and all of its parts are in the `<normal>` state. Subsequent operations generally leave them normal. However, an object or part of an object can become `<abnormal>` in the following ways:

5

- An assignment to the object is disrupted due to an abort (see Section 10.8 [9.8], page 385) or due to the failure of a language-defined check (see Section 12.6 [11.6], page 448).

6/2

- The object is not scalar, and is passed to an in out or out parameter of an imported procedure, the Read procedure of an instance of `Sequential_IO`, `Direct_IO`, or `Storage_IO`, or the stream attribute `T'Read`, if after return from the procedure the representation of the parameter does not represent a value of the parameter's subtype.

6.1/2

- The object is the return object of a function call of a nonscalar type, and the function is an imported function, an instance of `Unchecked_Conversion`, or the stream attribute `T'Input`, if after return from the function the representation of the return object does not represent a value of the function's subtype.

6.2/2

For an imported object, it is the programmer's responsibility to ensure that the object remains in a normal state.

7

Whether or not an object actually becomes abnormal in these cases is not specified. An abnormal object becomes normal again upon successful completion of an assignment to the object as a whole.

Erroneous Execution

8

It is erroneous to evaluate a primary that is a name denoting an abnormal object, or to evaluate a prefix that denotes an abnormal object.

Bounded (Run-Time) Errors

9

If the representation of a scalar object does not represent a value of the object's subtype (perhaps because the object was not initialized), the object is said to have an <invalid representation>. It is a bounded error to evaluate the value of such an object. If the error is detected, either `Constraint_Error` or `Program_Error` is raised. Otherwise, execution continues using the invalid representation. The rules of the language outside this subclause assume that all objects have valid representations. The semantics of operations on invalid representations are as follows:

10

- If the representation of the object represents a value of the object's type, the value of the type is used.

11

- If the representation of the object does not represent a value of the object's type, the semantics of operations on such representations is implementation-defined, but does not by itself lead to erroneous or unpredictable execution, or to other objects becoming abnormal.

Erroneous Execution

12/2

A call to an imported function or an instance of `Unchecked_Conversion` is erroneous if the result is scalar, the result object has an invalid representation, and the result is used other than as the expression of an `assignment_statement` or an `object_declaration`, or as the prefix of a `Valid` attribute. If such a result object is used as the source of an assignment, and the assigned value is an invalid representation for the target of the assignment, then any use of the target object prior to a further assignment to the target object, other than as the prefix of a `Valid` attribute reference, is erroneous.

13

The dereference of an access value is erroneous if it does not designate an object of an appropriate type or a subprogram with an appropriate profile, if it designates a nonexistent object, or if it is an `access-to-variable` value that designates a constant object. Such an access value can exist, for example, because of `Unchecked_Deallocation`, `Unchecked_Access`, or `Unchecked_Conversion`.

NOTES

14

18 Objects can become abnormal due to other kinds of actions that directly update the object's representation; such actions are generally considered directly erroneous, however.

14.9.2 13.9.2 The Valid Attribute

1

The Valid attribute can be used to check the validity of data produced by unchecked conversion, input, interface to foreign languages, and the like.

Static Semantics

2

For a prefix X that denotes a scalar object (after any implicit dereference), the following attribute is defined:

3

X'Valid

Yields True if and only if the object denoted by X is normal and has a valid representation. The value of this attribute is of the predefined type Boolean.

NOTES

4

19 Invalid data can be created in the following cases (not counting erroneous or unpredictable execution):

5

- an uninitialized scalar object,

6

- the result of an unchecked conversion,

7

- input,

8

- interface to another language (including machine code),

9

- aborting an assignment,

10

- disrupting an assignment due to the failure of a language-defined check (see Section 12.6 [11.6], page 448), and

11

- use of an object whose Address has been specified.

12

20 X'Valid is not considered to be a read of X; hence, it is not an error to check the validity of invalid data.

13/2

21 The Valid attribute may be used to check the result of calling an instance of Unchecked_Conversion (or any other operation that can return invalid values). However, an exception handler should also be provided because implementations are permitted to raise Constraint_Error or Program_Error if they detect the use of an invalid representation (see Section 14.9.1 [13.9.1], page 522).

14.10 13.10 Unchecked Access Value Creation

1

The attribute Unchecked_Access is used to create access values in an unsafe manner -- the programmer is responsible for preventing "dangling references."

Static Semantics

2

The following attribute is defined for a prefix X that denotes an aliased view of an object:

3

X'Unchecked_Access

All rules and semantics that apply to X'Access (see Section 4.10.2 [3.10.2], page 164) apply also to X'Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package.

NOTES

4

22 This attribute is provided to support the situation where a local object is to be inserted into a global linked data structure, when

the programmer knows that it will always be removed from the data structure prior to exiting the object's scope. The Access attribute would be illegal in this case (see Section 4.10.2 [3.10.2], page 164, "Section 4.10.2 [3.10.2], page 164, Operations of Access Types").

5

23 There is no Unchecked_Access attribute for subprograms.

14.11 13.11 Storage Management

1

Each access-to-object type has an associated storage pool. The storage allocated by an allocator comes from the pool; instances of Unchecked_Deallocation return storage to the pool. Several access types can share the same pool.

2/2

A storage pool is a variable of a type in the class rooted at Root_Storage_Pool, which is an abstract limited controlled type. By default, the implementation chooses a <standard storage pool> for each access-to-object type. The user may define new pool types, and may override the choice of pool for an access-to-object type by specifying Storage_Pool for the type.

Legality Rules

3

If Storage_Pool is specified for a given access type, Storage_Size shall not be specified for it.

Static Semantics

4

The following language-defined library package exists:

5

```
with Ada.Finalization;
with System.Storage_Elements;
```

```
package System.Storage_Pools is
  pragma Preelaborate(System.Storage_Pools);
```

6/2

```
  type
    Root_Storage_Pool is
      abstract new Ada.Finalization.Limited_Controlled with private;
      pragma Preelaborable_Initialization(Root_Storage_Pool);
```

7

```
  procedure
    Allocate(
```

```
Pool : in out Root_Storage_Pool;  
Storage_Address : out Address;  
Size_In_Storage_Elements : in Storage_Elements.Storage_Count;  
Alignment : in Storage_Elements.Storage_Count) is abstract;
```

8

```
procedure  
Deallocate(  
    Pool : in out Root_Storage_Pool;  
    Storage_Address : in Address;  
    Size_In_Storage_Elements : in Storage_Elements.Storage_Count;  
    Alignment : in Storage_Elements.Storage_Count) is abstract;
```

9

```
function  
Storage_Size(Pool : Root_Storage_Pool)  
    return Storage_Elements.Storage_Count is abstract;
```

10

```
private  
    ... -- <not specified by the language>  
end System.Storage_Pools;
```

11

A <storage pool type> (or <pool type>) is a descendant of `Root_Storage_Pool`. The <elements> of a storage pool are the objects allocated in the pool by allocators.

12/2

For every access-to-object subtype `S`, the following representation attributes are defined:

13

`S'Storage_Pool`

Denotes the storage pool of the type of `S`. The type of this attribute is `Root_Storage_Pool'Class`.

14

`S'Storage_Size`

Yields the result of calling `Storage_Size(S'Storage_Pool)`, which is intended to be a measure of the number of storage elements reserved for the pool. The type

of this attribute is
<universal_integer>.

15

Storage_Size or Storage_Pool may be specified for a non-derived access-to-object type via an attribute_definition_clause (see [S0286], page 487); the name in a Storage_Pool clause shall denote a variable.

16

An allocator of type T allocates storage from T's storage pool. If the storage pool is a user-defined object, then the storage is allocated by calling Allocate, passing T'Storage_Pool as the Pool parameter. The Size_In_Storage_Elements parameter indicates the number of storage elements to be allocated, and is no more than D'Max_Size_In_Storage_Elements, where D is the designated subtype. The Alignment parameter is D'Alignment. The result returned in the Storage_Address parameter is used by the allocator as the address of the allocated storage, which is a contiguous block of memory of Size_In_Storage_Elements storage elements. Any exception propagated by Allocate is propagated by the allocator.

17

If Storage_Pool is not specified for a type defined by an access_to_object_definition, then the implementation chooses a standard storage pool for it in an implementation-defined manner. In this case, the exception Storage_Error is raised by an allocator if there is not enough storage. It is implementation defined whether or not the implementation provides user-accessible names for the standard pool type(s).

18

If Storage_Size is specified for an access type, then the Storage_Size of this pool is at least that requested, and the storage for the pool is reclaimed when the master containing the declaration of the access type is left. If the implementation cannot satisfy the request, Storage_Error is raised at the point of the attribute_definition_clause (see [S0286], page 487). If neither Storage_Pool nor Storage_Size are specified, then the meaning of Storage_Size is implementation defined.

19

If Storage_Pool is specified for an access type, then the specified pool is used.

20

The effect of calling Allocate and Deallocate for a standard storage pool directly (rather than implicitly via an allocator or an instance of Unchecked_Deallocation) is unspecified.

Erroneous Execution

21

If Storage_Pool is specified for an access type, then if Allocate can satisfy the request, it should allocate a contiguous block of memory, and return the address of the first storage element in Storage_Address. The block should contain Size_In_Storage_Elements storage elements, and should be aligned according to Alignment. The allocated storage should not be used for any other purpose while the pool element remains in existence. If the request cannot be satisfied, then Allocate should propagate an exception (such as Storage_Error). If Allocate behaves in any other manner, then the program execution is erroneous.

Documentation Requirements

22

An implementation shall document the set of values that a user-defined Allocate procedure needs to accept for the Alignment parameter. An implementation shall document how the standard storage pool is chosen, and how storage is allocated by standard storage pools.

Implementation Advice

23

An implementation should document any cases in which it dynamically allocates heap storage for a purpose other than the evaluation of an allocator.

24

A default (implementation-provided) storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects.

25/2

The storage pool used for an allocator of an anonymous access type should be determined as follows:

25.1/2

- If the allocator is defining a coextension (see Section 4.10.2 [3.10.2], page 164) of an object being created by an outer allocator, then the storage pool used for the outer allocator should also be used for the coextension;

25.2/2

- For other access discriminants and access parameters, the storage pool should be created at the point of the allocator, and be reclaimed when the allocated object becomes inaccessible;

25.3/2

- Otherwise, a default storage pool should be created at the point where the anonymous access type is elaborated; such a storage pool need not support deallocation of individual objects.

NOTES

26

24 A user-defined storage pool type can be obtained by extending the Root_Storage_Pool type, and overriding the primitive subprograms Allocate, Deallocate, and Storage_Size. A user-defined storage pool can then be obtained by declaring an object of the type extension. The user can override Initialize and Finalize if there is any need for non-trivial initialization and finalization for a user-defined pool type. For example, Finalize might reclaim blocks of storage that are allocated separately from the pool object itself.

27

25 The writer of the user-defined allocation and deallocation procedures, and users of allocators for the associated access type, are responsible for dealing with any interactions with tasking. In particular:

28

- If the allocators are used in different tasks, they require mutual exclusion.

29

- If they are used inside protected objects, they cannot block.

30

- If they are used by interrupt handlers (see Section 17.3 [C.3], page 951, "Section 17.3 [C.3], page 951, Interrupt Support"), the mutual exclusion mechanism has to work properly in that context.

31

26 The primitives `Allocate`, `Deallocate`, and `Storage_Size` are declared as abstract (see Section 4.9.3 [3.9.3], page 149), and therefore they have to be overridden when a new (non-abstract) storage pool type is declared.

Examples

32

To associate an access type with a storage pool object, the user first declares a pool object of some type derived from `Root_Storage_Pool`. Then, the user defines its `Storage_Pool` attribute, as follows:

33

```
Pool_Object : Some_Storage_Pool_Type;
```

34

```
type T is access Designated;  
for T'Storage_Pool use Pool_Object;
```

35

Another access type may be added to an existing storage pool, via:

36

```
for T2'Storage_Pool use T'Storage_Pool;
```

37

The semantics of this is implementation defined for a standard storage pool.

38

As usual, a derivative of `Root_Storage_Pool` may define additional operations. For example, presuming that `Mark_Release_Pool_Type` has two additional operations, `Mark` and `Release`, the following is a possible use:

39/1

```
type Mark_Release_Pool_Type
  (Pool_Size : Storage_Elements.Storage_Count;
   Block_Size : Storage_Elements.Storage_Count)
  is new Root_Storage_Pool with private;
```

40

```
...
```

41

```
MR_Pool : Mark_Release_Pool_Type (Pool_Size => 2000,
                                   Block_Size => 100);
```

42

```
type Acc is access ...;
for Acc'Storage_Pool use MR_Pool;
...
```

43

```
Mark(MR_Pool);
... --< Allocate objects using "new Designated(...)".>
Release(MR_Pool); --< Reclaim the storage.>
```

14.11.1 13.11.1 The `Max_Size_In_Storage_Elements` Attribute

1

The `Max_Size_In_Storage_Elements` attribute is useful in writing user-defined pool types.

Static Semantics

2

For every subtype `S`, the following attribute is defined:

3/2

`S'Max_Size_In_Storage_Elements`

Denotes the maximum value for `Size_In_Storage_Elements` that could be requested by the implementation via `Allocate` for an access type whose

designated subtype is S. For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then this accounts for any calls on Allocate that could be performed to provide space for such coextensions. The value of this attribute is of type <universal_integer>.

14.11.2 13.11.2 Unchecked Storage Deallocation

1

Unchecked storage deallocation of an object designated by a value of an access type is achieved by a call to an instance of the generic procedure Unchecked_Deallocation.

Static Semantics

2

The following language-defined generic library procedure exists:

3

```
generic
  type Object(<>) is limited private;
  type Name is access Object;

  procedure Ada.Unchecked_Deallocation(X : in out Name);
  pragma Convention(Intrinsic, Ada.Unchecked_Deallocation);
  pragma Preelaborate(Ada.Unchecked_Deallocation);
```

Dynamic Semantics

4

Given an instance of Unchecked_Deallocation declared as follows:

5

```
procedure Free is
  new Ada.Unchecked_Deallocation(
    <object_subtype_name>, <access_to_variable_subtype_name>);
```

6

Procedure `Free` has the following effect:

7

1. After executing `Free(X)`, the value of `X` is null.

8

2. `Free(X)`, when `X` is already equal to null, has no effect.

9/2

3. `Free(X)`, when `X` is not equal to null first performs finalization of the object designated by `X` (and any coextensions of the object — see Section 4.10.2 [3.10.2], page 164), as described in Section 8.6.1 [7.6.1], page 299. It then deallocates the storage occupied by the object designated by `X` (and any coextensions). If the storage pool is a user-defined object, then the storage is deallocated by calling `Deallocate`, passing `<access_to_variable_subtype_name>'Storage.Pool` as the `Pool` parameter. `Storage_Address` is the value returned in the `Storage_Address` parameter of the corresponding `Allocate` call. `Size_In_Storage_Elements` and `Alignment` are the same values passed to the corresponding `Allocate` call. There is one exception: if the object being freed contains tasks, the object might not be deallocated.

10/2

After `Free(X)`, the object designated by `X`, and any subcomponents (and coextensions) thereof, no longer exist; their storage can be reused for other purposes.

Bounded (Run-Time) Errors

11

It is a bounded error to free a discriminated, unterminated task object. The possible consequences are:

12

- No exception is raised.

13

- `Program_Error` or `Tasking_Error` is raised at the point of the deallocation.

14

- `Program_Error` or `Tasking_Error` is raised in the task the next time it references any of the discriminants.

15

In the first two cases, the storage for the discriminants (and for any enclosing object if it is designated by an access discriminant of the task) is not reclaimed prior to task termination.

Erroneous Execution

16

Evaluating a name that denotes a nonexistent object is erroneous. The execution of a call to an instance of `Unchecked_Deallocation` is erroneous if the object was created other than by an allocator for an access type whose pool is `Name'Storage_Pool`.

Implementation Advice

17

For a standard storage pool, `Free` should actually reclaim the storage.

NOTES

18

27 The rules here that refer to `Free` apply to any instance of `Unchecked_Deallocation`.

19

28 `Unchecked_Deallocation` cannot be instantiated for an access-to-constant type. This is implied by the rules of Section 13.5.4 [12.5.4], page 468.

14.11.3 13.11.3 Pragma Controlled

1

`Pragma Controlled` is used to prevent any automatic reclamation of storage (garbage collection) for the objects created by allocators of a given access type.

Syntax

2

The form of a pragma `Controlled` is as follows:

3

```
pragma Controlled(<first_subtype_>local_name);
```

Legality Rules

4

The `<first_subtype_>local_name` of a pragma `Controlled` shall denote a non-derived access subtype.

Static Semantics

5

A pragma `Controlled` is a representation pragma that specifies the `<controlled>` aspect of representation.

6

`<Garbage collection>` is a process that automatically reclaims storage, or moves objects to a different address, while the objects still exist.

7

If a pragma `Controlled` is specified for an access type with a standard storage pool, then garbage collection is not performed for objects in that pool.

Implementation Permissions

8

An implementation need not support garbage collection, in which case, a pragma Controlled has no effect.

14.12 13.12 Pragma Restrictions

1

A pragma Restrictions expresses the user's intent to abide by certain restrictions. This may facilitate the construction of simpler run-time environments.

Syntax

2

The form of a pragma Restrictions is as follows:

3

```
pragma Restrictions(restriction{, restriction});
```

4/2

```
restriction ::= <restriction_>identifier  
             | <restriction_parameter_>identifier => restriction_parameter_argument
```

4.1/2

```
restriction_parameter_argument ::= name | expression  
                               Name Resolution Rules
```

5

Unless otherwise specified for a particular restriction, the expression is expected to be of any integer type.

Legality Rules

6

Unless otherwise specified for a particular restriction, the expression shall be static, and its value shall be nonnegative.

Static Semantics

7/2

The set of restrictions is implementation defined.

Post-Compilation Rules

8

A pragma Restrictions is a configuration pragma; unless otherwise specified for a particular restriction, a partition shall obey the restriction if a pragma Restrictions applies to any compilation unit included in the partition.

8.1/1

For the purpose of checking whether a partition contains constructs that violate any restriction (unless specified otherwise for a particular restriction):

8.2/1

- Generic instances are logically expanded at the point of instantiation;

8.3/1

- If an object of a type is declared or allocated and not explicitly initialized, then all expressions appearing in the definition for the type and any of its ancestors are presumed to be used;

8.4/1

- A `default_expression` for a formal parameter or a generic formal object is considered to be used if and only if the corresponding actual parameter is not provided in a given call or instantiation.

Implementation Permissions

9

An implementation may place limitations on the values of the expression that are supported, and limitations on the supported combinations of restrictions. The consequences of violating such limitations are implementation defined.

9.1/1

An implementation is permitted to omit restriction checks for code that is recognized at compile time to be unreachable and for which no code is generated.

9.2/1

Whenever enforcement of a restriction is not required prior to execution, an implementation may nevertheless enforce the restriction prior to execution of a partition to which the restriction applies, provided that every execution of the partition would violate the restriction.

NOTES

10/2

29 Restrictions intended to facilitate the construction of efficient tasking run-time systems are defined in Section 18.7 [D.7], page 1001. Restrictions intended for use when constructing high integrity systems are defined in Section 22.4 [H.4], page 1158.

11

30 An implementation has to enforce the restrictions in cases where enforcement is required, even if it chooses not to take advantage of the restrictions in terms of efficiency.

14.12.1 13.12.1 Language-Defined Restrictions

Static Semantics

1/2

The following `<restriction_>identifiers` are language-defined (additional restrictions are defined in the Specialized Needs Annexes):

2/2

`No_Implementation_Attributes`

There are no implementation-defined attributes. This restriction applies only to the current compilation or environment, not the entire partition.

3/2

No_Implementation_Pragmas

There are no implementation-defined pragmas or pragma arguments. This restriction applies only to the current compilation or environment, not the entire partition.

4/2

No_Obsolescent_Features

There is no use of language features defined in Annex J. It is implementation-defined if uses of the renamings of Section 23.1 [J.1], page 1166, are detected by this restriction. This restriction applies only to the current compilation or environment, not the entire partition.

5/2

The following <restriction_parameter>identifier is language defined:

6/2

No_Dependence

Specifies a library unit on which there are no semantic dependences.

Legality Rules

7/2

The `restriction_parameter_argument` of a `No_Dependence` restriction shall be a name; the name shall have the form of a full expanded name of a library unit, but need not denote a unit present in the environment.

Post-Compilation Rules

8/2

No compilation unit included in the partition shall depend semantically on the library unit identified by the name.

14.13 13.13 Streams

1

A `<stream>` is a sequence of elements comprising values from possibly different types and allowing sequential access to these values. A `<stream type>` is a type in the class whose root type is `Streams.Root_Stream_Type`. A stream type may be implemented in various ways, such as an external sequential file, an internal buffer, or a network channel.

14.13.1 13.13.1 The Package Streams

Static Semantics

1

The abstract type `Root_Stream_Type` is the root type of the class of stream types. The types in this class represent different kinds of streams. A new stream type is defined by extending the root type (or some other stream type), overriding the `Read` and `Write` operations, and optionally defining additional primitive subprograms, according to the requirements of the particular kind of stream. The predefined stream-oriented attributes like `T'Read` and `T'Write` make dispatching calls on the `Read` and `Write` procedures of the `Root_Stream_Type`. (User-defined `T'Read` and `T'Write` attributes can also make such calls, or can call the `Read` and `Write` attributes of other types.)

2

```
package Ada.Streams is
    pragma Pure(Streams)
;
```

3/2

```
type
Root_Stream_Type is abstract tagged limited private;
    pragma Preelaborable_Initialization(Root_Stream_Type);
```

4/1

```
type
Stream_Element is mod <implementation-defined>;
type
```

```

Stream_Element_Offset is range <implementation-defined>;
  subtype
Stream_Element_Count is
  Stream_Element_Offset range 0..Stream_Element_Offset'Last;
  type
Stream_Element_Array is
  array(Stream_Element_Offset range <>) of aliased Stream_Element;

```

5

```

  procedure
Read(
  Stream : in out Root_Stream_Type;
  Item   : out Stream_Element_Array;
  Last   : out Stream_Element_Offset) is abstract;

```

6

```

  procedure
Write(
  Stream : in out Root_Stream_Type;
  Item   : in Stream_Element_Array) is abstract;

```

7

```

private
  ... -- <not specified by the language>
end Ada.Streams;

```

8/2

The Read operation transfers stream elements from the specified stream to fill the array Item. Elements are transferred until Item'Length elements have been transferred, or until the end of the stream is reached. If any elements are transferred, the index of the last stream element transferred is returned in Last. Otherwise, Item'First - 1 is returned in Last. Last is less than Item'Last only if the end of the stream is reached.

9

The Write operation appends Item to the specified stream.

Implementation Permissions

9.1/1

If Stream_Element'Size is not a multiple of System.Storage_Unit, then the components of Stream_Element_Array need not be aliased.

NOTES

10

31 See Section 15.12.1 [A.12.1], page 746, "Section 15.12.1 [A.12.1], page 746, The Package Streams.Stream_IO" for an example of extending type Root_Stream_Type.

11/2

32 If the end of stream has been reached, and Item'First is Stream_Element_Offset'First, Read will raise Constraint_Error.

14.13.2 13.13.2 Stream-Oriented Attributes

1/1

The operational attributes Write, Read, Output, and Input convert values to a stream of elements and reconstruct values from a stream.

Static Semantics

1.1/2

For every subtype S of an elementary type <T>, the following representation attribute is defined:

1.2/2

S'Stream_Size

Denotes the number of bits occupied in a stream by items of subtype S. Hence, the number of stream elements required per item of elementary type <T> is:

1.3/2

$\langle T \rangle \text{'Stream_Size} / \text{Ada.Streams.Stream_Element'Size}$ ■

1.4/2

The value of this attribute is of type <universal_integer> and is a multiple of Stream_Element'Size.

1.5/2

Stream_Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, nonnegative, and a multiple of Stream_Element'Size. ■

Implementation Advice

1.6/2

If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size.

1.7/2

The recommended level of support for the Stream_Size attribute is:

1.8/2

- A Stream_Size clause should be supported for a discrete or fixed point type <T> if the specified Stream_Size is a multiple of Stream_Element_Size and is no less than the size of the first subtype of <T>, and no greater than the size of the largest type of the same elementary class (signed integer, modular integer, enumeration, ordinary fixed point, or decimal fixed point).

Static Semantics

2

For every subtype S of a specific type <T>, the following attributes are defined.

3

S'Write

S'Write denotes
a procedure with
the following
specification:

4/2

```
procedure S'Write(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item> : in <T>)
```

5

S'Write writes the
value of <Item> to
<Stream>.

6

S'Read

S'Read denotes
a procedure with
the following
specification:

7/2

```
procedure S'Read(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item> : out <T>)
```

<Stream> : not null access Ada.Streams.Root_Stream
<Item> : out <T>)

8

S'Read reads the
value of <Item> from
<Stream>.

8.1/2

For an untagged derived type, the Write (resp. Read) attribute is inherited according to the rules given in Section 14.1 [13.1], page 481, if the attribute is available for the parent type at the point where <T> is declared. For a tagged derived type, these attributes are not inherited, but rather the default implementations are used.

8.2/2

The default implementations of the Write and Read attributes, where available, execute as follows:

9/2

For elementary types, Read reads (and Write writes) the number of stream elements implied by the Stream_Size for the type <T>; the representation of those stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in canonical order, which is last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if <T> is an array type. If <T> is a discriminated type, discriminants are included only if they have defaults. If <T> is a tagged type, the tag is not included. For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of the parent type or any progenitor type of <T> is available anywhere within the immediate scope of <T>, and the attribute of the parent type or the type of any of the extension components is not available at the freezing point of <T>, then the attribute of <T> shall be directly specified.

9.1/2

Constraint_Error is raised by the predefined Write attribute if the value of the elementary item is outside the range of values representable using Stream_Size bits. For a signed integer type, an enumeration type, or a fixed point type, the range is unsigned only if the integer code for the lower bound of the first subtype is nonnegative, and a (symmetric) signed range that covers all values of the first subtype would require more than Stream_Size bits; otherwise the range is signed.

10

For every subtype S'Class of a class-wide type <T>'Class:

11

S'Class'Write

S'Class'Write
denotes a procedure
with the following
specification:

12/2

```
procedure S'Class'Write(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item>   : in <T>'Class)
```

13

Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item.

14

S'Class'Read

S'Class'Read denotes a procedure with the following specification:

15/2

```
procedure S'Class'Read(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item>   : out <T>'Class)
```

16

Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item.

Implementation Advice

17/2

<This paragraph was deleted.>

Static Semantics

18

For every subtype S of a specific type <T>, the following attributes are defined.

19

S'Output

S'Output denotes a procedure with the following specification:

20/2

```
procedure S'Output(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item> : in <T>)
```

21

S'Output writes the value of <Item> to <Stream>, including any bounds or discriminants.

22

S'Input

S'Input denotes a function with the following specification:

23/2

```
function S'Input(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  return <T>
```

24

S'Input reads and returns one value from <Stream>, using any bounds or discriminants written by a corresponding S'Output to determine how much to read.

25/2

For an untagged derived type, the Output (resp. Input) attribute is inherited according to the rules given in Section 14.1 [13.1], page 481, if the attribute is available for the parent type at the point where <T> is declared. For a tagged derived type, these attributes are not inherited, but rather the default implementations are used.

25.1/2

The default implementations of the Output and Input attributes, where available, execute as follows:

26

- If $\langle T \rangle$ is an array type, `S'Output` first writes the bounds, and `S'Input` first reads the bounds. If $\langle T \rangle$ has discriminants without defaults, `S'Output` first writes the discriminants (using `S'Write` for each), and `S'Input` first reads the discriminants (using `S'Read` for each).

27/2

- `S'Output` then calls `S'Write` to write the value of $\langle \text{Item} \rangle$ to the stream. `S'Input` then creates an object (with the bounds or discriminants, if any, taken from the stream), passes it to `S'Read`, and returns the value of the object. Normal default initialization and finalization take place for this object (see Section 4.3.1 [3.3.1], page 61, Section 8.6 [7.6], page 295, and Section 8.6.1 [7.6.1], page 299).

27.1/2

If $\langle T \rangle$ is an abstract type, then `S'Input` is an abstract function.

28

For every subtype `S'Class` of a class-wide type $\langle T \rangle$ 'Class:

29

`S'Class'Output`

`S'Class'Output`
denotes a procedure
with the following
specification:

30/2

```

procedure S'Class'Output(
  <Stream> : not null access Ada.Streams.Root_Stream_Type'Body;
  <Item>   : in <T>'Class)

```

31/2

First writes the external tag of $\langle \text{Item} \rangle$ to $\langle \text{Stream} \rangle$ (by calling `String'Output`($\langle \text{Stream} \rangle$, `Tags.External_Tag`($\langle \text{Item} \rangle$ 'Tag))
— see Section 4.9 [3.9], page 136) and then dispatches to the subprogram denoted by the `Output` attribute of the specific type identified by the tag. `Tag_Error` is raised if the tag of

Item identifies a type declared at an accessibility level deeper than that of S.

32

S'Class'Input

S'Class'Input denotes a function with the following specification:

33/2

```
function S'Class'Input(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  return <T>'Class
```

34/2

First reads the external tag from <Stream> and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(<Stream>), S'Tag) which might raise Tag_Error -- see Section 4.9 [3.9], page 136) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. If the specific type identified by the internal tag is not covered by <T>'Class or is abstract, Constraint_Error is raised.

35/2

In the default implementation of Read and Input for a composite type, for each scalar component that is a discriminant or whose component_declaration includes a default_expression,

a check is made that the value returned by `Read` for the component belongs to its subtype. `Constraint_Error` is raised if this check fails. For other scalar components, no check is made. For each component that is of an access type, if the implementation can detect that the value returned by `Read` for the component is not a value of its subtype, `Constraint_Error` is raised. If the value is not a value of its subtype and this error is not detected, the component has an abnormal value, and erroneous execution can result (see Section 14.9.1 [13.9.1], page 522). In the default implementation of `Read` for a composite type with defaulted discriminants, if the actual parameter of `Read` is constrained, a check is made that the discriminants read from the stream are equal to those of the actual parameter. `Constraint_Error` is raised if this check fails.

36/2

It is unspecified at which point and in which order these checks are performed. In particular, if `Constraint_Error` is raised due to the failure of one of these checks, it is unspecified how many stream elements have been read from the stream.

37/1

In the default implementation of `Read` and `Input` for a type, `End_Error` is raised if the end of the stream is reached before the reading of a value of the type is completed.

38/2

The stream-oriented attributes may be specified for any type via an `attribute_definition_clause`. The subprogram name given in such a clause shall not denote an abstract subprogram. Furthermore, if a stream-oriented attribute is specified for an interface type by an `attribute_definition_clause`, the subprogram name given in the clause shall statically denote a null procedure.

39/2

A stream-oriented attribute for a subtype of a specific type `<T>` is `<available>` at places where one of the following conditions is true:

40/2

- `<T>` is nonlimited.

41/2

- The `attribute_designator` is `Read` (resp. `Write`) and `<T>` is a limited record extension, and the attribute `Read` (resp. `Write`) is available for the parent type of `<T>` and for the types of all of the extension components.

42/2

- `<T>` is a limited untagged derived type, and the attribute was inherited for the type.

43/2

- The `attribute_designator` is `Input` (resp. `Output`), and `<T>` is a limited type, and the attribute `Read` (resp. `Write`) is available for `<T>`.

44/2

- The attribute has been specified via an `attribute_definition_clause`, and the `attribute_definition_clause` is visible.

45/2

A stream-oriented attribute for a subtype of a class-wide type `<T>'Class` is available at places where one of the following conditions is true:

46/2

- `<T>` is nonlimited;

47/2

- the attribute has been specified via an `attribute_definition_clause`, and the `attribute_definition_clause` is visible; or

48/2

- the corresponding attribute of `<T>` is available, provided that if `<T>` has a partial view, the corresponding attribute is available at the end of the visible part where `<T>` is declared.

49/2

An `attribute_reference` for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the `attribute_reference`. Furthermore, an `attribute_reference` for `<T>'Input` is illegal if `<T>` is an abstract type.

50/2

In the `parameter_and_result_profiles` for the stream-oriented attributes, the subtype of the Item parameter is the base subtype of `<T>` if `<T>` is a scalar type, and the first subtype otherwise. The same rule applies to the result of the Input attribute.

51/2

For an `attribute_definition_clause` specifying one of these attributes, the subtype of the Item parameter shall be the base subtype if scalar, and the first subtype otherwise. The same rule applies to the result of the Input function.

52/2

A type is said to `<support external streaming>` if Read and Write attributes are provided for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling the representation. A limited type supports external streaming only if it has available Read and Write attributes. A type with a part that is of an access type supports external streaming only if that access type or the type of some part that includes the access type component, has Read and Write attributes that have been specified via an `attribute_definition_clause`, and that `attribute_definition_clause` is visible. An anonymous access type does not support external streaming. All other types support external streaming.

Erroneous Execution

53/2

If the internal tag returned by `Descendant_Tag` to `T'Class'Input` identifies a type that is not library-level and whose tag has not been created, or does not exist in the partition at the time of the call, execution is erroneous.

Implementation Requirements

54/1

For every subtype <S> of a language-defined nonlimited specific type <T>, the output generated by S'Output or S'Write shall be readable by S'Input or S'Read, respectively. This rule applies across partitions if the implementation conforms to the Distributed Systems Annex.

55/2

If Constraint_Error is raised during a call to Read because of failure of one the above checks, the implementation must ensure that the discriminants of the actual parameter of Read are not modified.

Implementation Permissions

56/2

The number of calls performed by the predefined implementation of the stream-oriented attributes on the Read and Write operations of the stream type is unspecified. An implementation may take advantage of this permission to perform internal buffering. However, all the calls on the Read and Write operations of the stream type needed to implement an explicit invocation of a stream-oriented attribute must take place before this invocation returns. An explicit invocation is one appearing explicitly in the program text, possibly through a generic instantiation (see Section 13.3 [12.3], page 454).

NOTES

57

33 For a definite subtype S of a type <T>, only <T>'Write and <T>'Read are needed to pass an arbitrary value of the subtype through a stream. For an indefinite subtype S of a type <T>, <T>'Output and <T>'Input will normally be needed, since <T>'Write and <T>'Read do not pass bounds, discriminants, or tags.

58

34 User-specified attributes of S'Class are not inherited by other class-wide types descended from S.

Examples

59

<Example of user-defined Write attribute:>

60/2

```
procedure My_Write(  
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;  
  Item   : My_Integer'Base);  
for My_Integer'Write use My_Write;
```

14.14 13.14 Freezing Rules

1

This clause defines a place in the program text where each declared entity becomes "frozen." A use of an entity, such as a reference to it by name, or (for a type) an expression of the type, causes freezing of the entity in some contexts, as described below. The Legality Rules forbid certain kinds of uses of an entity in the region of text where it is frozen.

2

The <freezing> of an entity occurs at one or more places (<freezing points>) in the program text where the representation for the entity has to be fully determined. Each entity is frozen from its first freezing point to the end of the program text (given the ordering of compilation units defined in Section 11.1.4 [10.1.4], page 406).

3/1

The end of a declarative_part, protected_body, or a declaration of a library package or generic library package, causes <freezing> of each entity declared within it, except for incomplete types. A noninstance body other than a renames-as-body causes freezing of each entity declared before it within the same declarative_part.

4/1

A construct that (explicitly or implicitly) references an entity can cause the <freezing> of the entity, as defined by subsequent paragraphs. At the place where a construct causes freezing, each name, expression, implicit_dereference, or range within the construct causes freezing:

5

- The occurrence of a generic_instantiation causes freezing; also, if a parameter of the instantiation is defaulted, the default_expression or default_name for that parameter causes freezing.

6

- The occurrence of an object_declaration that has no corresponding completion causes freezing.

7

- The declaration of a record extension causes freezing of the parent subtype.

7.1/2

- The declaration of a record extension, interface type, task unit, or protected unit causes freezing of any progenitor types specified in the declaration.

8/1

A static expression causes freezing where it occurs. An object name or nonstatic expression causes freezing where it occurs, unless the name or expression is part of a default_expression, a default_name, or a per-object expression of a component's constraint, in which case, the freezing occurs later as part of another construct.

8.1/1

An implicit call freezes the same entities that would be frozen by an explicit call. This is true even if the implicit call is removed via implementation permissions.

8.2/1

If an expression is implicitly converted to a type or subtype $\langle T \rangle$, then at the place where the expression causes freezing, $\langle T \rangle$ is frozen.

9

The following rules define which entities are frozen at the place where a construct causes freezing:

10

- At the place where an expression causes freezing, the type of the expression is frozen, unless the expression is an enumeration literal used as a discrete_choice of the array_aggregate (see [S0111], page 196) of an enumeration_representation_clause (see [S0287], page 500).

11

- At the place where a name causes freezing, the entity denoted by the name is frozen, unless the name is a prefix of an expanded name; at the place where an object name causes freezing, the nominal subtype associated with the name is frozen.

11.1/1

- At the place where an implicit_dereference causes freezing, the nominal subtype associated with the implicit_dereference is frozen.

12

- At the place where a range causes freezing, the type of the range is frozen.

13

- At the place where an allocator causes freezing, the designated subtype of its type is frozen. If the type of the allocator is a derived type, then all ancestor types are also frozen.

14

- At the place where a callable entity is frozen, each subtype of its profile is frozen. If the callable entity is a member of an entry family, the index subtype of the family is frozen. At the place where a function call causes freezing, if a parameter of the call is defaulted, the default_expression (see [S0063], page 123) for that parameter causes freezing.

15

- At the place where a subtype is frozen, its type is frozen. At the place where a type is frozen, any expressions or names within the full type definition cause freezing; the

first subtype, and any component subtypes, index subtypes, and parent subtype of the type are frozen as well. For a specific tagged type, the corresponding class-wide type is frozen as well. For a class-wide type, the corresponding specific type is frozen as well.

15.1/2

- At the place where a specific tagged type is frozen, the primitive subprograms of the type are frozen.

Legality Rules

16

The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see Section 4.9.2 [3.9.2], page 145).

17

A type shall be completely defined before it is frozen (see Section 4.11.1 [3.11.1], page 177, and Section 8.3 [7.3], page 283).

18

The completion of a deferred constant declaration shall occur before the constant is frozen (see Section 8.4 [7.4], page 290).

19/1

An operational or representation item that directly specifies an aspect of an entity shall appear before the entity is frozen (see Section 14.1 [13.1], page 481).

Dynamic Semantics

20/2

The tag (see Section 4.9 [3.9], page 136) of a tagged type T is created at the point where T is frozen.

15 Annex A Predefined Language Environment

1

This Annex contains the specifications of library units that shall be provided by every implementation. There are three root library units: Ada, Interfaces, and System; other library units are children of these:

2/2

Standard -- Section 15.1 [A.1], page 556,
Ada -- Section 15.2 [A.2], page 565,
Assertions -- Section 12.4.2 [11.4.2], page 427,
Asynchronous_Task_Control -- Section 18.11 [D.11], page 1016,
Calendar -- Section 10.6 [9.6], page 358,
Arithmetic -- Section 10.6.1 [9.6.1], page 363,
Formatting -- Section 10.6.1 [9.6.1], page 363,
Time_Zones -- Section 10.6.1 [9.6.1], page 363,
Characters -- Section 15.3.1 [A.3.1], page 565,
Conversions -- Section 15.3.4 [A.3.4], page 580,
Handling -- Section 15.3.2 [A.3.2], page 566,
Latin_1 -- Section 15.3.3 [A.3.3], page 573,
Command_Line -- Section 15.15 [A.15], page 754,
Complex_Text_IO -- Section 21.1.3 [G.1.3], page 1097,
Containers -- Section 15.18.1 [A.18.1], page 779,
Doubly_Linked_Lists -- Section 15.18.3 [A.18.3], page 810,
Generic_Array_Sort -- Section 15.18.16 [A.18.16], page 891,
Generic_Constrained_Array_Sort
-- Section 15.18.16 [A.18.16], page 891,
Hashed_Maps -- Section 15.18.5 [A.18.5], page 839,
Hashed_Sets -- Section 15.18.8 [A.18.8], page 867,
Indefinite_Doubly_Linked_Lists
-- Section 15.18.11 [A.18.11], page 888,
Indefinite_Hashed_Maps -- Section 15.18.12 [A.18.12], page 889,
Indefinite_Hashed_Sets -- Section 15.18.14 [A.18.14], page 890,
Indefinite_Ordered_Maps -- Section 15.18.13 [A.18.13],
page 889,
Indefinite_Ordered_Sets -- Section 15.18.15 [A.18.15], page 891,
Indefinite_Vectors -- Section 15.18.10 [A.18.10], page 887,
Ordered_Maps -- Section 15.18.6 [A.18.6], page 846,
Ordered_Sets -- Section 15.18.9 [A.18.9], page 876,
Vectors -- Section 15.18.2 [A.18.2], page 779,
Decimal -- Section 20.2 [F.2], page 1055,
Direct_IO -- Section 15.8.4 [A.8.4], page 691,
Directories -- Section 15.16 [A.16], page 757,
Information -- Section 15.16 [A.16], page 757,
Dispatching -- Section 18.2.1 [D.2.1], page 978,

EDF -- Section 18.2.6 [D.2.6], page 987,
Round_Robin -- Section 18.2.5 [D.2.5], page 985,
Dynamic_Priorities -- Section 18.5 [D.5], page 996,

Standard (<...continued>)

Ada (<...continued>)

Environment_Variables -- Section 15.17 [A.17], page 775,
Exceptions -- Section 12.4.1 [11.4.1], page 423,
Execution_Time -- Section 18.14 [D.14], page 1021,
Group_Budgets -- Section 18.14.2 [D.14.2], page 1027,
Timers -- Section 18.14.1 [D.14.1], page 1024,
Finalization -- Section 8.6 [7.6], page 295,
Float_Text_IO -- Section 15.10.9 [A.10.9], page 731,
Float_Wide_Text_IO -- Section 15.11 [A.11], page 745,
Float_Wide_Wide_Text_IO -- Section 15.11 [A.11], page 745,
Integer_Text_IO -- Section 15.10.8 [A.10.8], page 727,
Integer_Wide_Text_IO -- Section 15.11 [A.11], page 745,
Integer_Wide_Wide_Text_IO -- Section 15.11 [A.11], page 745,
Interrupts -- Section 17.3.2 [C.3.2], page 957,
Names -- Section 17.3.2 [C.3.2], page 957,
IO_Exceptions -- Section 15.13 [A.13], page 752,
Numerics -- Section 15.5 [A.5], page 648,
Complex_Arrays -- Section 21.3.2 [G.3.2], page 1130,
Complex_Elementary_Functions -- Section 21.1.2 [G.1.2],
page 1091,
Complex_Types -- Section 21.1.1 [G.1.1], page 1084,
Discrete_Random -- Section 15.5.2 [A.5.2], page 654,
Elementary_Functions -- Section 15.5.1 [A.5.1], page 648,
Float_Random -- Section 15.5.2 [A.5.2], page 654,
Generic_Complex_Arrays -- Section 21.3.2 [G.3.2], page 1130,
Generic_Complex_Elementary_Functions
-- Section 21.1.2 [G.1.2], page 1091,
Generic_Complex_Types -- Section 21.1.1 [G.1.1], page 1084,
Generic_Elementary_Functions -- Section 15.5.1 [A.5.1],
page 648,
Generic_Real_Arrays -- Section 21.3.1 [G.3.1], page 1119,
Real_Arrays -- Section 21.3.1 [G.3.1], page 1119,
Real_Time -- Section 18.8 [D.8], page 1008,
Timing_Events -- Section 18.15 [D.15], page 1031,
Sequential_IO -- Section 15.8.1 [A.8.1], page 683,
Storage_IO -- Section 15.9 [A.9], page 695,
Streams -- Section 14.13.1 [13.13.1], page 538,
Stream_IO -- Section 15.12.1 [A.12.1], page 746,

Standard (<...continued>)

Ada (<...continued>)

Strings -- Section 15.4.1 [A.4.1], page 584,
Bounded -- Section 15.4.4 [A.4.4], page 610,

- Hash -- Section 15.4.9 [A.4.9], page 646,
- Fixed -- Section 15.4.3 [A.4.3], page 591,
- Hash -- Section 15.4.9 [A.4.9], page 646,
- Hash -- Section 15.4.9 [A.4.9], page 646,
- Maps -- Section 15.4.2 [A.4.2], page 585,
- Constants -- Section 15.4.6 [A.4.6], page 635,
- Unbounded -- Section 15.4.5 [A.4.5], page 625,
- Hash -- Section 15.4.9 [A.4.9], page 646,
- Wide_Bounded -- Section 15.4.7 [A.4.7], page 636,
- Wide_Hash -- Section 15.4.7 [A.4.7], page 636,
- Wide_Fixed -- Section 15.4.7 [A.4.7], page 636,
- Wide_Hash -- Section 15.4.7 [A.4.7], page 636,
- Wide_Hash -- Section 15.4.7 [A.4.7], page 636,
- Wide_Hash -- Section 15.4.7 [A.4.7], page 636,
- Wide_Maps -- Section 15.4.7 [A.4.7], page 636,
- Wide_Constants -- Section 15.4.7 [A.4.7], page 636,
- Wide_Unbounded -- Section 15.4.7 [A.4.7], page 636,
- Wide_Hash -- Section 15.4.7 [A.4.7], page 636,
- Wide_Wide_Bounded -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Hash -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Fixed -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Hash -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Hash -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Hash -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Maps -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Constants -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Unbounded -- Section 15.4.8 [A.4.8], page 641,
- Wide_Wide_Hash -- Section 15.4.8 [A.4.8], page 641,
- Synchronous_Task_Control -- Section 18.10 [D.10], page 1015,
- Tags -- Section 4.9 [3.9], page 136,
- Generic_Dispatching_Constructor -- Section 4.9 [3.9], page 136,
- Task_Attributes -- Section 17.7.2 [C.7.2], page 967,
- Task_Identification -- Section 17.7.1 [C.7.1], page 965,
- Task_Termination -- Section 17.7.3 [C.7.3], page 971,

Standard (<...continued>)

Ada (<...continued>)

- Text_IO -- Section 15.10.1 [A.10.1], page 698,
- Bounded_IO -- Section 15.10.11 [A.10.11], page 739,
- Complex_IO -- Section 21.1.3 [G.1.3], page 1097,
- Editing -- Section 20.3.3 [F.3.3], page 1073,
- Text_Streams -- Section 15.12.2 [A.12.2], page 750,
- Unbounded_IO -- Section 15.10.12 [A.10.12], page 742,
- Unchecked_Conversion -- Section 14.9 [13.9], page 520,
- Unchecked_Deallocation -- Section 14.11.2 [13.11.2], page 532,
- Wide_Characters -- Section 15.3.1 [A.3.1], page 565,
- Wide_Text_IO -- Section 15.11 [A.11], page 745,
- Complex_IO -- Section 21.1.4 [G.1.4], page 1103,
- Editing -- Section 20.3.4 [F.3.4], page 1081,

Text_Streams -- Section 15.12.3 [A.12.3], page 751,
 Wide_Bounded_IO -- Section 15.11 [A.11], page 745,
 Wide_Unbounded_IO -- Section 15.11 [A.11], page 745,
 Wide_Wide_Characters -- Section 15.3.1 [A.3.1], page 565,
 Wide_Wide_Text_IO -- Section 15.11 [A.11], page 745,
 Complex_IO -- Section 21.1.5 [G.1.5], page 1103,
 Editing -- Section 20.3.5 [F.3.5], page 1081,
 Text_Streams -- Section 15.12.4 [A.12.4], page 752,
 Wide_Wide_Bounded_IO -- Section 15.11 [A.11], page 745,
 Wide_Wide_Unbounded_IO -- Section 15.11 [A.11], page 745,

 Interfaces -- Section 16.2 [B.2], page 900,
 C -- Section 16.3 [B.3], page 901,
 Pointers -- Section 16.3.2 [B.3.2], page 922,
 Strings -- Section 16.3.1 [B.3.1], page 915,
 COBOL -- Section 16.4 [B.4], page 931,
 Fortran -- Section 16.5 [B.5], page 945,

 System -- Section 14.7 [13.7], page 510,
 Address_To_Access_Conversions -- Section 14.7.2 [13.7.2],
 page 517,
 Machine_Code -- Section 14.8 [13.8], page 518,
 RPC -- Section 19.5 [E.5], page 1050,
 Storage_Elements -- Section 14.7.1 [13.7.1], page 516,
 Storage_Pools -- Section 14.11 [13.11], page 526,
 Implementation Requirements

3/2

The implementation shall ensure that each language-defined subprogram is reentrant in the sense that concurrent calls on the same subprogram perform as specified, so long as all parameters that could be passed by reference denote nonoverlapping objects.

Implementation Permissions

4

The implementation may restrict the replacement of language-defined compilation units. The implementation may restrict children of language-defined library units (other than Standard).

15.1 A.1 The Package Standard

1

This clause outlines the specification of the package Standard containing all predefined identifiers in the language. The corresponding package body is not specified by the language.

2

The operators that are predefined for the types declared in the package Standard are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as <root_real>) and for undefined information (such as <implementation-defined>).

Static Semantics

3

The library package Standard has the following declaration:

4

```
package Standard is
  pragma Pure(Standard);
```

5

```
type Boolean is (False, True);
```

6

```
--< The predefined relational operators for this type are as follows:>■
```

7/1

```
-- function "=" (Left, Right : Boolean'Base) return Boolean;
-- function "/=" (Left, Right : Boolean'Base) return Boolean;
-- function "<" (Left, Right : Boolean'Base) return Boolean;
-- function "<=" (Left, Right : Boolean'Base) return Boolean;
-- function ">" (Left, Right : Boolean'Base) return Boolean;
-- function ">=" (Left, Right : Boolean'Base) return Boolean;
```

8

```
--< The predefined logical operators and the predefined logical>
--< negation operator are as follows:>
```

9/1

```
-- function "and" (Left, Right : Boolean'Base) return Boolean'Base;■
-- function "or" (Left, Right : Boolean'Base) return Boolean'Base;■
-- function "xor" (Left, Right : Boolean'Base) return Boolean'Base;■
```

10/1

```
-- function "not" (Right : Boolean'Base) return Boolean'Base;
```

11/2

```
--< The integer type root_integer and the>
--< corresponding universal type universal_integer are predefined.>■
```

12

```
type Integer is range <implementation-defined>;
```

13

```
subtype Natural is Integer range 0 .. Integer'Last;
```

```

14      subtype Positive is Integer range 1 .. Integer'Last;

--< The predefined operators for type Integer are as follows:>
15

-- function "=" (Left, Right : Integer'Base) return Boolean;
-- function "/=" (Left, Right : Integer'Base) return Boolean;
-- function "<" (Left, Right : Integer'Base) return Boolean;
-- function "<=" (Left, Right : Integer'Base) return Boolean;
-- function ">" (Left, Right : Integer'Base) return Boolean;
-- function ">=" (Left, Right : Integer'Base) return Boolean;
16

-- function "+" (Right : Integer'Base) return Integer'Base;
-- function "-" (Right : Integer'Base) return Integer'Base;
-- function "abs" (Right : Integer'Base) return Integer'Base;
17

-- function "+" (Left, Right : Integer'Base) return Integer'Base;█
-- function "-" (Left, Right : Integer'Base) return Integer'Base;█
-- function "*" (Left, Right : Integer'Base) return Integer'Base;█
-- function "/" (Left, Right : Integer'Base) return Integer'Base;█
-- function "rem" (Left, Right : Integer'Base) return Integer'Base;█
-- function "mod" (Left, Right : Integer'Base) return Integer'Base;█
18

-- function "**" (Left : Integer'Base; Right : Natural)
--      return Integer'Base;
19

--< The specification of each operator for the type>
--< root_integer, or for any additional predefined integer>
--< type, is obtained by replacing Integer by the name of the type>█
--< in the specification of the corresponding operator of the type>█
--< Integer. The right operand of the exponentiation operator>
--< remains as subtype Natural.>
20/2

--< The floating point type root_real and the>
--< corresponding universal type universal_real are predefined.>
21

type Float is digits <implementation-defined>;

```

22

```
--< The predefined operators for this type are as follows:>
```

23

```
-- function "=" (Left, Right : Float) return Boolean;
-- function "/=" (Left, Right : Float) return Boolean;
-- function "<" (Left, Right : Float) return Boolean;
-- function "<=" (Left, Right : Float) return Boolean;
-- function ">" (Left, Right : Float) return Boolean;
-- function ">=" (Left, Right : Float) return Boolean;
```

24

```
-- function "+" (Right : Float) return Float;
-- function "-" (Right : Float) return Float;
-- function "abs" (Right : Float) return Float;
```

25

```
-- function "+" (Left, Right : Float) return Float;
-- function "-" (Left, Right : Float) return Float;
-- function "*" (Left, Right : Float) return Float;
-- function "/" (Left, Right : Float) return Float;
```

26

```
-- function "**" (Left : Float; Right : Integer'Base) return Float;■
```

27

```
--< The specification of each operator for the type root_real, or for■
--< any additional predefined floating point type, is obtained by>■
--< replacing Float by the name of the type in the specification of the>■
--< corresponding operator of the type Float.>
```

28

```
--< In addition, the following operators are predefined for the root>■
--< numeric types:>
```

29

```
function "*" (Left : <root_integer>; Right : <root_real>)
  return <root_real>;
```

30

```
function "*" (Left : <root_real>; Right : <root_integer>)
  return <root_real>;
```

31

```
function "/" (Left : <root_real>; Right : <root_integer>)
  return <root_real>;
```

32

```
--< The type universal_fixed is predefined.>
--< The only multiplying operators defined between>
--< fixed point types are>
```

33

```
function "*" (Left : <universal_fixed>; Right : <universal_fixed>)
  return <universal_fixed>;
```

34

```
function "/" (Left : <universal_fixed>; Right : <universal_fixed>)
  return <universal_fixed>;
```

34.1/2

```
--< The type universal_access is predefined.>
--< The following equality operators are predefined:>
```

34.2/2

```
function "=" (Left, Right: <universal_access>) return Boolean;
function "/=" (Left, Right: <universal_access>) return Boolean;
```

35/2

```
--< The declaration of type Character is based on the standard ISO 8859-1
```

```
--< There are no character literals corresponding to the positions for cont
```

```
--< They are indicated in italics in this definition. See Section 4.5.2
[3.5.2], page 93.>
```

```
type Character is
```

```
(<nul>, <soh>, <stx>, <etx>, <eot>, <enq>, <ack>, <bel>, --<
<bs>, <ht>, <lf>, <vt>, <ff>, <cr>, <so>, <si>, --<8 (16#08
```

```
<dle>, <dc1>, <dc2>, <dc3>, <dc4>, <nak>, <syn>, <etb>, --<
<can>, <em>, <sub>, <esc>, <fs>, <gs>, <rs>, <us>, --<24 (1
```

```
' ', '!', '""', '#', '$', '%', '&', '''', --<32 (16#20#) .. 3
```

```
'(', ')', '*', '+', ',', '-', '.', '/', --<40 (16#28#) .. 4
```

```
'0', '1', '2', '3', '4', '5', '6', '7', --<48 (16#30#) .. 5
```



```

'8', '9', ':', ';', '<', '=', '>', '?', --<56 (16#38#) .. 63
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G', --<64 (16#40#) .. 71
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', --<72 (16#48#) .. 79
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', --<80 (16#50#) .. 87
'X', 'Y', 'Z', '[', '\', ']', '^', '_', --<88 (16#58#) .. 95
'', 'a', 'b', 'c', 'd', 'e', 'f', 'g', --<96 (16#60#) .. 103
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', --<104 (16#68#) .. 111
'p', 'q', 'r', 's', 't', 'u', 'v', 'w', --<112 (16#70#) .. 119
'x', 'y', 'z', '{', '|', '}', '~', <del>, --<120 (16#78#) .. 127

<reserved_128>, <reserved_129>, <bph>, <nbh>, --<128 (16#80#) .. 135
<reserved_132>, <nel>, <ssa>, <esa>, --<132 (16#84#) .. 139
<hts>, <htj>, <vts>, <pld>, <plu>, <ri>, <ss2>, <ss3>, --<140 (16#88#) .. 147

<dcs>, <pu1>, <pu2>, <sts>, <cch>, <mw>, <spa>, <epa>, --<148 (16#94#) .. 155
<sos>, <reserved_153>, <sci>, <csi>, --<152 (16#98#) .. 159
<st>, <osc>, <pm>, <apc>, --<156 (16#9C#) .. 159 (16#9F#)

' ', 'i', 'ϕ', 'ℓ', 'ϳ', 'Ϸ', '∣', '§', --<160 (16#A0#) .. 167
'...', '©', 'ª', '«', '¬', '„', '®', '¬', --<168 (16#A8#) .. 175

'°', '±', '²', '³', '´', 'μ', '¶', '·', --<176 (16#B0#) .. 183
'¸', '¹', 'º', '»', '¼', '½', '¾', '¿', --<184 (16#B8#) .. 191

'À', 'Á', 'Â', 'Ã', 'Ä', 'Å', 'Æ', 'Ç', --<192 (16#C0#) .. 199
'È', 'É', 'Ê', 'Ë', 'Ì', 'Í', 'Î', 'Ï', --<200 (16#C8#) .. 207

'Ð', 'Ñ', 'Ò', 'Ó', 'Ô', 'Õ', 'Ö', '×', --<208 (16#D0#) .. 215
'Ø', 'Ù', 'Ú', 'Û', 'Ü', 'Ý', 'Þ', 'ß', --<216 (16#D8#) .. 223

'à', 'á', 'â', 'ã', 'ä', 'å', 'æ', 'ç', --<224 (16#E0#) .. 231
'è', 'é', 'ê', 'ë', 'ì', 'í', 'î', 'ï', --<232 (16#E8#) .. 239

'ð', 'ñ', 'ò', 'ó', 'ô', 'õ', 'ö', '÷', --<240 (16#F0#) .. 247
'ø', 'ù', 'ú', 'û', 'ü', 'ý', 'þ', 'ÿ');--<248 (16#F8#) .. 255

```

36

```

--< The predefined operators for the type Character are the same as for>■
--< any enumeration type.>

```

36.1/2

--< The declaration of type Wide_Character is based on the standard ISO/IEC 10646:2003 character set. The first 256 positions have the same contents as type Character. See Section 3.5.2 [3.5.2], page 93.>

36.2/2

```
type Wide_Character is (<nul>, <soh> ... <Hex_0000FFFE>, <Hex_0000FFFF>);
```

--< The declaration of type Wide_Wide_Character is based on the full ISO/IEC 10646:2003 character set. The first 65536 positions have the same contents as type Wide_Character. See Section 4.5.2 [3.5.2], page 93.>

36.3/2

```
type Wide_Wide_Character is (<nul>, <soh> ... <Hex_7FFFFFFE>, <Hex_7FFFFFFF>);
for Wide_Wide_Character'Size use 32;
```

package ASCII is ... end ASCII; --<Obsolescent; see Section 23.5 [J.5], page 1169>

37

--< Predefined string types:>

```
type String is array(Positive range <>) of Character;
pragma Pack(String);
```

38

--< The predefined operators for this type are as follows:>

39

```
-- function "=" (Left, Right: String) return Boolean;
-- function "/=" (Left, Right: String) return Boolean;
-- function "<" (Left, Right: String) return Boolean;
-- function "<=" (Left, Right: String) return Boolean;
-- function ">" (Left, Right: String) return Boolean;
-- function ">=" (Left, Right: String) return Boolean;
```

40

```
-- function "&" (Left: String; Right: String) return String;
-- function "&" (Left: Character; Right: String) return String;
-- function "&" (Left: String; Right: Character) return String;
-- function "&" (Left: Character; Right: Character) return String;
```

41

```
type Wide_String is array(Positive range <>) of Wide_Character;  
pragma Pack(Wide_String);
```

42

```
--< The predefined operators for this type correspond to those for String.>■
```

42.1/2

```
type Wide_Wide_String is array (Positive range <>)  
  of Wide_Wide_Character;  
pragma Pack (Wide_Wide_String);
```

42.2/2

```
--< The predefined operators for this type correspond to those for String.>■
```

43

```
type Duration is delta <implementation-defined> range <implementation-defined>
```

44

```
--< The predefined operators for the type Duration are the same as for>■  
--< any fixed point type.>
```

45

```
--< The predefined exceptions:>
```

46

```
Constraint_Error: exception;  
Program_Error   : exception;  
Storage_Error   : exception;  
Tasking_Error   : exception;
```

47

```
end Standard;
```

48

Standard has no private part.

49/2

In each of the types Character, Wide_Character, and Wide_Wide_Character, the character literals for the space character (position 32) and the non-breaking space character (position 160) correspond to different values. Unless indicated otherwise, each occurrence of the character literal ' ' in this International Standard refers to the space character. Similarly, the character literals for hyphen (position 45) and soft hyphen (position 173) correspond to

different values. Unless indicated otherwise, each occurrence of the character literal '–' in this International Standard refers to the hyphen character.

Dynamic Semantics

50

Elaboration of the body of Standard has no effect.

Implementation Permissions

51

An implementation may provide additional predefined integer types and additional predefined floating point types. Not all of these types need have names.

Implementation Advice

52

If an implementation provides additional named predefined integer types, then the names should end with "Integer" as in "Long_Integer". If an implementation provides additional named predefined floating point types, then the names should end with "Float" as in "Long_Float".

NOTES

53

1 Certain aspects of the predefined entities cannot be completely described in the language itself. For example, although the enumeration type Boolean can be written showing the two enumeration literals False and True, the short-circuit control forms cannot be expressed in the language.

54

2 As explained in Section 9.1 [8.1], page 304, "Section 9.1 [8.1], page 304, Declarative Region" and Section 11.1.4 [10.1.4], page 406, "Section 11.1.4 [10.1.4], page 406, The Compilation Process", the declarative region of the package Standard encloses every library unit and consequently the main subprogram; the declaration of every library unit is assumed to occur within this declarative region. Library_items are assumed to be ordered in such a way that there are no forward semantic dependences. However, as explained in Section 9.3 [8.3], page 308, "Section 9.3 [8.3], page 308, Visibility", the only library units that are visible within a given compilation unit are the library units named by all with_clauses that apply to the given unit, and moreover, within the declarative region of a given library unit, that library unit itself.

55

3 If all block_statements of a program are named, then the name of each program unit can always be written as an expanded name starting with Standard (unless Standard is itself hidden). The name of a library unit cannot be a homograph of a name (such as Integer) that is already declared in Standard.

4 The exception `Standard.Numeric_Error` is defined in Section 23.6 [J.6], page 1170.

15.2 A.2 The Package Ada

Static Semantics

1
The following language-defined library package exists:

2

```
package Ada is
    pragma Pure(Ada);
end Ada;
```

3
Ada serves as the parent of most of the other language-defined library units; its declaration is empty (except for the pragma `Pure`).

Legality Rules

4
In the standard mode, it is illegal to compile a child of package Ada.

15.3 A.3 Character Handling

1/2

This clause presents the packages related to character processing: an empty pure package `Characters` and child packages `Characters.Handling` and `Characters.Latin_1`. The package `Characters.Handling` provides classification and conversion functions for `Character` data, and some simple functions for dealing with `Wide_Character` and `Wide_Wide_Character` data. The child package `Characters.Latin_1` declares a set of constants initialized to values of type `Character`.

15.3.1 A.3.1 The Packages `Characters`, `Wide_Characters`, and `Wide_Wide_Characters`

Static Semantics

1
The library package `Characters` has the following declaration:

2

```
package Ada.Characters is
    pragma Pure(Characters);
end Ada.Characters;
```

3/2

The library package `Wide_Characters` has the following declaration:

4/2

```
package Ada.Wide_Characters is
  pragma Pure(Wide_Characters);
end Ada.Wide_Characters;
```

5/2

The library package Wide_Wide_Characters has the following declaration:

6/2

```
package Ada.Wide_Wide_Characters is
  pragma Pure(Wide_Wide_Characters);
end Ada.Wide_Wide_Characters;
```

Implementation Advice

7/2

If an implementation chooses to provide implementation-defined operations on Wide_Character or Wide_String (such as case mapping, classification, collating and sorting, etc.) it should do so by providing child units of Wide_Characters. Similarly if it chooses to provide implementation-defined operations on Wide_Wide_Character or Wide_Wide_String it should do so by providing child units of Wide_Wide_Characters.

15.3.2 A.3.2 The Package Characters.Handling

Static Semantics

1

The library package Characters.Handling has the following declaration:

2/2

```
with Ada.Characters.Conversions;
package Ada.Characters.Handling is
  pragma Pure(Handling);
```

3

```
--<Character classification functions>
```

4

```
function Is_Control          (Item : in Character) return Boolean;
function Is_Graphic         (Item : in Character) return Boolean;
function Is_Letter          (Item : in Character) return Boolean;
function Is_Lower           (Item : in Character) return Boolean;
function Is_Upper           (Item : in Character) return Boolean;
function Is_Basic           (Item : in Character) return Boolean;
function Is_Digit           (Item : in Character) return Boolean;
function Is_Decimal_Digit   (Item : in Character) return Boolean;
                           renames Is_Digit;
function Is_Hexadecimal_Digit (Item : in Character) return Boolean;
```

5 function Is_Alphanumeric (Item : in Character) return Boolean;█
 function Is_Special (Item : in Character) return Boolean;█

5

--<Conversion functions for Character and String>

6

 function To_Lower (Item : in Character) return Character;
 function To_Upper (Item : in Character) return Character;
 function To_Basic (Item : in Character) return Character;

7

 function To_Lower (Item : in String) return String;
 function To_Upper (Item : in String) return String;
 function To_Basic (Item : in String) return String;

8

--<Classifications of and conversions between Character and ISO 646>█

9

 subtype ISO_646 is
 Character range Character'Val(0) .. Character'Val(127);

10

 function Is_ISO_646 (Item : in Character) return Boolean;
 function Is_ISO_646 (Item : in String) return Boolean;

11

 function To_ISO_646 (Item : in Character;
 Substitute : in ISO_646 := ' ')
 return ISO_646;

12

 function To_ISO_646 (Item : in String;
 Substitute : in ISO_646 := ' ')
 return String;

13/2

--< The functions Is_Character, Is_String, To_Character, To_String, To_Wide_Character
--< and To_Wide_String are obsolescent; see Section 23.14 [J.14],
page 1177.>

<Paragraphs 14 through 18 were deleted.>

19

end Ada.Characters.Handling;

20

In the description below for each function that returns a Boolean result, the effect is described in terms of the conditions under which the value True is returned. If these conditions are not met, then the function returns False.

21

Each of the following classification functions has a formal Character parameter, Item, and returns a Boolean result.

22

Is_Control

True if Item is a control character. A <control character> is a character whose position is in one of the ranges 0..31 or 127..159.

23

Is_Graphic

True if Item is a graphic character. A <graphic character> is a character whose position is in one of the ranges 32..126 or 160..255.

24

Is_Letter

True if Item is a letter. A <letter> is a character that is in one of the ranges 'A'..'Z' or 'a'..'z', or whose position is in one of the ranges 192..214, 216..246, or 248..255.

25

Is_Lower

True if Item is a lower-case letter. A <lower-case letter>

is a character that is in the range 'a'..'z', or whose position is in one of the ranges 223..246 or 248..255.

26

Is_Upper

True if Item is an upper-case letter. An <upper-case letter> is a character that is in the range 'A'..'Z' or whose position is in one of the ranges 192..214 or 216.. 222.

27

Is_Basic

True if Item is a basic letter. A <basic letter> is a character that is in one of the ranges 'A'..'Z' and 'a'..'z', or that is one of the following: 'Æ', 'æ', 'Ð', 'ð', 'P', 'p', or 'B'.

28

Is_Digit

True if Item is a decimal digit. A <decimal digit> is a character in the range '0'..'9'.

29

Is_Decimal_Digit

A renaming of Is_Digit.

30

Is_Hexadecimal_Digit

True if Item is a hexadecimal digit. A <hexadecimal digit> is a character that is either a decimal digit or that is in one of

the ranges 'A' .. 'F'
or 'a' .. 'f'.

31

Is_Alphanumeric

True if Item is an alphanumeric character. An <alphanumeric character> is a character that is either a letter or a decimal digit.

32

Is_Special

True if Item is a special graphic character. A <special graphic character> is a graphic character that is not alphanumeric.

33

Each of the names To_Lower, To_Upper, and To_Basic refers to two functions: one that converts from Character to Character, and the other that converts from String to String. The result of each Character-to-Character function is described below, in terms of the conversion applied to Item, its formal Character parameter. The result of each String-to-String conversion is obtained by applying to each element of the function's String parameter the corresponding Character-to-Character conversion; the result is the null String if the value of the formal parameter is the null String. The lower bound of the result String is 1.

34

To_Lower

Returns the corresponding lower-case value for Item if Is_Upper(Item), and returns Item otherwise.

35

To_Upper

Returns the corresponding upper-case value for Item if Is_Lower(Item) and Item has an upper-case form,

and returns Item otherwise. The lower case letters 'ß' and 'ÿ' do not have upper case forms.

36

To_Basic

Returns the letter corresponding to Item but with no diacritical mark, if Item is a letter but not a basic letter; returns Item otherwise.

37

The following set of functions test for membership in the ISO 646 character range, or convert between ISO 646 and Character.

38

Is_ISO_646

The function whose formal parameter, Item, is of type Character returns True if Item is in the subtype ISO_646.

39

Is_ISO_646

The function whose formal parameter, Item, is of type String returns True if Is_ISO_646(Item(I)) is True for each I in Item'Range.

40

To_ISO_646

The function whose first formal parameter, Item, is of type Character returns Item if Is_ISO_646(Item), and returns the

Substitute ISO_646
character otherwise.

41
To_ISO_646

The function
whose first formal
parameter, Item,
is of type String
returns the String
whose Range is
1..Item'Length
and each of whose
elements is given
by To_ISO_646 of
the corresponding
element in Item.

<Paragraphs 42 through 48 were deleted.>

Implementation Advice

49/2

<This paragraph was deleted.>

NOTES

50

5 A basic letter is a letter without a diacritical mark.

51

6 Except for the hexadecimal digits, basic letters, and ISO_646 characters, the categories identified in the classification functions form a strict hierarchy:

52

-- Control characters

53

-- Graphic characters

54

-- Alphanumeric characters

55

-- Letters

56

57 -- Upper-case letters

58 -- Lower-case letters

59 -- Decimal digits

 -- Special graphic characters

15.3.3 A.3.3 The Package Characters.Latin_1

1
The package Characters.Latin_1 declares constants for characters in ISO 8859-1.
Static Semantics

2
The library package Characters.Latin_1 has the following declaration:

3

```
package Ada.Characters.Latin_1 is
  pragma Pure(Latin_1);
4
```

5

```

NUL          : constant Character := Character'Val(0);
SOH          : constant Character := Character'Val(1);
STX          : constant Character := Character'Val(2);
ETX          : constant Character := Character'Val(3);
EOT          : constant Character := Character'Val(4);
ENQ          : constant Character := Character'Val(5);
ACK          : constant Character := Character'Val(6);
BEL          : constant Character := Character'Val(7);
BS           : constant Character := Character'Val(8);
HT           : constant Character := Character'Val(9);
LF           : constant Character := Character'Val(10);
VT           : constant Character := Character'Val(11);
FF           : constant Character := Character'Val(12);
CR           : constant Character := Character'Val(13);
SO           : constant Character := Character'Val(14);
SI           : constant Character := Character'Val(15);
--< Control characters:>
```

6

```
DLE           : constant Character := Character'Val(16);
DC1           : constant Character := Character'Val(17);
DC2           : constant Character := Character'Val(18);
DC3           : constant Character := Character'Val(19);
DC4           : constant Character := Character'Val(20);
NAK           : constant Character := Character'Val(21);
SYN           : constant Character := Character'Val(22);
ETB           : constant Character := Character'Val(23);
CAN           : constant Character := Character'Val(24);
EM            : constant Character := Character'Val(25);
SUB           : constant Character := Character'Val(26);
ESC           : constant Character := Character'Val(27);
FS            : constant Character := Character'Val(28);
GS            : constant Character := Character'Val(29);
RS            : constant Character := Character'Val(30);
US            : constant Character := Character'Val(31);
```

7

--< ISO 646 graphic characters:>

8

```
Space         : constant Character := ' '; --< Character'Val(32)>■
Exclamation   : constant Character := '!'; --< Character'Val(33)>■
Quotation     : constant Character := '"'; --< Character'Val(34)>■
Number_Sign   : constant Character := '#'; --< Character'Val(35)>■
Dollar_Sign   : constant Character := '$'; --< Character'Val(36)>■
Percent_Sign  : constant Character := '%'; --< Character'Val(37)>■
Ampersand     : constant Character := '&'; --< Character'Val(38)>■
Apostrophe    : constant Character := '''; --< Character'Val(39)>■
Left_Parenthesis : constant Character := '('; --< Character'Val(40)>■
Right_Parenthesis : constant Character := ')'; --< Character'Val(41)>■
Asterisk      : constant Character := '*'; --< Character'Val(42)>■
Plus_Sign     : constant Character := '+'; --< Character'Val(43)>■
Comma        : constant Character := ','; --< Character'Val(44)>■
Hyphen       : constant Character := '-'; --< Character'Val(45)>■
Minus_Sign   : Character renames Hyphen;
Full_Stop    : constant Character := '.'; --< Character'Val(46)>■
Solidus      : constant Character := '/'; --< Character'Val(47)>■
```

9

--< Decimal digits '0' though '9' are at positions 48 through 57>■

10

```

Colon          : constant Character := ':';  --< Character'Val(58)>█
Semicolon     : constant Character := ';';  --< Character'Val(59)>█
Less_Than_Sign : constant Character := '<';  --< Character'Val(60)>█
Equals_Sign    : constant Character := '=';  --< Character'Val(61)>█
Greater_Than_Sign : constant Character := '>';  --< Character'Val(62)>█
Question      : constant Character := '?';  --< Character'Val(63)>█
Commercial_At : constant Character := '@';  --< Character'Val(64)>█

```

11

```
--< Letters 'A' through 'Z' are at positions 65 through 90>
```

12

```

Left_Square_Bracket : constant Character := '[';  --< Character'Val(91)>█
Reverse_Solidus     : constant Character := '\';  --< Character'Val(92)>█
Right_Square_Bracket : constant Character := ']';  --< Character'Val(93)>█
Circumflex         : constant Character := '^';  --< Character'Val(94)>█
Low_Line           : constant Character := '_';  --< Character'Val(95)>█

```

13

```

Grave          : constant Character := '`';  --< Character'Val(96)>█
LC_A           : constant Character := 'a';  --< Character'Val(97)>█
LC_B           : constant Character := 'b';  --< Character'Val(98)>█
LC_C           : constant Character := 'c';  --< Character'Val(99)>█
LC_D           : constant Character := 'd';  --< Character'Val(100)>█
LC_E           : constant Character := 'e';  --< Character'Val(101)>█
LC_F           : constant Character := 'f';  --< Character'Val(102)>█
LC_G           : constant Character := 'g';  --< Character'Val(103)>█
LC_H           : constant Character := 'h';  --< Character'Val(104)>█
LC_I           : constant Character := 'i';  --< Character'Val(105)>█
LC_J           : constant Character := 'j';  --< Character'Val(106)>█
LC_K           : constant Character := 'k';  --< Character'Val(107)>█
LC_L           : constant Character := 'l';  --< Character'Val(108)>█
LC_M           : constant Character := 'm';  --< Character'Val(109)>█
LC_N           : constant Character := 'n';  --< Character'Val(110)>█
LC_O           : constant Character := 'o';  --< Character'Val(111)>█

```

14

```

LC_P           : constant Character := 'p';  --< Character'Val(112)>█
LC_Q           : constant Character := 'q';  --< Character'Val(113)>█
LC_R           : constant Character := 'r';  --< Character'Val(114)>█
LC_S           : constant Character := 's';  --< Character'Val(115)>█
LC_T           : constant Character := 't';  --< Character'Val(116)>█
LC_U           : constant Character := 'u';  --< Character'Val(117)>█
LC_V           : constant Character := 'v';  --< Character'Val(118)>█
LC_W           : constant Character := 'w';  --< Character'Val(119)>█

```

```

LC_X      : constant Character := 'x';  --< Character'Val(120)>█
LC_Y      : constant Character := 'y';  --< Character'Val(121)>█
LC_Z      : constant Character := 'z';  --< Character'Val(122)>█
Left_Curly_Bracket : constant Character := '{';  --< Character'Val(123)>█
Vertical_Line : constant Character := '|';  --< Character'Val(124)>█
Right_Curly_Bracket : constant Character := '}';  --< Character'Val(125)>█
Tilde     : constant Character := '~';  --< Character'Val(126)>█
DEL       : constant Character := Character'Val(127);█

```

15

--< ISO 6429 control characters:>

16

```

IS4       : Character renames FS;
IS3       : Character renames GS;
IS2       : Character renames RS;
IS1       : Character renames US;

```

17

```

Reserved_128 : constant Character := Character'Val(128);█
Reserved_129 : constant Character := Character'Val(129);█
BPH         : constant Character := Character'Val(130);█
NBH         : constant Character := Character'Val(131);█
Reserved_132 : constant Character := Character'Val(132);█
NEL         : constant Character := Character'Val(133);█
SSA         : constant Character := Character'Val(134);█
ESA         : constant Character := Character'Val(135);█
HTS         : constant Character := Character'Val(136);█
HTJ         : constant Character := Character'Val(137);█
VTS         : constant Character := Character'Val(138);█
PLD         : constant Character := Character'Val(139);█
PLU         : constant Character := Character'Val(140);█
RI          : constant Character := Character'Val(141);█
SS2         : constant Character := Character'Val(142);█
SS3         : constant Character := Character'Val(143);█

```

18

```

DCS       : constant Character := Character'Val(144);█
PU1       : constant Character := Character'Val(145);█
PU2       : constant Character := Character'Val(146);█
STS       : constant Character := Character'Val(147);█
CCH       : constant Character := Character'Val(148);█
MW        : constant Character := Character'Val(149);█
SPA       : constant Character := Character'Val(150);█
EPA       : constant Character := Character'Val(151);█

```


19

```
SOS                : constant Character := Character'Val(152);█
Reserved_153      : constant Character := Character'Val(153);█
SCI                : constant Character := Character'Val(154);█
CSI                : constant Character := Character'Val(155);█
ST                 : constant Character := Character'Val(156);█
OSC                : constant Character := Character'Val(157);█
PM                 : constant Character := Character'Val(158);█
APC                : constant Character := Character'Val(159);█
```

20

```
--< Other graphic characters:>
```

21

```
--< Character positions 160 (16#A0#) .. 175 (16#AF#):>
```

```
No_Break_Space    : constant Character := ' '; --<Character'Val(160)
NBSP               : Character renames No_Break_Space;
Inverted_Exclamation : constant Character := '¡'; --<Character'Val(161)
Cent_Sign          : constant Character := '¢'; --<Character'Val(162)
Pound_Sign         : constant Character := '£'; --<Character'Val(163)
Currency_Sign      : constant Character := '¤'; --<Character'Val(164)
Yen_Sign           : constant Character := '¥'; --<Character'Val(165)
Broken_Bar         : constant Character := '¦'; --<Character'Val(166)
Section_Sign       : constant Character := '§'; --<Character'Val(167)
Diaeresis          : constant Character := '¨'; --<Character'Val(168)
Copyright_Sign     : constant Character := '©'; --<Character'Val(169)
Feminine_Ordinal_Indicator : constant Character := 'ª'; --<Character'Val(170)
Left_Angle_Quotation : constant Character := '«'; --<Character'Val(171)
Not_Sign           : constant Character := '¬'; --<Character'Val(172)
Soft_Hyphen        : constant Character := '¸'; --<Character'Val(173)>
Registered_Trade_Mark_Sign : constant Character := '®'; --<Character'Val(174)
Macron             : constant Character := '¯'; --<Character'Val(175)
```

22

```
--< Character positions 176 (16#B0#) .. 191 (16#BF#):>
```

```
Degree_Sign       : constant Character := '°'; --<Character'Val(176)
Ring_Above        : Character renames Degree_Sign;
Plus_Minus_Sign   : constant Character := '±'; --<Character'Val(177)
Superscript_Two   : constant Character := '²'; --<Character'Val(178)
Superscript_Three : constant Character := '³'; --<Character'Val(179)
Acute             : constant Character := '´'; --<Character'Val(180)
Micro_Sign        : constant Character := 'µ'; --<Character'Val(181)
Pilcrow_Sign      : constant Character := '¶'; --<Character'Val(182)
Paragraph_Sign    : Character renames Pilcrow_Sign;
```

```

Middle_Dot           : constant Character := '·'; --<Character'Val(183)
Cedilla              : constant Character := '¸'; --<Character'Val(184)
Superscript_One      : constant Character := '¹'; --<Character'Val(185)
Masculine_Ordinal_Indicator: constant Character := 'º'; --<Character'Val(186)
Right_Angle_Quotation : constant Character := '»'; --<Character'Val(187)
Fraction_One_Quarter : constant Character := '¼'; --<Character'Val(188)
Fraction_One_Half     : constant Character := '½'; --<Character'Val(189)
Fraction_Three_Quarters : constant Character := '¾'; --<Character'Val(190)
Inverted_Question    : constant Character := '¿'; --<Character'Val(191)

```

23

```

--< Character positions 192 (16#C0#) .. 207 (16#CF#):>
UC_A_Grave           : constant Character := 'À'; --<Character'Val(192)
UC_A_Acute           : constant Character := 'Á'; --<Character'Val(193)
UC_A_Circumflex      : constant Character := 'Â'; --<Character'Val(194)
UC_A_Tilde           : constant Character := 'Ã'; --<Character'Val(195)
UC_A_Diaeresis       : constant Character := 'Ä'; --<Character'Val(196)
UC_A_Ring            : constant Character := 'Å'; --<Character'Val(197)
UC_AE_Diphthong      : constant Character := 'Æ'; --<Character'Val(198)
UC_C_Cedilla         : constant Character := 'Ç'; --<Character'Val(199)
UC_E_Grave           : constant Character := 'È'; --<Character'Val(200)
UC_E_Acute           : constant Character := 'É'; --<Character'Val(201)
UC_E_Circumflex      : constant Character := 'Ê'; --<Character'Val(202)
UC_E_Diaeresis       : constant Character := 'Ë'; --<Character'Val(203)
UC_I_Grave           : constant Character := 'Ì'; --<Character'Val(204)
UC_I_Acute           : constant Character := 'Í'; --<Character'Val(205)
UC_I_Circumflex      : constant Character := 'Î'; --<Character'Val(206)
UC_I_Diaeresis       : constant Character := 'Ï'; --<Character'Val(207)

```

24

```

--< Character positions 208 (16#D0#) .. 223 (16#DF#):>
UC_Icelandic_Eth     : constant Character := 'Ð'; --<Character'Val(208)
UC_N_Tilde           : constant Character := 'Ñ'; --<Character'Val(209)
UC_O_Grave           : constant Character := 'Ò'; --<Character'Val(210)
UC_O_Acute           : constant Character := 'Ó'; --<Character'Val(211)
UC_O_Circumflex      : constant Character := 'Ô'; --<Character'Val(212)
UC_O_Tilde           : constant Character := 'Õ'; --<Character'Val(213)
UC_O_Diaeresis       : constant Character := 'Ö'; --<Character'Val(214)
Multiplication_Sign   : constant Character := '×'; --<Character'Val(215)
UC_O_Oblique_Stroke  : constant Character := 'Ø'; --<Character'Val(216)
UC_U_Grave           : constant Character := 'Ù'; --<Character'Val(217)
UC_U_Acute           : constant Character := 'Ú'; --<Character'Val(218)
UC_U_Circumflex      : constant Character := 'Û'; --<Character'Val(219)
UC_U_Diaeresis       : constant Character := 'Ü'; --<Character'Val(220)
UC_Y_Acute           : constant Character := 'Ý'; --<Character'Val(221)
UC_Icelandic_Thorn   : constant Character := 'Þ'; --<Character'Val(222)

```

```
LC_German_Sharp_S      : constant Character := 'ß'; --<Character'Val(223)
```

25

```
--< Character positions 224 (16#E0#) .. 239 (16#EF#):>
LC_A_Grave             : constant Character := 'à'; --<Character'Val(224)
LC_A_Acute             : constant Character := 'á'; --<Character'Val(225)
LC_A_Circumflex       : constant Character := 'â'; --<Character'Val(226)
LC_A_Tilde            : constant Character := 'ã'; --<Character'Val(227)
LC_A_Diaeresis       : constant Character := 'ä'; --<Character'Val(228)
LC_A_Ring             : constant Character := 'å'; --<Character'Val(229)
LC_AE_Diphthong      : constant Character := 'æ'; --<Character'Val(230)
LC_C_Cedilla         : constant Character := 'ç'; --<Character'Val(231)
LC_E_Grave           : constant Character := 'è'; --<Character'Val(232)
LC_E_Acute           : constant Character := 'é'; --<Character'Val(233)
LC_E_Circumflex     : constant Character := 'ê'; --<Character'Val(234)
LC_E_Diaeresis     : constant Character := 'ë'; --<Character'Val(235)
LC_I_Grave          : constant Character := 'ì'; --<Character'Val(236)
LC_I_Acute          : constant Character := 'í'; --<Character'Val(237)
LC_I_Circumflex    : constant Character := 'î'; --<Character'Val(238)
LC_I_Diaeresis    : constant Character := 'ï'; --<Character'Val(239)
```

26

```
--< Character positions 240 (16#F0#) .. 255 (16#FF#):>
LC_Icelandic_Eth     : constant Character := 'ð'; --<Character'Val(240)
LC_N_Tilde          : constant Character := 'ñ'; --<Character'Val(241)
LC_O_Grave         : constant Character := 'ò'; --<Character'Val(242)
LC_O_Acute        : constant Character := 'ó'; --<Character'Val(243)
LC_O_Circumflex   : constant Character := 'ô'; --<Character'Val(244)
LC_O_Tilde       : constant Character := 'õ'; --<Character'Val(245)
LC_O_Diaeresis   : constant Character := 'ö'; --<Character'Val(246)
Division_Sign    : constant Character := '÷'; --<Character'Val(247)
LC_O_Oblique_Stroke : constant Character := 'ø'; --<Character'Val(248)
LC_U_Grave       : constant Character := 'ù'; --<Character'Val(249)
LC_U_Acute      : constant Character := 'ú'; --<Character'Val(250)
LC_U_Circumflex : constant Character := 'û'; --<Character'Val(251)
LC_U_Diaeresis  : constant Character := 'ü'; --<Character'Val(252)
LC_Y_Acute     : constant Character := 'ý'; --<Character'Val(253)
LC_Icelandic_Thorn : constant Character := 'þ'; --<Character'Val(254)
LC_Y_Diaeresis : constant Character := 'ÿ'; --<Character'Val(255)
end Ada.Characters.Latin_1;
```

Implementation Permissions

27

An implementation may provide additional packages as children of `Ada.Characters`, to declare names for the symbols of the local character set or other character sets.

15.3.4 A.3.4 The Package Characters.Conversions

Static Semantics

1/2

The library package Characters.Conversions has the following declaration:

2/2

```
package Ada.Characters.Conversions is
  pragma Pure(Conversions);
```

3/2

```
function Is_Character (Item : in Wide_Character)      return Boolean;
function Is_String    (Item : in Wide_String)        return Boolean;
function Is_Character (Item : in Wide_Wide_Character) return Boolean;
function Is_String    (Item : in Wide_Wide_String)   return Boolean;
function Is_Wide_Character (Item : in Wide_Wide_Character)
  return Boolean;
function Is_Wide_String   (Item : in Wide_Wide_String)
  return Boolean;
```

4/2

```
function To_Wide_Character (Item : in Character) return Wide_Character;
function To_Wide_String   (Item : in String)   return Wide_String;
function To_Wide_Wide_Character (Item : in Character)
  return Wide_Wide_Character;
function To_Wide_Wide_String   (Item : in String)
  return Wide_Wide_String;
function To_Wide_Wide_Character (Item : in Wide_Character)
  return Wide_Wide_Character;
function To_Wide_Wide_String   (Item : in Wide_String)
  return Wide_Wide_String;
```

5/2

```
function To_Character (Item      : in Wide_Character;
                      Substitute : in Character := ' ')
  return Character;
function To_String   (Item      : in Wide_String;
                      Substitute : in Character := ' ')
  return String;
function To_Character (Item : in Wide_Wide_Character;
                      Substitute : in Character := ' ')
  return Character;
function To_String   (Item : in Wide_Wide_String;
                      Substitute : in Character := ' ')
  return String;
```

```

function To_Wide_Character (Item :      in Wide_Wide_Character;
                           Substitute : in Wide_Character := ' ')■
    return Wide_Character;
function To_Wide_String   (Item :      in Wide_Wide_String;
                           Substitute : in Wide_Character := ' ')■
    return Wide_String;

```

6/2

```

end Ada.Characters.Conversions;

```

7/2

The functions in package Characters.Conversions test Wide_Wide_Character or Wide_Character values for membership in Wide_Character or Character, or convert between corresponding characters of Wide_Wide_Character, Wide_Character, and Character.

8/2

```

function Is_Character (Item : in Wide_Character) return Boolean;

```

9/2

```

Returns True if Wide_Character'Pos(Item)
<= Character'Pos(Character'Last).

```

10/2

```

function Is_Character (Item : in Wide_Wide_Character) return Boolean;■

```

11/2

```

Returns True if Wide_Wide_Character'Pos(Item)
<= Character'Pos(Character'Last).

```

12/2

```

function Is_Wide_Character (Item : in Wide_Wide_Character) return Boolean;■

```

13/2

```

Returns True if Wide_Wide_Character'Pos(Item)
<= Wide_Character'Pos(Wide_Character'Last).

```

14/2

```

function Is_String (Item : in Wide_String)      return Boolean;
function Is_String (Item : in Wide_Wide_String) return Boolean;

```

15/2

```

Returns True if Is_Character(Item(I)) is True
for each I in Item'Range.

```

16/2

```
function Is_Wide_String (Item : in Wide_Wide_String) return Boolean;■
```

17/2

Returns True if Is_Wide_Character(Item(I))
is True for each I in Item'Range.

18/2

```
function To_Character (Item :      in Wide_Character;  
                      Substitute : in Character := ' ') return Character;■  
function To_Character (Item :      in Wide_Wide_Character;  
                      Substitute : in Character := ' ') return Character;■
```

19/2

Returns the Character corresponding to Item
if Is_Character(Item), and returns the Sub-
stitute Character otherwise.

20/2

```
function To_Wide_Character (Item : in Character) return Wide_Character;■
```

21/2

Returns the Wide_Character X such that
Character'Pos(Item) = Wide_Character'Pos
(X).

22/2

```
function To_Wide_Character (Item :      in Wide_Wide_Character;  
                          Substitute : in Wide_Character := ' ')  
  return Wide_Character;
```

23/2

Returns the Wide_Character corresponding
to Item if Is_Wide_Character(Item), and re-
turns the Substitute Wide_Character other-
wise.

24/2

```
function To_Wide_Wide_Character (Item : in Character)  
  return Wide_Wide_Character;
```

25/2

Returns the Wide_Wide_Character X such that Character'Pos(Item) = Wide_Wide_Character'Pos (X).

26/2

```
function To_Wide_Wide_Character (Item : in Wide_Character)
  return Wide_Wide_Character;
```

27/2

Returns the Wide_Wide_Character X such that Wide_Character'Pos(Item) = Wide_Wide_Character'Pos (X).

28/2

```
function To_String (Item :      in Wide_String;
                   Substitute : in Character := ' ') return String;■
function To_String (Item :      in Wide_Wide_String;
                   Substitute : in Character := ' ') return String;■
```

29/2

Returns the String whose range is 1..Item'Length and each of whose elements is given by To_Character of the corresponding element in Item.

30/2

```
function To_Wide_String (Item : in String) return Wide_String;
```

31/2

Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item.

32/2

```
function To_Wide_String (Item :      in Wide_Wide_String;
                       Substitute : in Wide_Character := ' ')
  return Wide_String;
```

33/2

Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the

corresponding element in Item with the given Substitute Wide_Character.

34/2

```
function To_Wide_Wide_String (Item : in String) return Wide_Wide_String;
function To_Wide_Wide_String (Item : in Wide_String)
  return Wide_Wide_String;
```

35/2

Returns the Wide_Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Wide_Character of the corresponding element in Item.

15.4 A.4 String Handling

1/2

This clause presents the specifications of the package Strings and several child packages, which provide facilities for dealing with string data. Fixed-length, bounded-length, and unbounded-length strings are supported, for String, Wide_String, and Wide_Wide_String. The string-handling subprograms include searches for pattern strings and for characters in program-specified sets, translation (via a character-to-character mapping), and transformation (replacing, inserting, overwriting, and deleting of substrings).

15.4.1 A.4.1 The Package Strings

1

The package Strings provides declarations common to the string handling packages.

Static Semantics

2

The library package Strings has the following declaration:

3

```
package Ada.Strings is
  pragma Pure(Strings);
```

4/2

```
Space      : constant Character      := ' ';
Wide_Space : constant Wide_Character := ' ';
Wide_Wide_Space : constant Wide_Wide_Character := ' ';
```

5

```
Length_Error, Pattern_Error, Index_Error, Translation_Error : exception;
```

6

```
type Alignment is (Left, Right, Center);
```



```

    type Truncation is (Left, Right, Error);
    type Membership is (Inside, Outside);
    type Direction is (Forward, Backward);
    type Trim_End is (Left, Right, Both);
end Ada.Strings;

```

15.4.2 A.4.2 The Package Strings.Maps

1

The package Strings.Maps defines the types, operations, and other entities needed for character sets and character-to-character mappings.

Static Semantics

2

The library package Strings.Maps has the following declaration:

3/2

```

package Ada.Strings.Maps is
    pragma Pure(Maps);

```

4/2

```

    --< Representation for a set of character values:>
    type Character_Set is private;
    pragma Preelaborable_Initialization(Character_Set);

```

5

```

    Null_Set : constant Character_Set;

```

6

```

    type Character_Range is
        record
            Low : Character;
            High : Character;
        end record;
    -- <Represents Character range Low..High>

```

7

```

    type Character_Ranges is array (Positive range <>) of Character_Range;

```

8

```

    function To_Set (Ranges : in Character_Ranges) return Character_Set;

```

9

```

    function To_Set (Span : in Character_Range) return Character_Set;

```

10

```
function To_Ranges (Set      : in Character_Set) return Character_Ranges;█
```

11

```
function "=" (Left, Right : in Character_Set) return Boolean;
```

12

```
function "not" (Right : in Character_Set)      return Character_Set;█  
function "and" (Left, Right : in Character_Set) return Character_Set;█  
function "or"  (Left, Right : in Character_Set) return Character_Set;█  
function "xor" (Left, Right : in Character_Set) return Character_Set;█  
function "-"   (Left, Right : in Character_Set) return Character_Set;█
```

13

```
function Is_In (Element : in Character;  
               Set      : in Character_Set)  
  return Boolean;
```

14

```
function Is_Subset (Elements : in Character_Set;  
                  Set       : in Character_Set)  
  return Boolean;
```

15

```
function "<=" (Left  : in Character_Set;  
            Right : in Character_Set)  
  return Boolean renames Is_Subset;
```

16

```
--< Alternative representation for a set of character values:>  
subtype Character_Sequence is String;
```

17

```
function To_Set (Sequence : in Character_Sequence) return Character_Set;█
```

18

```
function To_Set (Singleton : in Character)      return Character_Set;█
```

19

```
function To_Sequence (Set : in Character_Set) return Character_Sequence;█
```

20/2

```
--< Representation for a character to character mapping:>  
type Character_Mapping is private;  
pragma Preelaborable_Initialization(Character_Mapping);
```

21

```
function Value (Map      : in Character_Mapping;  
               Element : in Character)  
  return Character;
```

22

```
Identity : constant Character_Mapping;
```

23

```
function To_Mapping (From, To : in Character_Sequence)  
  return Character_Mapping;
```

24

```
function To_Domain (Map : in Character_Mapping)  
  return Character_Sequence;  
function To_Range  (Map : in Character_Mapping)  
  return Character_Sequence;
```

25

```
type Character_Mapping_Function is  
  access function (From : in Character) return Character;
```

26

```
private  
  ... -- <not specified by the language>  
end Ada.Strings.Maps;
```

27

An object of type `Character_Set` represents a set of characters.

28

`Null_Set` represents the set containing no characters.

29

An object `Obj` of type `Character_Range` represents the set of characters in the range `Obj.Low .. Obj.High`.

30

An object `Obj` of type `Character_Ranges` represents the union of the sets corresponding to `Obj(I)` for `I` in `Obj'Range`.

31

```
function To_Set (Ranges : in Character_Ranges) return Character_Set;■
```

32

If Ranges'Length=0 then Null_Set is returned; otherwise the returned value represents the set corresponding to Ranges.

33

```
function To_Set (Span : in Character_Range) return Character_Set;
```

34

The returned value represents the set containing each character in Span.

35

```
function To_Ranges (Set : in Character_Set) return Character_Ranges;■
```

36

If Set = Null_Set then an empty Character_Ranges array is returned; otherwise the shortest array of contiguous ranges of Character values in Set, in increasing order of Low, is returned.

37

```
function "=" (Left, Right : in Character_Set) return Boolean;
```

38

The function "=" returns True if Left and Right represent identical sets, and False otherwise.

39

Each of the logical operators "not", "and", "or", and "xor" returns a Character_Set value that represents the set obtained by applying the corresponding operation to the set(s) represented by the parameter(s) of the operator. "-"(Left, Right) is equivalent to "and"(Left, "not"(Right)).

40

```
function Is_In (Element : in Character;
               Set      : in Character_Set);
return Boolean;
```

41

Is_In returns True if Element is in Set, and False otherwise.

42

```
function Is_Subset (Elements : in Character_Set;
                   Set       : in Character_Set)
  return Boolean;
```

43

Is_Subset returns True if Elements is a subset of Set, and False otherwise.

44

```
subtype Character_Sequence is String;
```

45

The Character_Sequence subtype is used to portray a set of character values and also to identify the domain and range of a character mapping.

46

```
function To_Set (Sequence : in Character_Sequence) return Character_Set;■
```

```
function To_Set (Singleton : in Character)          return Character_Set;■
```

47

Sequence portrays the set of character values that it explicitly contains (ignoring duplicates). Singleton portrays the set comprising a single Character. Each of the To_Set functions returns a Character_Set value that represents the set portrayed by Sequence or Singleton.

48

```
function To_Sequence (Set : in Character_Set) return Character_Sequence;■
```

49

The function To_Sequence returns a Character_Sequence value containing each of the

characters in the set represented by Set, in ascending order with no duplicates.

50

```
type Character_Mapping is private;
```

51

An object of type Character_Mapping represents a Character-to-Character mapping.

52

```
function Value (Map      : in Character_Mapping;  
               Element  : in Character)  
  return Character;
```

53

The function Value returns the Character value to which Element maps with respect to the mapping represented by Map.

54

A character C <matches> a pattern character P with respect to a given Character_Mapping value Map if Value(Map, C) = P. A string S <matches> a pattern string P with respect to a given Character_Mapping if their lengths are the same and if each character in S matches its corresponding character in the pattern string P.

55

String handling subprograms that deal with character mappings have parameters whose type is Character_Mapping.

56

```
Identity : constant Character_Mapping;
```

57

Identity maps each Character to itself.

58

```
function To_Mapping (From, To : in Character_Sequence)  
  return Character_Mapping;
```

59

To-Mapping produces a Character_Mapping such that each element of From maps to the corresponding element of To, and each other character maps to itself. If From'Length /=

To'Length, or if some character is repeated in From, then Translation_Error is propagated.

60

```
function To_Domain (Map : in Character_Mapping) return Character_Sequence;■
```

61

To_Domain returns the shortest Character_Sequence value D such that each character not in D maps to itself, and such that the characters in D are in ascending order. The lower bound of D is 1.

62

```
function To_Range (Map : in Character_Mapping) return Character_Sequence;■
```

63/1

To_Range returns the Character_Sequence value R, such that if D = To_Domain(Map), then R has the same bounds as D, and D(I) maps to R(I) for each I in D'Range.

64

An object F of type Character_Mapping_Function maps a Character value C to the Character value F.all(C), which is said to <match> C with respect to mapping function F.

NOTES

65

7 Character_Mapping and Character_Mapping_Function are used both for character equivalence mappings in the search subprograms (such as for case insensitivity) and as transformational mappings in the Translate subprograms.

66

8 To_Domain(Identity) and To_Range(Identity) each returns the null string.

Examples

67

To_Mapping("ABCD", "ZZAB") returns a Character_Mapping that maps 'A' and 'B' to 'Z', 'C' to 'A', 'D' to 'B', and each other Character to itself.

15.4.3 A.4.3 Fixed-Length String Handling

1

The language-defined package Strings.Fixed provides string-handling subprograms for

fixed-length strings; that is, for values of type `Standard.String`. Several of these subprograms are procedures that modify the contents of a `String` that is passed as an out or an in out parameter; each has additional parameters to control the effect when the logical length of the result differs from the parameter's length.

2

For each function that returns a `String`, the lower bound of the returned value is 1.

3

The basic model embodied in the package is that a fixed-length string comprises significant characters and possibly padding (with space characters) on either or both ends. When a shorter string is copied to a longer string, padding is inserted, and when a longer string is copied to a shorter one, padding is stripped. The `Move` procedure in `Strings.Fixed`, which takes a `String` as an out parameter, allows the programmer to control these effects. Similar control is provided by the string transformation procedures.

Static Semantics

4

The library package `Strings.Fixed` has the following declaration:

5

```
with Ada.Strings.Maps;
package Ada.Strings.Fixed is
  pragma Preelaborate(Fixed);
```

6

```
--< "Copy" procedure for strings of possibly different lengths>
```

7

```
  procedure Move (Source  : in  String;
                 Target  : out String;
                 Drop     : in  Truncation := Error;
                 Justify  : in  Alignment  := Left;
                 Pad      : in  Character  := Space);
```

8

```
--< Search subprograms>
```

8.1/2

```
  function Index (Source  : in String;
                 Pattern  : in String;
                 From     : in Positive;
                 Going    : in Direction := Forward;
                 Mapping  : in Maps.Character_Mapping := Maps.Identity)
    return Natural;
```

8.2/2


```
function Index (Source : in String;
               Pattern : in String;
               From    : in Positive;
               Going   : in Direction := Forward;
               Mapping : in Maps.Character_Mapping_Function)
  return Natural;
```

9

```
function Index (Source : in String;
               Pattern : in String;
               Going   : in Direction := Forward;
               Mapping : in Maps.Character_Mapping_Function
                       := Maps.Identity)
  return Natural;
```

10

```
function Index (Source : in String;
               Pattern : in String;
               Going   : in Direction := Forward;
               Mapping : in Maps.Character_Mapping_Function)
  return Natural;
```

10.1/2

```
function Index (Source : in String;
               Set      : in Maps.Character_Set;
               From    : in Positive;
               Test     : in Membership := Inside;
               Going   : in Direction := Forward)
  return Natural;
```

11

```
function Index (Source : in String;
               Set      : in Maps.Character_Set;
               Test     : in Membership := Inside;
               Going   : in Direction := Forward)
  return Natural;
```

11.1/2

```
function Index_Non_Blank (Source : in String;
                         From    : in Positive;
                         Going   : in Direction := Forward)
  return Natural;
```

12

```
function Index_Non_Blank (Source : in String;
```

```

    Going : in Direction := Forward)
return Natural;
13

function Count (Source : in String;
               Pattern : in String;
               Mapping : in Maps.Character_Mapping
                   := Maps.Identity)
return Natural;
14

function Count (Source : in String;
               Pattern : in String;
               Mapping : in Maps.Character_Mapping_Function)
return Natural;
15

function Count (Source : in String;
               Set : in Maps.Character_Set)
return Natural;
16

procedure Find-Token (Source : in String;
                    Set : in Maps.Character_Set;
                    Test : in Membership;
                    First : out Positive;
                    Last : out Natural);
17

--< String translation subprograms>
18

function Translate (Source : in String;
                  Mapping : in Maps.Character_Mapping)
return String;
19

procedure Translate (Source : in out String;
                   Mapping : in Maps.Character_Mapping);
20

function Translate (Source : in String;
                  Mapping : in Maps.Character_Mapping_Function)■

```

```

    return String;
21
    procedure Translate (Source  : in out String;
                        Mapping : in Maps.Character_Mapping_Function);█
22
--< String transformation subprograms>
23
    function Replace_Slice (Source  : in String;
                            Low     : in Positive;
                            High    : in Natural;
                            By      : in String)
        return String;
24
    procedure Replace_Slice (Source  : in out String;
                            Low     : in Positive;
                            High    : in Natural;
                            By      : in String;
                            Drop    : in Truncation := Error;
                            Justify : in Alignment  := Left;
                            Pad     : in Character  := Space);
25
    function Insert (Source  : in String;
                    Before  : in Positive;
                    New_Item : in String)
        return String;
26
    procedure Insert (Source  : in out String;
                     Before  : in Positive;
                     New_Item : in String;
                     Drop    : in Truncation := Error);
27
    function Overwrite (Source  : in String;
                       Position : in Positive;
                       New_Item : in String)
        return String;

```

28

```
procedure Overwrite (Source   : in out String;
                    Position : in Positive;
                    New_Item  : in String;
                    Drop      : in Truncation := Right);
```

29

```
function Delete (Source : in String;
                From    : in Positive;
                Through : in Natural)
  return String;
```

30

```
procedure Delete (Source : in out String;
                 From    : in Positive;
                 Through : in Natural;
                 Justify : in Alignment := Left;
                 Pad     : in Character := Space);
```

31

--<String selector subprograms>

```
function Trim (Source : in String;
              Side    : in Trim_End)
  return String;
```

32

```
procedure Trim (Source : in out String;
               Side    : in Trim_End;
               Justify : in Alignment := Left;
               Pad     : in Character := Space);
```

33

```
function Trim (Source : in String;
              Left    : in Maps.Character_Set;
              Right   : in Maps.Character_Set)
  return String;
```

34

```
procedure Trim (Source : in out String;
               Left    : in Maps.Character_Set;
               Right   : in Maps.Character_Set;
               Justify : in Alignment := Strings.Left;
               Pad     : in Character := Space);
```

35

```
function Head (Source : in String;
              Count  : in Natural;
              Pad    : in Character := Space)
  return String;
```

36

```
procedure Head (Source : in out String;
               Count  : in Natural;
               Justify : in Alignment := Left;
               Pad    : in Character := Space);
```

37

```
function Tail (Source : in String;
              Count  : in Natural;
              Pad    : in Character := Space)
  return String;
```

38

```
procedure Tail (Source : in out String;
               Count  : in Natural;
               Justify : in Alignment := Left;
               Pad    : in Character := Space);
```

39

--<String constructor functions>

40

```
function "*" (Left  : in Natural;
             Right : in Character) return String;
```

41

```
function "*" (Left  : in Natural;
             Right : in String) return String;
```

42

```
end Ada.Strings.Fixed;
```

43

The effects of the above subprograms are as follows.

44

```
procedure Move (Source : in String;
```

```
Target : out String;
Drop   : in  Truncation := Error;
Justify : in  Alignment  := Left;
Pad    : in  Character  := Space);
```

45

The Move procedure copies characters from Source to Target. If Source has the same length as Target, then the effect is to assign Source to Target. If Source is shorter than Target then:

46

- If Justify=Left, then Source is copied into the first Source'Length characters of Target.

47

- If Justify=Right, then Source is copied into the last Source'Length characters of Target.

48

- If Justify=Center, then Source is copied into the middle Source'Length characters of Target. In this case, if the difference in length between Target and Source is odd, then the extra Pad character is on the right.

49

- Pad is copied to each Target character not otherwise assigned.

50

If Source is longer than Target, then the effect is based on Drop.

51

- If Drop=Left, then the rightmost Target'Length characters of Source are copied into Target.

52

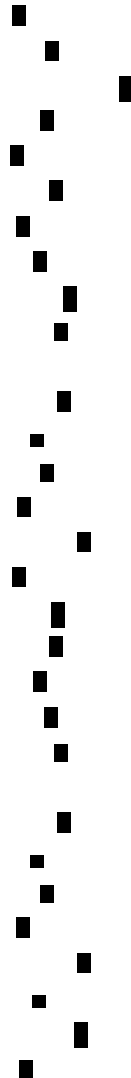
- If Drop=Right, then the leftmost Target'Length characters of Source are copied into Target.

53

- If Drop=Error, then the effect depends on the value of the Justify parameter and also on whether any characters in Source other than Pad would fail to be copied:

54

- If Justify=Left, and if each of the rightmost Source'Length-Target'Length characters in Source is Pad, then the leftmost Target'Length characters of Source are copied to



Tar-
get.

- If
Jus-
tify=Right,
and
if
each
of
the
left-
most
 $Source'Length - Target'Length$
char-
ac-
ters
in
Source
is
Pad,
then
the
right-
most
 $Target'Length$
char-
ac-
ters
of
Source
are
copied
to
Tar-
get.

- Otherwise,
 $Length_Error$
is
prop-
a-
gated.

56.1/2

```
function Index (Source : in String;
               Pattern : in String;
               From    : in Positive;
               Going   : in Direction := Forward;
               Mapping : in Maps.Character_Mapping := Maps.Identity)
return Natural;

function Index (Source : in String;
               Pattern : in String;
               From    : in Positive;
               Going   : in Direction := Forward;
               Mapping : in Maps.Character_Mapping_Function)
return Natural;
```

56.2/2

Each Index function searches, starting from From, for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If From is not in Source'Range, then Index_Error is propagated. If Going = Forward, then Index returns the smallest index I which is greater than or equal to From such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern and has an upper bound less than or equal to From. If there is no such slice, then 0 is returned. If Pattern is the null string, then Pattern_Error is propagated.

57

```
function Index (Source : in String;
               Pattern : in String;
               Going   : in Direction := Forward;
               Mapping : in Maps.Character_Mapping
                       := Maps.Identity)
return Natural;

function Index (Source : in String;
               Pattern : in String;
               Going   : in Direction := Forward;
```

```
        Mapping : in Maps.Character_Mapping_Function)
return Natural;
```

58/2

If Going = Forward, returns

58.1/2

```
Index (Source, Pattern, Source'First, Forward, Mapping);
```

58.2/2

otherwise returns

58.3/2

```
Index (Source, Pattern, Source'Last, Backward, Mapping);
```

58.4/2

```
function Index (Source : in String;
                Set      : in Maps.Character_Set;
                From     : in Positive;
                Test      : in Membership := Inside;
                Going    : in Direction := Forward)
return Natural;
```

58.5/2

Index searches for the first or last occurrence of any of a set of characters (when Test=Inside), or any of the complement of a set of characters (when Test=Outside). If From is not in Source'Range, then Index_Error is propagated. Otherwise, it returns the smallest index $I \geq \text{From}$ (if Going=Forward) or the largest index $I \leq \text{From}$ (if Going=Backward) such that Source(I) satisfies the Test condition with respect to Set; it returns 0 if there is no such Character in Source.

59

```
function Index (Source : in String;
                Set      : in Maps.Character_Set;
                Test      : in Membership := Inside;
                Going    : in Direction := Forward)
return Natural;
```

60/2

If Going = Forward, returns

60.1/2

```
Index (Source, Set, Source'First, Test, Forward);
```

60.2/2

otherwise returns

60.3/2

```
Index (Source, Set, Source'Last, Test, Backward);
```

60.4/2

```
function Index_Non_Blank (Source : in String;  
                          From   : in Positive;  
                          Going  : in Direction := Forward)  
  return Natural;
```

60.5/2

Returns Index (Source, Maps.To_Set(Space),
From, Outside, Going);

61

```
function Index_Non_Blank (Source : in String;  
                          Going  : in Direction := Forward)  
  return Natural;
```

62

Returns Index(Source, Maps.To_Set(Space),
Outside, Going)

63

```
function Count (Source   : in String;  
               Pattern  : in String;  
               Mapping  : in Maps.Character_Mapping  
                       := Maps.Identity)  
  return Natural;  
  
function Count (Source   : in String;  
               Pattern  : in String;  
               Mapping  : in Maps.Character_Mapping_Function)  
  return Natural;
```

64

Returns the maximum number of nonoverlapping slices of Source that match Pattern with respect to Mapping. If Pattern is the null string then Pattern_Error is propagated.

65

```
function Count (Source   : in String;
               Set      : in Maps.Character_Set)
  return Natural;
```

66

Returns the number of occurrences in Source of characters that are in Set.

67

```
procedure Find-Token (Source : in String;
                    Set      : in Maps.Character_Set;
                    Test     : in Membership;
                    First    : out Positive;
                    Last     : out Natural);
```

68/1

Find-Token returns in First and Last the indices of the beginning and end of the first slice of Source all of whose elements satisfy the Test condition, and such that the elements (if any) immediately before and after the slice do not satisfy the Test condition. If no such slice exists, then the value returned for Last is zero, and the value returned for First is Source'First; however, if Source'First is not in Positive then Constraint_Error is raised.

69

```
function Translate (Source : in String;
                  Mapping  : in Maps.Character_Mapping)
  return String;

function Translate (Source : in String;
                  Mapping  : in Maps.Character_Mapping_Function)
  return String;
```

70

Returns the string S whose length is $\text{Source}'\text{Length}$ and such that $S(I)$ is the character to which Mapping maps the corresponding element of Source , for I in $1..\text{Source}'\text{Length}$.

71

```
procedure Translate (Source  : in out String;
                   Mapping  : in Maps.Character_Mapping);

procedure Translate (Source  : in out String;
                   Mapping  : in Maps.Character_Mapping_Function);
```

72

Equivalent to $\text{Source} := \text{Translate}(\text{Source}, \text{Mapping})$.

73

```
function Replace_Slice (Source  : in String;
                       Low      : in Positive;
                       High     : in Natural;
                       By       : in String)
  return String;
```

74/1

If $\text{Low} > \text{Source}'\text{Last}+1$, or $\text{High} < \text{Source}'\text{First}-1$, then Index_Error is propagated. Otherwise:

74.1/1

- If $\text{High} \geq \text{Low}$, then the returned string comprises $\text{Source}(\text{Source}'\text{First}..\text{Low}-1) \& \text{By} \& \text{Source}(\text{High}+1..\text{Source}'\text{Last})$, but with lower bound 1.

74.2/1

- If $\text{High} < \text{Low}$, then the returned string is $\text{Insert}(\text{Source}, \text{Before} \Rightarrow \text{Low}, \text{New_Item} \Rightarrow \text{By})$.

75

```
procedure Replace_Slice (Source  : in out String;
                       Low      : in Positive;
```

```
High      : in Natural;  
By        : in String;  
Drop      : in Truncation := Error;  
Justify   : in Alignment  := Left;  
Pad       : in Character  := Space);
```

76

Equivalent to Move(Replace_Slice(Source,
Low, High, By), Source, Drop, Justify, Pad).

77

```
function Insert (Source   : in String;  
                Before    : in Positive;  
                New_Item  : in String)  
  return String;
```

78

Propagates Index_Error if Before is not in
Source'First .. Source'Last+1; otherwise
returns Source(Source'First..Before-1) &
New_Item & Source(Before..Source'Last),
but with lower bound 1.

79

```
procedure Insert (Source   : in out String;  
                 Before    : in Positive;  
                 New_Item  : in String;  
                 Drop      : in Truncation := Error);
```

80

Equivalent to Move(Insert(Source, Before,
New_Item), Source, Drop).

81

```
function Overwrite (Source   : in String;  
                   Position  : in Positive;  
                   New_Item  : in String)  
  return String;
```

82

Propagates Index_Error if Position is not
in Source'First .. Source'Last+1; otherwise
returns the string obtained from Source by
consecutively replacing characters starting

at Position with corresponding characters from New_Item. If the end of Source is reached before the characters in New_Item are exhausted, the remaining characters from New_Item are appended to the string.

83

```
procedure Overwrite (Source   : in out String;
                    Position : in Positive;
                    New_Item  : in String;
                    Drop      : in Truncation := Right);
```

84

Equivalent to Move(Overwrite(Source, Position, New_Item), Source, Drop).

85

```
function Delete (Source  : in String;
                From     : in Positive;
                Through  : in Natural)
return String;
```

86/1

If From <= Through, the returned string is Replace_Slice(Source, From, Through, ""), otherwise it is Source with lower bound 1.

87

```
procedure Delete (Source  : in out String;
                From     : in Positive;
                Through  : in Natural;
                Justify  : in Alignment := Left;
                Pad      : in Character := Space);
```

88

Equivalent to Move(Delete(Source, From, Through), Source, Justify => Justify, Pad => Pad).

89

```
function Trim (Source : in String;
              Side   : in Trim_End)
return String;
```

90

Returns the string obtained by removing from Source all leading Space characters (if Side = Left), all trailing Space characters (if Side = Right), or all leading and trailing Space characters (if Side = Both).

91

```
procedure Trim (Source : in out String;
               Side    : in Trim_End;
               Justify  : in Alignment := Left;
               Pad     : in Character := Space);
```

92

Equivalent to Move(Trim(Source, Side), Source, Justify=>Justify, Pad=>Pad).

93

```
function Trim (Source : in String;
              Left    : in Maps.Character_Set;
              Right   : in Maps.Character_Set)
return String;
```

94

Returns the string obtained by removing from Source all leading characters in Left and all trailing characters in Right.

95

```
procedure Trim (Source : in out String;
               Left    : in Maps.Character_Set;
               Right   : in Maps.Character_Set;
               Justify  : in Alignment := Strings.Left;
               Pad     : in Character := Space);
```

96

Equivalent to Move(Trim(Source, Left, Right), Source, Justify => Justify, Pad=>Pad).

97

```
function Head (Source : in String;
```



```
        Count  : in Natural;
        Pad    : in Character := Space)
return String;
```

98

Returns a string of length Count. If Count <= Source'Length, the string comprises the first Count characters of Source. Otherwise its contents are Source concatenated with Count-Source'Length Pad characters.

99

```
procedure Head (Source  : in out String;
               Count    : in Natural;
               Justify  : in Alignment := Left;
               Pad      : in Character := Space);
```

100

Equivalent to Move(Head(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad).

101

```
function Tail (Source : in String;
              Count    : in Natural;
              Pad      : in Character := Space)
return String;
```

102

Returns a string of length Count. If Count <= Source'Length, the string comprises the last Count characters of Source. Otherwise its contents are Count-Source'Length Pad characters concatenated with Source.

103

```
procedure Tail (Source  : in out String;
               Count    : in Natural;
               Justify  : in Alignment := Left;
               Pad      : in Character := Space);
```

104

Equivalent to `Move(Tail(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad)`.

105

```
function "*" (Left : in Natural;  
             Right : in Character) return String;  
  
function "*" (Left : in Natural;  
             Right : in String) return String;
```

106/1

These functions replicate a character or string a specified number of times. The first function returns a string whose length is `Left` and each of whose elements is `Right`. The second function returns a string whose length is `Left*Right'Length` and whose value is the null string if `Left = 0` and otherwise is `(Left-1)*Right & Right` with lower bound 1.

NOTES

107

9 In the `Index` and `Count` functions taking `Pattern` and `Mapping` parameters, the actual `String` parameter passed to `Pattern` should comprise characters occurring as target characters of the mapping. Otherwise the pattern will not match.

108

10 In the `Insert` subprograms, inserting at the end of a string is obtained by passing `Source'Last+1` as the `Before` parameter.

109

11 If a null `Character-Mapping-Function` is passed to any of the string handling subprograms, `Constraint_Error` is propagated.

15.4.4 A.4.4 Bounded-Length String Handling

1

The language-defined package `Strings.Bounded` provides a generic package each of whose instances yields a private type `Bounded_String` and a set of operations. An object of a particular `Bounded_String` type represents a `String` whose low bound is 1 and whose length can vary conceptually between 0 and a maximum size established at the generic instantiation. The subprograms for fixed-length string handling are either overloaded directly for `Bounded_String`, or are modified as needed to reflect the variability in length. Additionally,

since the Bounded_String type is private, appropriate constructor and selector operations are provided.

Static Semantics

2

The library package Strings.Bounded has the following declaration:

3

```
with Ada.Strings.Maps;  
package Ada.Strings.Bounded is  
  pragma Preelaborate(Bounded);
```

4

```
  generic  
    Max    : Positive;    --< Maximum length of a Bounded_String>  
  package Generic_Bounded_Length is
```

5

```
    Max_Length : constant Positive := Max;
```

6

```
    type Bounded_String is private;
```

7

```
    Null_Bounded_String : constant Bounded_String;
```

8

```
    subtype Length_Range is Natural range 0 .. Max_Length;
```

9

```
    function Length (Source : in Bounded_String) return Length_Range;■
```

10

```
    --< Conversion, Concatenation, and Selection functions>
```

11

```
    function To_Bounded_String (Source : in String;  
                                Drop    : in Truncation := Error)  
      return Bounded_String;
```

12

```
    function To_String (Source : in Bounded_String) return String;■
```

12.1/2

```
procedure Set_Bounded_String
  (Target : out Bounded_String;
   Source : in String;
   Drop   : in Truncation := Error);
```

13

```
function Append (Left, Right : in Bounded_String;
                 Drop         : in Truncation := Error)
  return Bounded_String;
```

14

```
function Append (Left  : in Bounded_String;
                 Right : in String;
                 Drop   : in Truncation := Error)
  return Bounded_String;
```

15

```
function Append (Left  : in String;
                 Right : in Bounded_String;
                 Drop   : in Truncation := Error)
  return Bounded_String;
```

16

```
function Append (Left  : in Bounded_String;
                 Right : in Character;
                 Drop   : in Truncation := Error)
  return Bounded_String;
```

17

```
function Append (Left  : in Character;
                 Right : in Bounded_String;
                 Drop   : in Truncation := Error)
  return Bounded_String;
```

18

```
procedure Append (Source  : in out Bounded_String;
                  New_Item : in Bounded_String;
                  Drop     : in Truncation := Error);
```

19

```
procedure Append (Source : in out Bounded_String;
```

```

                New_Item : in String;
                Drop      : in Truncation := Error);
20

procedure Append (Source   : in out Bounded_String;
                 New_Item  : in Character;
                 Drop      : in Truncation := Error);
21

function "&" (Left, Right : in Bounded_String)
    return Bounded_String;
22

function "&" (Left : in Bounded_String; Right : in String)
    return Bounded_String;
23

function "&" (Left : in String; Right : in Bounded_String)
    return Bounded_String;
24

function "&" (Left : in Bounded_String; Right : in Character)
    return Bounded_String;
25

function "&" (Left : in Character; Right : in Bounded_String)
    return Bounded_String;
26

function Element (Source : in Bounded_String;
                 Index   : in Positive)
    return Character;
27

procedure Replace_Element (Source : in out Bounded_String;
                          Index   : in Positive;
                          By      : in Character);
28

function Slice (Source : in Bounded_String;
               Low     : in Positive;
               High    : in Natural)
    return String;
```

28.1/2

```
function Bounded_Slice
  (Source : in Bounded_String;
   Low    : in Positive;
   High   : in Natural)
  return Bounded_String;
```

28.2/2

```
procedure Bounded_Slice
  (Source : in    Bounded_String;
   Target : out Bounded_String;
   Low    : in    Positive;
   High   : in    Natural);
```

29

```
function "=" (Left, Right : in Bounded_String) return Boolean;█
function "=" (Left : in Bounded_String; Right : in String)
  return Boolean;
```

30

```
function "=" (Left : in String; Right : in Bounded_String)
  return Boolean;
```

31

```
function "<" (Left, Right : in Bounded_String) return Boolean;█
```

32

```
function "<" (Left : in Bounded_String; Right : in String)
  return Boolean;
```

33

```
function "<" (Left : in String; Right : in Bounded_String)
  return Boolean;
```

34

```
function "<=" (Left, Right : in Bounded_String) return Boolean;█
```

35

```
function "<=" (Left : in Bounded_String; Right : in String)
  return Boolean;
```

36

```
function "<=" (Left : in String; Right : in Bounded_String)
  return Boolean;
```

37

```
function ">" (Left, Right : in Bounded_String) return Boolean;■
```

38

```
function ">" (Left : in Bounded_String; Right : in String)
  return Boolean;
```

39

```
function ">" (Left : in String; Right : in Bounded_String)
  return Boolean;
```

40

```
function ">=" (Left, Right : in Bounded_String) return Boolean;■
```

41

```
function ">=" (Left : in Bounded_String; Right : in String)
  return Boolean;
```

42

```
function ">=" (Left : in String; Right : in Bounded_String)
  return Boolean;
```

43/2

--< Search subprograms>

43.1/2

```
function Index (Source : in Bounded_String;
               Pattern : in String;
               From    : in Positive;
               Going   : in Direction := Forward;
               Mapping : in Maps.Character_Mapping := Maps.Identity)■
  return Natural;
```

43.2/2

```
function Index (Source : in Bounded_String;
               Pattern : in String;
               From    : in Positive;
               Going   : in Direction := Forward;
```

```
        Mapping : in Maps.Character_Mapping_Function)
return Natural;
```

44

```
function Index (Source   : in Bounded_String;
               Pattern   : in String;
               Going     : in Direction := Forward;
               Mapping   : in Maps.Character_Mapping
                       := Maps.Identity)
return Natural;
```

45

```
function Index (Source   : in Bounded_String;
               Pattern   : in String;
               Going     : in Direction := Forward;
               Mapping   : in Maps.Character_Mapping_Function)
return Natural;
```

45.1/2

```
function Index (Source   : in Bounded_String;
               Set       : in Maps.Character_Set;
               From      : in Positive;
               Test      : in Membership := Inside;
               Going     : in Direction := Forward)
return Natural;
```

46

```
function Index (Source : in Bounded_String;
               Set     : in Maps.Character_Set;
               Test    : in Membership := Inside;
               Going   : in Direction := Forward)
return Natural;
```

46.1/2

```
function Index_Non_Blank (Source : in Bounded_String;
                        From     : in Positive;
                        Going    : in Direction := Forward)
return Natural;
```

47

```
function Index_Non_Blank (Source : in Bounded_String;
                        Going    : in Direction := Forward)
return Natural;
```


48

```
function Count (Source   : in Bounded_String;
                Pattern  : in String;
                Mapping   : in Maps.Character_Mapping
                        := Maps.Identity)
    return Natural;
```

49

```
function Count (Source   : in Bounded_String;
                Pattern  : in String;
                Mapping   : in Maps.Character_Mapping_Function)
    return Natural;
```

50

```
function Count (Source   : in Bounded_String;
                Set       : in Maps.Character_Set)
    return Natural;
```

51

```
procedure Find-Token (Source : in Bounded_String;
                     Set     : in Maps.Character_Set;
                     Test    : in Membership;
                     First   : out Positive;
                     Last    : out Natural);
```

52

--< String translation subprograms>

53

```
function Translate (Source   : in Bounded_String;
                  Mapping   : in Maps.Character_Mapping)
    return Bounded_String;
```

54

```
procedure Translate (Source   : in out Bounded_String;
                   Mapping   : in Maps.Character_Mapping);
```

55

```
function Translate (Source   : in Bounded_String;
                  Mapping   : in Maps.Character_Mapping_Function)
    return Bounded_String;
```

56

```
procedure Translate (Source : in out Bounded_String;  
                    Mapping : in Maps.Character_Mapping_Function);
```

57

--< String transformation subprograms>

58

```
function Replace_Slice (Source : in Bounded_String;  
                       Low     : in Positive;  
                       High    : in Natural;  
                       By      : in String;  
                       Drop     : in Truncation := Error)  
return Bounded_String;
```

59

```
procedure Replace_Slice (Source : in out Bounded_String;  
                        Low     : in Positive;  
                        High    : in Natural;  
                        By      : in String;  
                        Drop     : in Truncation := Error);
```

60

```
function Insert (Source : in Bounded_String;  
                Before  : in Positive;  
                New_Item : in String;  
                Drop     : in Truncation := Error)  
return Bounded_String;
```

61

```
procedure Insert (Source : in out Bounded_String;  
                 Before  : in Positive;  
                 New_Item : in String;  
                 Drop     : in Truncation := Error);
```

62

```
function Overwrite (Source : in Bounded_String;  
                   Position : in Positive;  
                   New_Item : in String;  
                   Drop     : in Truncation := Error)  
return Bounded_String;
```

63

```

64      procedure Overwrite (Source      : in out Bounded_String;
                             Position   : in Positive;
                             New_Item   : in String;
                             Drop       : in Truncation := Error);

function Delete (Source  : in Bounded_String;
                From     : in Positive;
                Through  : in Natural)
    return Bounded_String;

65

procedure Delete (Source : in out Bounded_String;
                From     : in Positive;
                Through  : in Natural);

66

--<String selector subprograms>

67

function Trim (Source : in Bounded_String;
              Side    : in Trim_End)
    return Bounded_String;
procedure Trim (Source : in out Bounded_String;
              Side     : in Trim_End);

68

function Trim (Source : in Bounded_String;
              Left    : in Maps.Character_Set;
              Right   : in Maps.Character_Set)
    return Bounded_String;

69

procedure Trim (Source : in out Bounded_String;
              Left     : in Maps.Character_Set;
              Right    : in Maps.Character_Set);

70

function Head (Source : in Bounded_String;
              Count   : in Natural;
              Pad     : in Character := Space;
              Drop    : in Truncation := Error)
    return Bounded_String;

```

71

```
procedure Head (Source : in out Bounded_String;
               Count  : in Natural;
               Pad    : in Character := Space;
               Drop   : in Truncation := Error);
```

72

```
function Tail (Source : in Bounded_String;
              Count  : in Natural;
              Pad    : in Character := Space;
              Drop   : in Truncation := Error)
  return Bounded_String;
```

73

```
procedure Tail (Source : in out Bounded_String;
               Count  : in Natural;
               Pad    : in Character := Space;
               Drop   : in Truncation := Error);
```

74

--<String constructor subprograms>

75

```
function "*" (Left  : in Natural;
             Right : in Character)
  return Bounded_String;
```

76

```
function "*" (Left  : in Natural;
             Right : in String)
  return Bounded_String;
```

77

```
function "*" (Left  : in Natural;
             Right : in Bounded_String)
  return Bounded_String;
```

78

```
function Replicate (Count : in Natural;
                  Item  : in Character;
                  Drop   : in Truncation := Error)
  return Bounded_String;
```

79

```
function Replicate (Count : in Natural;  
                   Item   : in String;  
                   Drop   : in Truncation := Error)  
  return Bounded_String;
```

80

```
function Replicate (Count : in Natural;  
                   Item   : in Bounded_String;  
                   Drop   : in Truncation := Error)  
  return Bounded_String;
```

81

```
private  
  ... -- <not specified by the language>  
end Generic_Bounded_Length;
```

82

```
end Ada.Strings.Bounded;
```

83

Null_Bounded_String represents the null string. If an object of type Bounded_String is not otherwise initialized, it will be initialized to the same value as Null_Bounded_String.

84

```
function Length (Source : in Bounded_String) return Length_Range;
```

85

The Length function returns the length of the string represented by Source.

86

```
function To_Bounded_String (Source : in String;  
                            Drop    : in Truncation := Error)  
  return Bounded_String;
```

87

If Source'Length <= Max_Length then this function returns a Bounded_String that represents Source. Otherwise the effect depends on the value of Drop:

88

- If Drop=Left, then the result is a Bounded_String that represents the string comprising the rightmost Max_Length characters of Source.

89

- If Drop=Right, then the result is a Bounded_String that represents the string comprising the leftmost Max_Length characters of Source.

90

- If Drop=Error, then Strings.Length_Error is propagated. ■

91

```
function To_String (Source : in Bounded_String) return String;
```

92

To_String returns the String value with lower bound 1 represented by Source. If B is a Bounded_String, then B = To_Bounded_String(To_String(B)).

92.1/2

```
procedure Set_Bounded_String
(Target : out Bounded_String;
 Source : in String;
 Drop : in Truncation := Error);
```

92.2/2

Equivalent to Target := To_Bounded_String (Source, Drop);

93

Each of the Append functions returns a Bounded_String obtained by concatenating the string or character given or represented by one of the parameters, with the string or character given or represented by the other parameter, and applying To_Bounded_String to the concatenation result string, with Drop as provided to the Append function.

94

Each of the procedures Append(Source, New_Item, Drop) has the same effect as the corresponding assignment Source := Append(Source, New_Item, Drop).

95

Each of the "&" functions has the same effect as the corresponding Append function, with Error as the Drop parameter.

96

```
function Element (Source : in Bounded_String;
                 Index   : in Positive)
  return Character;
```

97

Returns the character at position Index in the string represented by Source; propagates Index_Error if Index > Length(Source).

98

```
procedure Replace_Element (Source : in out Bounded_String;
                          Index   : in Positive;
                          By      : in Character);
```

99

Updates Source such that the character at position Index in the string represented by Source is By; propagates Index_Error if Index > Length(Source).

100

```
function Slice (Source : in Bounded_String;
               Low     : in Positive;
               High    : in Natural)
  return String;
```

101/1

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). The bounds of the returned string are Low and High..

101.1/2

```
function Bounded_Slice
  (Source : in Bounded_String;
   Low    : in Positive;
   High   : in Natural)
```

```
return Bounded_String;
```

101.2/2

Returns the slice at positions Low through High in the string represented by Source as a bounded string; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source).

101.3/2

```
procedure Bounded_Slice
(Source : in      Bounded_String;
 Target : out    Bounded_String;
 Low    : in      Positive;
 High   : in      Natural);
```

101.4/2

Equivalent to Target := Bounded_Slice
(Source, Low, High);

102

Each of the functions "=", "<", ">", "<=", and ">=" returns the same result as the corresponding String operation applied to the String values given or represented by the two parameters.

103

Each of the search subprograms (Index, Index_Non_Blank, Count, Find-Token) has the same effect as the corresponding subprogram in Strings.Fixed applied to the string represented by the Bounded_String parameter.

104

Each of the Translate subprograms, when applied to a Bounded_String, has an analogous effect to the corresponding subprogram in Strings.Fixed. For the Translate function, the translation is applied to the string represented by the Bounded_String parameter, and the result is converted (via To_Bounded_String) to a Bounded_String. For the Translate procedure, the string represented by the Bounded_String parameter after the translation is given by the Translate function for fixed-length strings applied to the string represented by the original value of the parameter.

105/1

Each of the transformation subprograms (Replace_Slice, Insert, Overwrite, Delete), selector subprograms (Trim, Head, Tail), and constructor functions ("*") has an effect based on its corresponding subprogram in Strings.Fixed, and Replicate is based on Fixed.*. In the case of a function, the corresponding fixed-length string subprogram is applied to the string represented by the Bounded_String parameter. To_Bounded_String is applied the result string, with Drop (or Error in the case of Generic_Bounded_Length.*) determining the effect when the string length exceeds Max_Length. In the case of a procedure, the corresponding function in Strings.Bounded.Generic_Bounded_Length is applied, with the result assigned into the Source parameter.

106

Bounded string objects should not be implemented by implicit pointers and dynamic allocation.

15.4.5 A.4.5 Unbounded-Length String Handling

1

The language-defined package `Strings.Unbounded` provides a private type `Unbounded_String` and a set of operations. An object of type `Unbounded_String` represents a `String` whose low bound is 1 and whose length can vary conceptually between 0 and `Natural'Last`. The subprograms for fixed-length string handling are either overloaded directly for `Unbounded_String`, or are modified as needed to reflect the flexibility in length. Since the `Unbounded_String` type is private, relevant constructor and selector operations are provided.

Static Semantics

2

The library package `Strings.Unbounded` has the following declaration:

3

```
with Ada.Strings.Maps;  
package Ada.Strings.Unbounded is  
  pragma Preelaborate(Unbounded);
```

4/2

```
  type Unbounded_String is private;  
  pragma Preelaborable_Initialization(Unbounded_String);
```

5

```
  Null_Unbounded_String : constant Unbounded_String;
```

6

```
  function Length (Source : in Unbounded_String) return Natural;
```

7

```
  type String_Access is access all String;  
  procedure Free (X : in out String_Access);
```

8

```
  --< Conversion, Concatenation, and Selection functions>
```

9

```
  function To_Unbounded_String (Source : in String)  
    return Unbounded_String;
```

10

```
function To_Unbounded_String (Length : in Natural)
    return Unbounded_String;
```

11

```
function To_String (Source : in Unbounded_String) return String;
```

11.1/2

```
procedure Set_Unbounded_String
    (Target : out Unbounded_String;
     Source : in String);
```

12

```
procedure Append (Source : in out Unbounded_String;
                 New_Item : in Unbounded_String);
```

13

```
procedure Append (Source : in out Unbounded_String;
                 New_Item : in String);
```

14

```
procedure Append (Source : in out Unbounded_String;
                 New_Item : in Character);
```

15

```
function "&" (Left, Right : in Unbounded_String)
    return Unbounded_String;
```

16

```
function "&" (Left : in Unbounded_String; Right : in String)
    return Unbounded_String;
```

17

```
function "&" (Left : in String; Right : in Unbounded_String)
    return Unbounded_String;
```

18

```
function "&" (Left : in Unbounded_String; Right : in Character)
    return Unbounded_String;
```

19

```
function "&" (Left : in Character; Right : in Unbounded_String)
```

```

    return Unbounded_String;
20

function Element (Source : in Unbounded_String;
                  Index  : in Positive)
    return Character;
21

procedure Replace_Element (Source : in out Unbounded_String;
                           Index  : in Positive;
                           By      : in Character);
22

function Slice (Source : in Unbounded_String;
               Low     : in Positive;
               High    : in Natural)
    return String;
22.1/2

function Unbounded_Slice
    (Source : in Unbounded_String;
     Low    : in Positive;
     High   : in Natural)
    return Unbounded_String;
22.2/2

procedure Unbounded_Slice
    (Source : in    Unbounded_String;
     Target : out  Unbounded_String;
     Low    : in    Positive;
     High   : in    Natural);
23

function "=" (Left, Right : in Unbounded_String) return Boolean;■
24

function "=" (Left : in Unbounded_String; Right : in String)
    return Boolean;
25

function "=" (Left : in String; Right : in Unbounded_String)
    return Boolean;

```

26

```
function "<" (Left, Right : in Unbounded_String) return Boolean;■
```

27

```
function "<" (Left : in Unbounded_String; Right : in String)  
  return Boolean;
```

28

```
function "<" (Left : in String; Right : in Unbounded_String)  
  return Boolean;
```

29

```
function "<=" (Left, Right : in Unbounded_String) return Boolean;■
```

30

```
function "<=" (Left : in Unbounded_String; Right : in String)  
  return Boolean;
```

31

```
function "<=" (Left : in String; Right : in Unbounded_String)  
  return Boolean;
```

32

```
function ">" (Left, Right : in Unbounded_String) return Boolean;■
```

33

```
function ">" (Left : in Unbounded_String; Right : in String)  
  return Boolean;
```

34

```
function ">" (Left : in String; Right : in Unbounded_String)  
  return Boolean;
```

35

```
function ">=" (Left, Right : in Unbounded_String) return Boolean;■
```

36

```
function ">=" (Left : in Unbounded_String; Right : in String)  
  return Boolean;
```

37

```
function ">=" (Left : in String; Right : in Unbounded_String)
  return Boolean;
```

38

--< Search subprograms>

38.1/2

```
function Index (Source  : in Unbounded_String;
               Pattern  : in String;
               From     : in Positive;
               Going    : in Direction := Forward;
               Mapping  : in Maps.Character_Mapping := Maps.Identity)
  return Natural;
```

38.2/2

```
function Index (Source  : in Unbounded_String;
               Pattern  : in String;
               From     : in Positive;
               Going    : in Direction := Forward;
               Mapping  : in Maps.Character_Mapping_Function)
  return Natural;
```

39

```
function Index (Source  : in Unbounded_String;
               Pattern  : in String;
               Going    : in Direction := Forward;
               Mapping  : in Maps.Character_Mapping
                       := Maps.Identity)
  return Natural;
```

40

```
function Index (Source  : in Unbounded_String;
               Pattern  : in String;
               Going    : in Direction := Forward;
               Mapping  : in Maps.Character_Mapping_Function)
  return Natural;
```

40.1/2

```
function Index (Source  : in Unbounded_String;
               Set      : in Maps.Character_Set;
               From     : in Positive;
               Test     : in Membership := Inside;
```

```

        Going      : in Direction := Forward)
    return Natural;
41

function Index (Source : in Unbounded_String;
               Set      : in Maps.Character_Set;
               Test     : in Membership := Inside;
               Going    : in Direction := Forward) return Natural;█
41.1/2

function Index_Non_Blank (Source : in Unbounded_String;
                        From      : in Positive;
                        Going     : in Direction := Forward)
    return Natural;
42

function Index_Non_Blank (Source : in Unbounded_String;
                        Going    : in Direction := Forward)
    return Natural;
43

function Count (Source      : in Unbounded_String;
               Pattern     : in String;
               Mapping      : in Maps.Character_Mapping
                           := Maps.Identity)
    return Natural;
44

function Count (Source      : in Unbounded_String;
               Pattern     : in String;
               Mapping      : in Maps.Character_Mapping_Function)
    return Natural;
45

function Count (Source      : in Unbounded_String;
               Set          : in Maps.Character_Set)
    return Natural;
46

procedure Find-Token (Source : in Unbounded_String;
                    Set      : in Maps.Character_Set;
                    Test     : in Membership;
                    First    : out Positive;
                    Last     : out Natural);

```

47

```
--< String translation subprograms>
```

48

```
function Translate (Source  : in Unbounded_String;
                   Mapping : in Maps.Character_Mapping)
  return Unbounded_String;
```

49

```
procedure Translate (Source  : in out Unbounded_String;
                   Mapping : in Maps.Character_Mapping);
```

50

```
function Translate (Source  : in Unbounded_String;
                   Mapping : in Maps.Character_Mapping_Function)
  return Unbounded_String;
```

51

```
procedure Translate (Source  : in out Unbounded_String;
                   Mapping : in Maps.Character_Mapping_Function);
```

52

```
--< String transformation subprograms>
```

53

```
function Replace_Slice (Source  : in Unbounded_String;
                       Low      : in Positive;
                       High     : in Natural;
                       By       : in String)
  return Unbounded_String;
```

54

```
procedure Replace_Slice (Source  : in out Unbounded_String;
                       Low      : in Positive;
                       High     : in Natural;
                       By       : in String);
```

55

```
function Insert (Source  : in Unbounded_String;
                Before   : in Positive;
                New_Item : in String)
  return Unbounded_String;
```

56

```
procedure Insert (Source   : in out Unbounded_String;
                 Before   : in Positive;
                 New_Item  : in String);
```

57

```
function Overwrite (Source   : in Unbounded_String;
                   Position  : in Positive;
                   New_Item  : in String)
  return Unbounded_String;
```

58

```
procedure Overwrite (Source   : in out Unbounded_String;
                   Position  : in Positive;
                   New_Item  : in String);
```

59

```
function Delete (Source : in Unbounded_String;
                From    : in Positive;
                Through : in Natural)
  return Unbounded_String;
```

60

```
procedure Delete (Source : in out Unbounded_String;
                 From    : in Positive;
                 Through : in Natural);
```

61

```
function Trim (Source : in Unbounded_String;
              Side   : in Trim_End)
  return Unbounded_String;
```

62

```
procedure Trim (Source : in out Unbounded_String;
               Side   : in Trim_End);
```

63

```
function Trim (Source : in Unbounded_String;
              Left    : in Maps.Character_Set;
              Right   : in Maps.Character_Set)
  return Unbounded_String;
```


64

```
procedure Trim (Source : in out Unbounded_String;  
               Left   : in Maps.Character_Set;  
               Right  : in Maps.Character_Set);
```

65

```
function Head (Source : in Unbounded_String;  
              Count   : in Natural;  
              Pad     : in Character := Space)  
  return Unbounded_String;
```

66

```
procedure Head (Source : in out Unbounded_String;  
               Count   : in Natural;  
               Pad     : in Character := Space);
```

67

```
function Tail (Source : in Unbounded_String;  
              Count   : in Natural;  
              Pad     : in Character := Space)  
  return Unbounded_String;
```

68

```
procedure Tail (Source : in out Unbounded_String;  
               Count   : in Natural;  
               Pad     : in Character := Space);
```

69

```
function "*" (Left   : in Natural;  
             Right  : in Character)  
  return Unbounded_String;
```

70

```
function "*" (Left   : in Natural;  
             Right  : in String)  
  return Unbounded_String;
```

71

```
function "*" (Left   : in Natural;  
             Right  : in Unbounded_String)  
  return Unbounded_String;
```

72

```
private
  ... -- <not specified by the language>
end Ada.Strings.Unbounded;
```

72.1/2

The type `Unbounded_String` needs finalization (see Section 8.6 [7.6], page 295).

73

`Null_Unbounded_String` represents the null `String`. If an object of type `Unbounded_String` is not otherwise initialized, it will be initialized to the same value as `Null_Unbounded_String`.

74

The function `Length` returns the length of the `String` represented by `Source`.

75

The type `String_Access` provides a (non-private) access type for explicit processing of unbounded-length strings. The procedure `Free` performs an unchecked deallocation of an object of type `String_Access`.

76

The function `To_Unbounded_String(Source : in String)` returns an `Unbounded_String` that represents `Source`. The function `To_Unbounded_String(Length : in Natural)` returns an `Unbounded_String` that represents an uninitialized `String` whose length is `Length`.

77

The function `To_String` returns the `String` with lower bound 1 represented by `Source`. `To_String` and `To_Unbounded_String` are related as follows:

78

- If `S` is a `String`, then `To_String(To_Unbounded_String(S)) = S`.

79

- If `U` is an `Unbounded_String`, then `To_Unbounded_String(To_String(U)) = U`.

79.1/2

The procedure `Set_Unbounded_String` sets `Target` to an `Unbounded_String` that represents `Source`.

80

For each of the `Append` procedures, the resulting string represented by the `Source` parameter is given by the concatenation of the original value of `Source` and the value of `New_Item`.

81

Each of the "&" functions returns an `Unbounded_String` obtained by concatenating the string or character given or represented by one of the parameters, with the string or character given or represented by the other parameter, and applying `To_Unbounded_String` to the concatenation result string.

82

The `Element`, `Replace_Element`, and `Slice` subprograms have the same effect as the corresponding bounded-length string subprograms.

82.1/2

The function `Unbounded.Slice` returns the slice at positions `Low` through `High` in the string represented by `Source` as an `Unbounded.String`. The procedure `Unbounded.Slice` sets `Target` to the `Unbounded.String` representing the slice at positions `Low` through `High` in the string represented by `Source`. Both routines propagate `Index_Error` if `Low > Length(Source)+1` or `High > Length(Source)`.

83

Each of the functions `"=`", `"<`", `">`", `"<=`", and `">=`" returns the same result as the corresponding `String` operation applied to the `String` values given or represented by `Left` and `Right`.

84

Each of the search subprograms (`Index`, `Index.Non.Blank`, `Count`, `Find.Token`) has the same effect as the corresponding subprogram in `Strings.Fixed` applied to the string represented by the `Unbounded.String` parameter.

85

The `Translate` function has an analogous effect to the corresponding subprogram in `Strings.Fixed`. The translation is applied to the string represented by the `Unbounded.String` parameter, and the result is converted (via `To.Unbounded.String`) to an `Unbounded.String`.

86

Each of the transformation functions (`Replace.Slice`, `Insert`, `Overwrite`, `Delete`), selector functions (`Trim`, `Head`, `Tail`), and constructor functions (`"*"`) is likewise analogous to its corresponding subprogram in `Strings.Fixed`. For each of the subprograms, the corresponding fixed-length string subprogram is applied to the string represented by the `Unbounded.String` parameter, and `To.Unbounded.String` is applied the result string.

87

For each of the procedures `Translate`, `Replace.Slice`, `Insert`, `Overwrite`, `Delete`, `Trim`, `Head`, and `Tail`, the resulting string represented by the `Source` parameter is given by the corresponding function for fixed-length strings applied to the string represented by `Source`'s original value.

Implementation Requirements

88

No storage associated with an `Unbounded.String` object shall be lost upon assignment or scope exit.

15.4.6 A.4.6 String-Handling Sets and Mappings

1

The language-defined package `Strings.Maps.Constants` declares `Character.Set` and `Character.Mapping` constants corresponding to classification and conversion functions in package `Characters.Handling`.

Static Semantics

2

The library package `Strings.Maps.Constants` has the following declaration:

3/2

```
package Ada.Strings.Maps.Constants is
  pragma Pure(Constants);
```

4

```
Control_Set          : constant Character_Set;
Graphic_Set          : constant Character_Set;
Letter_Set           : constant Character_Set;
Lower_Set            : constant Character_Set;
Upper_Set            : constant Character_Set;
Basic_Set            : constant Character_Set;
Decimal_Digit_Set   : constant Character_Set;
Hexadecimal_Digit_Set : constant Character_Set;
Alphanumeric_Set    : constant Character_Set;
Special_Set          : constant Character_Set;
ISO_646_Set          : constant Character_Set;
```

5

```
Lower_Case_Map       : constant Character_Mapping;
  --<Maps to lower case for letters, else identity>
Upper_Case_Map       : constant Character_Mapping;
  --<Maps to upper case for letters, else identity>
Basic_Map            : constant Character_Mapping;
  --<Maps to basic letter for letters, else identity>
```

6

```
private
  ... -- <not specified by the language>
end Ada.Strings.Maps.Constants;
```

7

Each of these constants represents a correspondingly named set of characters or character mapping in Characters.Handling (see Section 15.3.2 [A.3.2], page 566).

15.4.7 A.4.7 Wide_String Handling

1/2

Facilities for handling strings of Wide_Character elements are found in the packages Strings.Wide_Maps, Strings.Wide_Fixed, Strings.Wide_Bounded, Strings.Wide_Unbounded, and Strings.Wide_Maps.Wide_Constants, and in the functions Strings.Wide_Hash, Strings.Wide_Fixed.Wide_Hash, Strings.Wide_Bounded.Wide_Hash, and Strings.Wide_Unbounded.Wide_Hash. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements.

Static Semantics

2

The package Strings.Wide_Maps has the following declaration.

3

```

package Ada.Strings.Wide_Maps is
  pragma Preelaborate(Wide_Maps);
4/2

  --< Representation for a set of Wide_Character values:>
  type Wide_Character_Set is private;
  pragma Preelaborable_Initialization(Wide_Character_Set);
5

  Null_Set : constant Wide_Character_Set;
6

  type Wide_Character_Range is
    record
      Low   : Wide_Character;
      High  : Wide_Character;
    end record;
  -- <Represents Wide_Character range Low..High>
7

  type Wide_Character_Ranges is array (Positive range <>)
    of Wide_Character_Range;
8

  function To_Set   (Ranges : in Wide_Character_Ranges)
    return Wide_Character_Set;
9

  function To_Set   (Span   : in Wide_Character_Range)
    return Wide_Character_Set;
10

  function To_Ranges (Set    : in Wide_Character_Set)
    return Wide_Character_Ranges;
11

  function "="      (Left, Right : in Wide_Character_Set) return Boolean;█
12

  function "not"    (Right : in Wide_Character_Set)
    return Wide_Character_Set;
  function "and"    (Left, Right : in Wide_Character_Set)
    return Wide_Character_Set;

```

```
function "or" (Left, Right : in Wide_Character_Set)
    return Wide_Character_Set;
function "xor" (Left, Right : in Wide_Character_Set)
    return Wide_Character_Set;
function "-" (Left, Right : in Wide_Character_Set)
    return Wide_Character_Set;
```

13

```
function Is_In (Element : in Wide_Character;
               Set      : in Wide_Character_Set)
    return Boolean;
```

14

```
function Is_Subset (Elements : in Wide_Character_Set;
                  Set       : in Wide_Character_Set)
    return Boolean;
```

15

```
function "<=" (Left  : in Wide_Character_Set;
             Right : in Wide_Character_Set)
    return Boolean renames Is_Subset;
```

16

```
--< Alternative representation for a set of Wide_Character values:>■
subtype Wide_Character_Sequence is Wide_String;
```

17

```
function To_Set (Sequence : in Wide_Character_Sequence)
    return Wide_Character_Set;
```

18

```
function To_Set (Singleton : in Wide_Character)
    return Wide_Character_Set;
```

19

```
function To_Sequence (Set : in Wide_Character_Set)
    return Wide_Character_Sequence;
```

20/2

```
--< Representation for a Wide_Character to Wide_Character mapping:>■
type Wide_Character_Mapping is private;
pragma Preelaborable_Initialization(Wide_Character_Mapping);
```

21

```
function Value (Map      : in Wide_Character_Mapping;  
               Element : in Wide_Character)  
  return Wide_Character;
```

22

```
Identity : constant Wide_Character_Mapping;
```

23

```
function To_Mapping (From, To : in Wide_Character_Sequence)  
  return Wide_Character_Mapping;
```

24

```
function To_Domain (Map : in Wide_Character_Mapping)  
  return Wide_Character_Sequence;
```

25

```
function To_Range (Map : in Wide_Character_Mapping)  
  return Wide_Character_Sequence;
```

26

```
type Wide_Character_Mapping_Function is  
  access function (From : in Wide_Character) return Wide_Character;■
```

27

```
private  
  ... -- <not specified by the language>  
end Ada.Strings.Wide_Maps;
```

28

The context clause for each of the packages `Strings.Wide.Fixed`, `Strings.Wide.Bounded`, and `Strings.Wide.Unbounded` identifies `Strings.Wide.Maps` instead of `Strings.Maps`.

29/2

For each of the packages `Strings.Fixed`, `Strings.Bounded`, `Strings.Unbounded`, and `Strings.Maps.Constants`, and for functions `Strings.Hash`, `Strings.Fixed.Hash`, `Strings.Bounded.Hash`, and `Strings.Unbounded.Hash`, the corresponding wide string package has the same contents except that

30

- `Wide.Space` replaces `Space`

31

- `Wide.Character` replaces `Character`

32

- `Wide_String` replaces `String`

33

- `Wide_Character_Set` replaces `Character_Set`

34

- `Wide_Character_Mapping` replaces `Character_Mapping`

35

- `Wide_Character_Mapping_Function` replaces `Character_Mapping_Function`

36

- `Wide_Maps` replaces `Maps`

37

- `Bounded_Wide_String` replaces `Bounded_String`

38

- `Null_Bounded_Wide_String` replaces `Null_Bounded_String`

39

- `To_Bounded_Wide_String` replaces `To_Bounded_String`

40

- `To_Wide_String` replaces `To_String`

40.1/2

- `Set_Bounded_Wide_String` replaces `Set_Bounded_String`

41

- `Unbounded_Wide_String` replaces `Unbounded_String`

42

- `Null_Unbounded_Wide_String` replaces `Null_Unbounded_String`

43

- `Wide_String_Access` replaces `String_Access`

44

- `To_Unbounded_Wide_String` replaces `To_Unbounded_String`

44.1/2

- `Set_Unbounded_Wide_String` replaces `Set_Unbounded_String`

45

The following additional declaration is present in `Strings.Wide_Maps.Wide_Constants`:

46/2

```
Character_Set : constant Wide_Maps.Wide_Character_Set;  
--<Contains each Wide_Character value WC such that>  
--<Characters.Conversions.Is_Character(WC) is True>
```

46.1/2

Each `Wide_Character_Set` constant in the package `Strings.Wide_Maps.Wide_Constants` contains no values outside the `Character` portion of `Wide_Character`. Similarly, each `Wide_Character_Mapping` constant in this package is the identity mapping when applied to any element outside the `Character` portion of `Wide_Character`.

46.2/2

`Pragma_Pure` is replaced by `pragma_Preelaborate` in `Strings.Wide_Maps.Wide_Constants`.

NOTES

47

12 If a null `Wide_Character_Mapping_Function` is passed to any of the `Wide_String` handling subprograms, `Constraint_Error` is propagated.

48/2

<This paragraph was deleted.>

15.4.8 A.4.8 Wide_Wide_String Handling

1/2

Facilities for handling strings of `Wide_Wide_Character` elements are found in the packages `Strings.Wide_Wide_Maps`, `Strings.Wide_Wide_Fixed`, `Strings.Wide_Wide_Bounded`, `Strings.Wide_Wide_Unbounded`, and `Strings.Wide_Wide_Maps.Wide_Wide_Constants`, and in the functions `Strings.Wide_Wide_Hash`, `Strings.Wide_Wide_Fixed.Wide_Wide_Hash`, `Strings.Wide_Wide_Bounded.Wide_Wide_Hash`, and `Strings.Wide_Wide_Unbounded.Wide_Wide_Hash`. They provide the same string-handling operations as the corresponding packages and functions for strings of `Character` elements.

Static Semantics

2/2

The library package `Strings.Wide_Wide_Maps` has the following declaration.

3/2

```
package Ada.Strings.Wide_Wide_Maps is
  pragma Preelaborate(Wide_Wide_Maps);
```

4/2

```
-- <Representation for a set of Wide_Wide_Character values:>
type Wide_Wide_Character_Set is private;
pragma Preelaborable_Initialization(Wide_Wide_Character_Set);
```

5/2

```
Null_Set : constant Wide_Wide_Character_Set;
```

6/2

```
type Wide_Wide_Character_Range is
  record
    Low   : Wide_Wide_Character;
    High  : Wide_Wide_Character;
  end record;
-- <Represents Wide_Wide_Character range Low..High>
```

7/2

```
type Wide_Wide_Character_Ranges is array (Positive range <>)
  of Wide_Wide_Character_Range;
```

8/2

```
function To_Set (Ranges : in Wide_Wide_Character_Ranges)
  return Wide_Wide_Character_Set;
```

9/2

```
function To_Set (Span : in Wide_Wide_Character_Range)
  return Wide_Wide_Character_Set;
```

10/2

```
function To_Ranges (Set : in Wide_Wide_Character_Set)
  return Wide_Wide_Character_Ranges;
```

11/2

```
function "=" (Left, Right : in Wide_Wide_Character_Set) return Boolean;■
```

12/2

```
function "not" (Right : in Wide_Wide_Character_Set)
  return Wide_Wide_Character_Set;
```

```
function "and" (Left, Right : in Wide_Wide_Character_Set)
    return Wide_Wide_Character_Set;
function "or" (Left, Right : in Wide_Wide_Character_Set)
    return Wide_Wide_Character_Set;
function "xor" (Left, Right : in Wide_Wide_Character_Set)
    return Wide_Wide_Character_Set;
function "-" (Left, Right : in Wide_Wide_Character_Set)
    return Wide_Wide_Character_Set;
```

13/2

```
function Is_In (Element : in Wide_Wide_Character;
               Set      : in Wide_Wide_Character_Set)
    return Boolean;
```

14/2

```
function Is_Subset (Elements : in Wide_Wide_Character_Set;
                   Set       : in Wide_Wide_Character_Set)
    return Boolean;
```

15/2

```
function "<=" (Left  : in Wide_Wide_Character_Set;
             Right : in Wide_Wide_Character_Set)
    return Boolean renames Is_Subset;
```

16/2

```
-- <Alternative representation for a set of Wide_Wide_Character values:>
subtype Wide_Wide_Character_Sequence is Wide_Wide_String;
```

17/2

```
function To_Set (Sequence : in Wide_Wide_Character_Sequence)
    return Wide_Wide_Character_Set;
```

18/2

```
function To_Set (Singleton : in Wide_Wide_Character)
    return Wide_Wide_Character_Set;
```

19/2

```
function To_Sequence (Set : in Wide_Wide_Character_Set)
    return Wide_Wide_Character_Sequence;
```

20/2

```
-- <Representation for a Wide_Wide_Character to Wide_Wide_Character>
-- <mapping:>
```

```

type Wide_Wide_Character_Mapping is private;
pragma Preelaborable_Initialization(Wide_Wide_Character_Mapping);■
21/2

function Value (Map      : in Wide_Wide_Character_Mapping;
               Element  : in Wide_Wide_Character)
               return Wide_Wide_Character;
22/2

Identity : constant Wide_Wide_Character_Mapping;
23/2

function To_Mapping (From, To : in Wide_Wide_Character_Sequence)
               return Wide_Wide_Character_Mapping;
24/2

function To_Domain (Map : in Wide_Wide_Character_Mapping)
               return Wide_Wide_Character_Sequence;
25/2

function To_Range (Map : in Wide_Wide_Character_Mapping)
               return Wide_Wide_Character_Sequence;
26/2

type Wide_Wide_Character_Mapping_Function is
  access function (From : in Wide_Wide_Character)
  return Wide_Wide_Character;
27/2

private
  ... -- <not specified by the language>
end Ada.Strings.Wide_Wide_Maps;

```

28/2

The context clause for each of the packages `Strings.Wide_Wide_Fixed`, `Strings.Wide_Wide_Bounded`, and `Strings.Wide_Wide_Unbounded` identifies `Strings.Wide_Wide_Maps` instead of `Strings.Maps`.

29/2

For each of the packages `Strings.Fixed`, `Strings.Bounded`, `Strings.Unbounded`, and `Strings.Maps.Constants`, and for functions `Strings.Hash`, `Strings.Fixed.Hash`, `Strings.Bounded.Hash`, and `Strings.Unbounded.Hash`, the corresponding wide wide string package or function has the same contents except that

30/2

- `Wide_Wide_Space` replaces `Space`

31/2

- `Wide_Wide_Character` replaces `Character`

32/2

- `Wide_Wide_String` replaces `String`

33/2

- `Wide_Wide_Character_Set` replaces `Character_Set`

34/2

- `Wide_Wide_Character_Mapping` replaces `Character_Mapping`

35/2

- `Wide_Wide_Character_Mapping_Function` replaces `Character_Mapping_Function`

36/2

- `Wide_Wide_Maps` replaces `Maps`

37/2

- `Bounded_Wide_Wide_String` replaces `Bounded_String`

38/2

- `Null_Bounded_Wide_Wide_String` replaces `Null_Bounded_String`

39/2

- `To_Bounded_Wide_Wide_String` replaces `To_Bounded_String`

40/2

- `To_Wide_Wide_String` replaces `To_String`

41/2

- `Set_Bounded_Wide_Wide_String` replaces `Set_Bounded_String`

42/2

- `Unbounded_Wide_Wide_String` replaces `Unbounded_String`

43/2

- `Null_Unbounded_Wide_Wide_String` replaces `Null_Unbounded_String`

44/2

- `Wide_Wide_String_Access` replaces `String_Access`

45/2

- `To_Unbounded_Wide_Wide_String` replaces `To_Unbounded_String`

46/2

- `Set_Unbounded_Wide_Wide_String` replaces `Set_Unbounded_String`

47/2

The following additional declarations are present in `Strings.Wide_Wide_Maps.Wide_Wide_Constants`:

48/2

```
Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
-- <Contains each Wide_Wide_Character value WWC such that>
-- <Characters.Conversions.Is_Character(WWC) is True>
Wide_Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
-- <Contains each Wide_Wide_Character value WWC such that>
-- <Characters.Conversions.Is_Wide_Character(WWC) is True>
```

49/2

Each `Wide_Wide_Character_Set` constant in the package `Strings.Wide_Wide_Maps.Wide_Wide_Constants` contains no values outside the `Character` portion of `Wide_Wide_Character`. Similarly, each `Wide_Wide_Character_Mapping` constant in this package is the identity mapping when applied to any element outside the `Character` portion of `Wide_Wide_Character`.

50/2

`Pragma_Pure` is replaced by `pragma_Preelaborate` in `Strings.Wide_Wide_Maps.Wide_Wide_Constants`.

NOTES

51/2

13 If a null `Wide_Wide_Character_Mapping_Function` is passed to any of the `Wide_Wide_String` handling subprograms, `Constraint_Error` is propagated.

15.4.9 A.4.9 String Hashing

Static Semantics

1/2

The library function `Strings.Hash` has the following declaration:

2/2

```
with Ada.Containers;
function Ada.Strings.Hash (Key : String) return Containers.Hash_Type;
```

```
pragma Pure(Hash);
```

3/2

Returns an implementation-defined value which is a function of the value of Key. If <A> and are strings such that <A> equals , Hash(<A>) equals Hash().

4/2

The library function Strings.Fixed.Hash has the following declaration:

5/2

```
with Ada.Containers, Ada.Strings.Hash;
function Ada.Strings.Fixed.Hash (Key : String) return Containers.Hash_Type
renames Ada.Strings.Hash;
pragma Pure(Hash);
```

6/2

The generic library function Strings.Bounded.Hash has the following declaration:

7/2

```
with Ada.Containers;
generic
  with package Bounded is
    new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Hash (Key : Bounded.Bounded_String)
return Containers.Hash_Type;
pragma Preelaborate(Hash);
```

8/2

Strings.Bounded.Hash is equivalent to the function call Strings.Hash (Bounded.To_String (Key));

9/2

The library function Strings.Unbounded.Hash has the following declaration:

10/2

```
with Ada.Containers;
function Ada.Strings.Unbounded.Hash (Key : Unbounded_String)
return Containers.Hash_Type;
pragma Preelaborate(Hash);
```

11/2

Strings.Unbounded.Hash is equivalent to the function call Strings.Hash (To_String (Key));

12/2

The Hash functions should be good hash functions, returning a wide spread of values for different string values. It should be unlikely for similar strings to return the same value.

15.5 A.5 The Numerics Packages

1

The library package Numerics is the parent of several child units that provide facilities for mathematical computation. One child, the generic package `Generic_Elementary_Functions`, is defined in Section 15.5.1 [A.5.1], page 648, together with nongeneric equivalents; two others, the package `Float_Random` and the generic package `Discrete_Random`, are defined in Section 15.5.2 [A.5.2], page 654. Additional (optional) children are defined in Chapter 21 [Annex G], page 1083, "Chapter 21 [Annex G], page 1083, Numerics".

Static Semantics

2/1

<This paragraph was deleted.>

3/2

```
package Ada.Numerics is
  pragma Pure(Numerics);
  Argument_Error : exception;
  Pi : constant :=
    3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;■
  PI : constant := Pi;
  e : constant :=
    2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;■
end Ada.Numerics;
```

4

The `Argument_Error` exception is raised by a subprogram in a child unit of Numerics to signal that one or more of the actual subprogram parameters are outside the domain of the corresponding mathematical function.

Implementation Permissions

5

The implementation may specify the values of Pi and e to a larger number of significant digits.

15.5.1 A.5.1 Elementary Functions

1

Implementation—defined approximations to the mathematical functions known as the "elementary functions" are provided by the subprograms in `Numerics.Generic_Elementary_Functions`. Nongeneric equivalents of this generic package for each of the predefined floating point types are also provided as children of Numerics.

Static Semantics

2

The generic library package Numerics.Generic_Elementary_Functions has the following declaration:

3

```
generic
  type Float_Type is digits <>;
```

```
package Ada.Numerics.Generic_Elementary_Functions is
  pragma Pure(Generic_Elementary_Functions);
```

4

```
function Sqrt      (X          : Float_Type'Base) return Float_Type'Base;
function Log       (X          : Float_Type'Base) return Float_Type'Base;
function Log       (X, Base   : Float_Type'Base) return Float_Type'Base;
function Exp       (X          : Float_Type'Base) return Float_Type'Base;
function "**"       (Left, Right : Float_Type'Base) return Float_Type'Base;
```

5

```
function Sin       (X          : Float_Type'Base) return Float_Type'Base;
function Sin       (X, Cycle   : Float_Type'Base) return Float_Type'Base;
function Cos       (X          : Float_Type'Base) return Float_Type'Base;
function Cos       (X, Cycle   : Float_Type'Base) return Float_Type'Base;
function Tan       (X          : Float_Type'Base) return Float_Type'Base;
function Tan       (X, Cycle   : Float_Type'Base) return Float_Type'Base;
function Cot       (X          : Float_Type'Base) return Float_Type'Base;
function Cot       (X, Cycle   : Float_Type'Base) return Float_Type'Base;
```

6

```
function Arcsin    (X          : Float_Type'Base) return Float_Type'Base;
function Arcsin    (X, Cycle   : Float_Type'Base) return Float_Type'Base;
function Arccos    (X          : Float_Type'Base) return Float_Type'Base;
function Arccos    (X, Cycle   : Float_Type'Base) return Float_Type'Base;
function Arctan    (Y          : Float_Type'Base;
                   X          : Float_Type'Base := 1.0)
                   return Float_Type'Base;
function Arctan    (Y          : Float_Type'Base;
                   X          : Float_Type'Base := 1.0;
                   Cycle     : Float_Type'Base) return Float_Type'Base;
function Arccot    (X          : Float_Type'Base;
                   Y          : Float_Type'Base := 1.0)
                   return Float_Type'Base;
function Arccot    (X          : Float_Type'Base;
                   Y          : Float_Type'Base := 1.0;
                   Cycle     : Float_Type'Base) return Float_Type'Base;
```

7

```
function Sinh      (X          : Float_Type'Base) return Float_Type'Base;█  
function Cosh      (X          : Float_Type'Base) return Float_Type'Base;█  
function Tanh      (X          : Float_Type'Base) return Float_Type'Base;█  
function Coth      (X          : Float_Type'Base) return Float_Type'Base;█  
function Arcsinh   (X          : Float_Type'Base) return Float_Type'Base;█  
function Arccosh   (X          : Float_Type'Base) return Float_Type'Base;█  
function Arctanh   (X          : Float_Type'Base) return Float_Type'Base;█  
function Arccoth   (X          : Float_Type'Base) return Float_Type'Base;█
```

8

```
end Ada.Numerics.Generic_Elementary_Functions;
```

9/1

The library package `Numerics.Elementary_Functions` is declared pure and defines the same subprograms as `Numerics.Generic_Elementary_Functions`, except that the predefined type `Float` is systematically substituted for `Float_Type'Base` throughout. Nongeneric equivalents of `Numerics.Generic_Elementary_Functions` for each of the other predefined floating point types are defined similarly, with the names `Numerics.Short_Elementary_Functions`, `Numerics.Long_Elementary_Functions`, etc.

10

The functions have their usual mathematical meanings. When the `Base` parameter is specified, the `Log` function computes the logarithm to the given base; otherwise, it computes the natural logarithm. When the `Cycle` parameter is specified, the parameter `X` of the forward trigonometric functions (`Sin`, `Cos`, `Tan`, and `Cot`) and the results of the inverse trigonometric functions (`Arcsin`, `Arccos`, `Arctan`, and `Arccot`) are measured in units such that a full cycle of revolution has the given value; otherwise, they are measured in radians.

11

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

12

- The results of the `Sqrt` and `Arccosh` functions and that of the exponentiation operator are nonnegative.

13

- The result of the `Arcsin` function is in the quadrant containing the point $(1.0, \langle x \rangle)$, where $\langle x \rangle$ is the value of the parameter `X`. This quadrant is I or IV; thus, the range of the `Arcsin` function is approximately $-\text{PI}/2.0$ to $\text{PI}/2.0$ ($-\text{Cycle}/4.0$ to $\text{Cycle}/4.0$, if the parameter `Cycle` is specified).

14

- The result of the `Arccos` function is in the quadrant containing the point $(\langle x \rangle, 1.0)$, where $\langle x \rangle$ is the value of the parameter `X`. This quadrant is I or II; thus, the `Arccos`

function ranges from 0.0 to approximately PI ($\text{Cycle}/2.0$, if the parameter `Cycle` is specified).

15

- The results of the `Arctan` and `Arccot` functions are in the quadrant containing the point $(\langle x \rangle, \langle y \rangle)$, where $\langle x \rangle$ and $\langle y \rangle$ are the values of the parameters `X` and `Y`, respectively. This may be any quadrant (I through IV) when the parameter `X` (resp., `Y`) of `Arctan` (resp., `Arccot`) is specified, but it is restricted to quadrants I and IV (resp., I and II) when that parameter is omitted. Thus, the range when that parameter is specified is approximately $-\text{PI}$ to PI ($-\text{Cycle}/2.0$ to $\text{Cycle}/2.0$, if the parameter `Cycle` is specified); when omitted, the range of `Arctan` (resp., `Arccot`) is that of `Arcsin` (resp., `Arccos`), as given above. When the point $(\langle x \rangle, \langle y \rangle)$ lies on the negative x -axis, the result approximates

16

- PI (resp., $-\text{PI}$) when the sign of the parameter `Y` is positive (resp., negative), if `Float_Type'Signed_Zeros` is `True`;

17

- PI , if `Float_Type'Signed_Zeros` is `False`.

18

(In the case of the inverse trigonometric functions, in which a result lying on or near one of the axes may not be exactly representable, the approximation inherent in computing the result may place it in an adjacent quadrant, close to but on the wrong side of the axis.)

Dynamic Semantics

19

The exception `Numerics.Argument_Error` is raised, signaling a parameter value outside the domain of the corresponding mathematical function, in the following cases:

20

- by any forward or inverse trigonometric function with specified cycle, when the value of the parameter `Cycle` is zero or negative;

21

- by the `Log` function with specified base, when the value of the parameter `Base` is zero, one, or negative;

22

- by the `Sqrt` and `Log` functions, when the value of the parameter `X` is negative;

23

- by the exponentiation operator, when the value of the left operand is negative or when both operands have the value zero;

24

- by the Arcsin, Arccos, and Arctanh functions, when the absolute value of the parameter X exceeds one;

25

- by the Arctan and Arccot functions, when the parameters X and Y both have the value zero;

26

- by the Arccosh function, when the value of the parameter X is less than one; and

27

- by the Arccoth function, when the absolute value of the parameter X is less than one.

28

The exception `Constraint_Error` is raised, signaling a pole of the mathematical function (analogous to dividing by zero), in the following cases, provided that `Float_Type'Machine_Overflows` is `True`:

29

- by the Log, Cot, and Coth functions, when the value of the parameter X is zero;

30

- by the exponentiation operator, when the value of the left operand is zero and the value of the exponent is negative;

31

- by the Tan function with specified cycle, when the value of the parameter X is an odd multiple of the quarter cycle;

32

- by the Cot function with specified cycle, when the value of the parameter X is zero or a multiple of the half cycle; and

33

- by the Arctanh and Arccoth functions, when the absolute value of the parameter X is one.

34

`Constraint_Error` can also be raised when a finite result overflows (see Section 21.2.4 [G.2.4], page 1113); this may occur for parameter values sufficiently `<near>` poles, and, in the case

of some of the functions, for parameter values with sufficiently large magnitudes. When `Float_Type'Machine_Overflows` is `False`, the result at poles is unspecified.

35

When one parameter of a function with multiple parameters represents a pole and another is outside the function's domain, the latter takes precedence (i.e., `Numerics.Argument_Error` is raised).

Implementation Requirements

36

In the implementation of `Numerics.Generic_Elementary_Functions`, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype `Float_Type`.

37

In the following cases, evaluation of an elementary function shall yield the <prescribed result>, provided that the preceding rules do not call for an exception to be raised:

38

- When the parameter `X` has the value zero, the `Sqrt`, `Sin`, `Arcsin`, `Tan`, `Sinh`, `Arcsinh`, `Tanh`, and `Arctanh` functions yield a result of zero, and the `Exp`, `Cos`, and `Cosh` functions yield a result of one.

39

- When the parameter `X` has the value one, the `Sqrt` function yields a result of one, and the `Log`, `Arccos`, and `Arccosh` functions yield a result of zero.

40

- When the parameter `Y` has the value zero and the parameter `X` has a positive value, the `Arctan` and `Arccot` functions yield a result of zero.

41

- The results of the `Sin`, `Cos`, `Tan`, and `Cot` functions with specified cycle are exact when the mathematical result is zero; those of the first two are also exact when the mathematical result is ± 1.0 .

42

- Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand. Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero.

43

Other accuracy requirements for the elementary functions, which apply only in implementations conforming to the `Numerics Annex`, and then only in the "strict" mode defined there (see Section 21.2 [G.2], page 1103), are given in Section 21.2.4 [G.2.4], page 1113.

44

When `Float_Type'Signed_Zeros` is `True`, the sign of a zero result shall be as follows:

45

- A prescribed zero result delivered <at the origin> by one of the odd functions (Sin, Arcsin, Sinh, Arcsinh, Tan, Arctan or Arccot as a function of Y when X is fixed and positive, Tanh, and Arctanh) has the sign of the parameter X (Y, in the case of Arctan or Arccot).

46

- A prescribed zero result delivered by one of the odd functions <away from the origin>, or by some other elementary function, has an implementation–defined sign.

47

- A zero result that is not a prescribed result (i.e., one that results from rounding or underflow) has the correct mathematical sign.

Implementation Permissions

48

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

15.5.2 A.5.2 Random Number Generation

1

Facilities for the generation of pseudo–random floating point numbers are provided in the package Numerics.Float_Random; the generic package Numerics.Discrete_Random provides similar facilities for the generation of pseudo–random integers and pseudo–random values of enumeration types. For brevity, pseudo–random values of any of these types are called <random numbers>.

2

Some of the facilities provided are basic to all applications of random numbers. These include a limited private type each of whose objects serves as the generator of a (possibly distinct) sequence of random numbers; a function to obtain the "next" random number from a given sequence of random numbers (that is, from its generator); and subprograms to initialize or reinitialize a given generator to a time–dependent state or a state denoted by a single integer.

3

Other facilities are provided specifically for advanced applications. These include subprograms to save and restore the state of a given generator; a private type whose objects can be used to hold the saved state of a generator; and subprograms to obtain a string representation of a given generator state, or, given such a string representation, the corresponding state.

Static Semantics

4

The library package Numerics.Float_Random has the following declaration:

5

```

6      package Ada.Numerics.Float_Random is
7
8          -- <Basic facilities>
9
10         type Generator is limited private;
11
12         subtype Uniformly_Distributed is Float range 0.0 .. 1.0;
13         function Random (Gen : Generator) return Uniformly_Distributed;
14
15         procedure Reset (Gen      : in Generator;
16                         Initiator : in Integer);
17         procedure Reset (Gen      : in Generator);
18
19         -- <Advanced facilities>
20
21         type State is private;
22
23         procedure Save (Gen      : in Generator;
24                       To_State  : out State);
25         procedure Reset (Gen      : in Generator;
26                        From_State : in State);
27
28         Max_Image_Width : constant := <implementation-defined integer value>;
29
30         function Image (Of_State : State) return String;
31         function Value (Coded_State : String) return State;
32
33         private
34             ... -- <not specified by the language>
35         end Ada.Numerics.Float_Random;

```

15.1/2

The type Generator needs finalization (see Section 8.6 [7.6], page 295).

16

The generic library package Numerics.Discrete_Random has the following declaration:

17

```
generic
  type Result_Subtype is (<>);
package Ada.Numerics.Discrete_Random is
```

18

```
-- <Basic facilities>
```

19

```
type Generator is limited private;
```

20

```
function Random (Gen : Generator) return Result_Subtype;
```

21

```
procedure Reset (Gen      : in Generator;
                 Initiator : in Integer);
procedure Reset (Gen      : in Generator);
```

22

```
-- <Advanced facilities>
```

23

```
type State is private;
```

24

```
procedure Save (Gen      : in Generator;
               To_State  : out State);
procedure Reset (Gen      : in Generator;
               From_State : in State);
```

25

```
Max_Image_Width : constant := <implementation-defined integer value>;■
```

26

```
function Image (Of_State : State) return String;
function Value (Coded_State : String) return State;
```


27

```
private
  ... -- <not specified by the language>
end Ada.Numerics.Discrete_Random;
```

27.1/2

The type `Generator` needs finalization (see Section 8.6 [7.6], page 295) in every instantiation of `Numerics.Discrete_Random`.

28

An object of the limited private type `Generator` is associated with a sequence of random numbers. Each generator has a hidden (internal) state, which the operations on generators use to determine the position in the associated sequence. All generators are implicitly initialized to an unspecified state that does not vary from one program execution to another; they may also be explicitly initialized, or reinitialized, to a time-dependent state, to a previously saved state, or to a state uniquely denoted by an integer value.

29

An object of the private type `State` can be used to hold the internal state of a generator. Such objects are only needed if the application is designed to save and restore generator states or to examine or manufacture them.

30

The operations on generators affect the state and therefore the future values of the associated sequence. The semantics of the operations on generators and states are defined below.

31

```
function Random (Gen : Generator) return Uniformly_Distributed;
function Random (Gen : Generator) return Result_Subtype;
```

32

Obtains the "next" random number from the given generator, relative to its current state, according to an implementation-defined algorithm. The result of the function in `Numerics.Float_Random` is delivered as a value of the subtype `Uniformly_Distributed`, which is a subtype of the predefined type `Float` having a range of 0.0 .. 1.0. The result of the function in an instantiation of `Numerics.Discrete_Random` is delivered as a value of the generic formal subtype `Result_Subtype`.

33

```
procedure Reset (Gen          : in Generator;
                Initiator : in Integer);
```

```
procedure Reset (Gen      : in Generator);
```

34

Sets the state of the specified generator to one that is an unspecified function of the value of the parameter Initiator (or to a time-dependent state, if only a generator parameter is specified). The latter form of the procedure is known as the <time-dependent Reset procedure>.

35

```
procedure Save (Gen      : in Generator;
               To_State  : out State);
procedure Reset (Gen      : in Generator;
                From_State : in State);
```

36

Save obtains the current state of a generator. Reset gives a generator the specified state. A generator that is reset to a state previously obtained by invoking Save is restored to the state it had when Save was invoked.

37

```
function Image (Of_State  : State) return String;
function Value (Coded_State : String) return State;
```

38

Image provides a representation of a state coded (in an implementation-defined way) as a string whose length is bounded by the value of Max_Image_Width. Value is the inverse of Image: Value(Image(S)) = S for each state S that can be obtained from a generator by invoking Save.

Dynamic Semantics

39

Instantiation of Numerics.Discrete_Random with a subtype having a null range raises Constraint_Error.

40/1

<This paragraph was deleted.>

Bounded (Run-Time) Errors

40.1/1

It is a bounded error to invoke `Value` with a string that is not the image of any generator state. If the error is detected, `Constraint_Error` or `Program_Error` is raised. Otherwise, a call to `Reset` with the resulting state will produce a generator such that calls to `Random` with this generator will produce a sequence of values of the appropriate subtype, but which might not be random in character. That is, the sequence of values might not fulfill the implementation requirements of this subclause.

Implementation Requirements

41

A sufficiently long sequence of random numbers obtained by successive calls to `Random` is approximately uniformly distributed over the range of the result subtype.

42

The `Random` function in an instantiation of `Numerics.Discrete.Random` is guaranteed to yield each value in its result subtype in a finite number of calls, provided that the number of such values does not exceed 2^{15} .

43

Other performance requirements for the random number generator, which apply only in implementations conforming to the Numerics Annex, and then only in the "strict" mode defined there (see Section 21.2 [G.2], page 1103), are given in Section 21.2.5 [G.2.5], page 1115.

Documentation Requirements

44

No one algorithm for random number generation is best for all applications. To enable the user to determine the suitability of the random number generators for the intended application, the implementation shall describe the algorithm used and shall give its period, if known exactly, or a lower bound on the period, if the exact period is unknown. Periods that are so long that the periodicity is unobservable in practice can be described in such terms, without giving a numerical bound.

45

The implementation also shall document the minimum time interval between calls to the time-dependent `Reset` procedure that are guaranteed to initiate different sequences, and it shall document the nature of the strings that `Value` will accept without raising `Constraint_Error`.

Implementation Advice

46

Any storage associated with an object of type `Generator` should be reclaimed on exit from the scope of the object.

47

If the generator period is sufficiently long in relation to the number of distinct initiator values, then each possible value of `Initiator` passed to `Reset` should initiate a sequence of random numbers that does not, in a practical sense, overlap the sequence initiated by any other value. If this is not possible, then the mapping between initiator values and generator states should be a rapidly varying function of the initiator value.

NOTES

48

14 If two or more tasks are to share the same generator, then the tasks have to synchronize their access to the generator as for any shared variable (see Section 10.10 [9.10], page 389).

49

15 Within a given implementation, a repeatable random number sequence can be obtained by relying on the implicit initialization of generators or by explicitly initializing a generator with a repeatable initiator value. Different sequences of random numbers can be obtained from a given generator in different program executions by explicitly initializing the generator to a time-dependent state.

50

16 A given implementation of the `Random` function in `Numerics.Float_Random` may or may not be capable of delivering the values 0.0 or 1.0. Portable applications should assume that these values, or values sufficiently close to them to behave indistinguishably from them, can occur. If a sequence of random integers from some fixed range is needed, the application should use the `Random` function in an appropriate instantiation of `Numerics.Discrete_Random`, rather than transforming the result of the `Random` function in `Numerics.Float_Random`. However, some applications with unusual requirements, such as for a sequence of random integers each drawn from a different range, will find it more convenient to transform the result of the floating point `Random` function. For $M \geq 1$, the expression

51

$$\text{Integer}(\text{Float}(M) * \text{Random}(G)) \bmod M$$

52

transforms the result of `Random(G)` to an integer uniformly distributed over the range 0 .. $M-1$; it is valid even if `Random` delivers 0.0 or 1.0. Each value of the result range is possible, provided that M is not too large. Exponentially distributed (floating point) random numbers with mean and standard deviation 1.0 can be obtained by the transformation

53/2

$$-\text{Log}(\text{Random}(G) + \text{Float}'\text{Model_Small})$$

54

where `Log` comes from `Numerics.Elementary_Functions` (see Section 15.5.1 [A.5.1], page 648); in this expression, the addition of

Float'Model.Small avoids the exception that would be raised were Log to be given the value zero, without affecting the result (in most implementations) when Random returns a nonzero value.

Examples

55

<Example of a program that plays a simulated dice game:>

56

```
with Ada.Numerics.Discrete_Random;
procedure Dice_Game is
  subtype Die is Integer range 1 .. 6;
  subtype Dice is Integer range 2*Die'First .. 2*Die'Last;
  package Random_Die is new Ada.Numerics.Discrete_Random (Die);
  use Random_Die;
  G : Generator;
  D : Dice;
begin
  Reset (G); -- <Start the generator in a unique state in each run>
  loop
    -- <Roll a pair of dice; sum and process the results>
    D := Random(G) + Random(G);
    ...
  end loop;
end Dice_Game;
```

57

<Example of a program that simulates coin tosses:>

58

```
with Ada.Numerics.Discrete_Random;
procedure Flip_A_Coin is
  type Coin is (Heads, Tails);
  package Random_Coin is new Ada.Numerics.Discrete_Random (Coin);
  use Random_Coin;
  G : Generator;
begin
  Reset (G); -- <Start the generator in a unique state in each run>
  loop
    -- <Toss a coin and process the result>
    case Random(G) is
      when Heads =>
        ...
      when Tails =>
        ...
    end case;
  ...
end Flip_A_Coin;
```

```

        end loop;
    end Flip_A_Coin;

```

59

<Example of a parallel simulation of a physical system, with a separate generator of event probabilities in each task:>

60

```

with Ada.Numerics.Float_Random;
procedure Parallel_Simulation is
    use Ada.Numerics.Float_Random;
    task type Worker is
        entry Initialize_Generator (Initiator : in Integer);
        ...
    end Worker;
    W : array (1 .. 10) of Worker;
    task body Worker is
        G : Generator;
        Probability_Of_Event : Uniformly_Distributed;
    begin
        accept Initialize_Generator (Initiator : in Integer) do
            Reset (G, Initiator);
        end Initialize_Generator;
        loop
            ...
            Probability_Of_Event := Random(G);
            ...
        end loop;
    end Worker;
begin
    -- <Initialize the generators in the Worker tasks to different states>
    for I in W'Range loop
        W(I).Initialize_Generator (I);
    end loop;
    ... -- <Wait for the Worker tasks to terminate>
end Parallel_Simulation;

```

NOTES

61

17 <Notes on the last example:> Although each Worker task initializes its generator to a different state, those states will be the same in every execution of the program. The generator states can be initialized uniquely in each program execution by instantiating `Ada.Numerics.Discrete_Random` for the type `Integer` in the main procedure, resetting the generator obtained from that instance to a time-dependent state, and then using random integers obtained from that generator to initialize the generators in each Worker task.

15.5.3 A.5.3 Attributes of Floating Point Types

Static Semantics

1

The following <representation-oriented attributes> are defined for every subtype S of a floating point type <T>.

2

S'Machine_Radix

Yields the radix of the hardware representation of the type <T>. The value of this attribute is of the type <universal_integer>.

3

The values of other representation-oriented attributes of a floating point subtype, and of the "primitive function" attributes of a floating point subtype described later, are defined in terms of a particular representation of nonzero values called the <canonical form>. The canonical form (for the type <T>) is the form

$\pm \text{<mantissa>} \cdot \text{<T>'Machine_Radix}^{\text{<exponent>}}$

where

4

- <mantissa> is a fraction in the number base <T>'Machine_Radix, the first digit of which is nonzero, and

5

- <exponent> is an integer.

6

S'Machine_Mantissa

Yields the largest value of <p> such that every value expressible in the canonical form (for the type <T>), having a <p>-digit <mantissa> and an <exponent> between <T>'Machine_Emin and <T>'Machine_Emax, is a machine number (see Section 4.5.7 [3.5.7], page 103) of

the type `<T>`. This attribute yields a value of the type `<universal_integer>`.

7
S'Machine_Emin

Yields the smallest (most negative) value of `<exponent>` such that every value expressible in the canonical form (for the type `<T>`), having a `<mantissa>` of `<T>'Machine_Mantissa` digits, is a machine number (see Section 4.5.7 [3.5.7], page 103) of the type `<T>`. This attribute yields a value of the type `<universal_integer>`.

8
S'Machine_Emax

Yields the largest (most positive) value of `<exponent>` such that every value expressible in the canonical form (for the type `<T>`), having a `<mantissa>` of `<T>'Machine_Mantissa` digits, is a machine number (see Section 4.5.7 [3.5.7], page 103) of the type `<T>`. This attribute yields a value of the type `<universal_integer>`.

9
S'Denorm

Yields the value True if every value

expressible in the form

$$\pm \langle \text{mantissa} \rangle \cdot \langle T \rangle^{\text{Machine_Radix} - \langle T \rangle^{\text{Machine_Emin}}}$$

where $\langle \text{mantissa} \rangle$ is a nonzero $\langle T \rangle^{\text{Machine_Mantissa-digit}}$ fraction in the number base $\langle T \rangle^{\text{Machine_Radix}}$, the first digit of which is zero, is a machine number (see Section 4.5.7 [3.5.7], page 103) of the type $\langle T \rangle$; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

10

The values described by the formula in the definition of S^{Denorm} are called $\langle \text{denormalized numbers} \rangle$. A nonzero machine number that is not a denormalized number is a $\langle \text{normalized number} \rangle$. A normalized number $\langle x \rangle$ of a given type $\langle T \rangle$ is said to be $\langle \text{represented in canonical form} \rangle$ when it is expressed in the canonical form (for the type $\langle T \rangle$) with a $\langle \text{mantissa} \rangle$ having $\langle T \rangle^{\text{Machine_Mantissa}}$ digits; the resulting form is the $\langle \text{canonical-form representation} \rangle$ of $\langle x \rangle$.

11

$S^{\text{Machine_Rounds}}$

Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type $\langle T \rangle$; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

12

$S^{\text{Machine_Overflows}}$

Yields the value True if overflow and

divide-by-zero
are detected and
reported by raising
Constraint_Error
for every predefined
operation that yields
a result of the type
<T>; yields the value
False otherwise. The
value of this attribute
is of the predefined
type Boolean.

13

S'Signed_Zeros

Yields the value
True if the hardware
representation for
the type <T> has
the capability of
representing both
positively and
negatively signed
zeros, these being
generated and used
by the predefined
operations of the
type <T> as specified
in IEC 559:1989;
yields the value False
otherwise. The value
of this attribute is of
the predefined type
Boolean.

14

For every value <x> of a floating point type <T>, the <normalized exponent> of <x> is defined as follows:

15

- the normalized exponent of zero is (by convention) zero;

16

- for nonzero <x>, the normalized exponent of <x> is the unique integer <k> such that $\langle T \rangle \text{'Machine_Radix} \langle k \rangle - 1 \leq | \langle x \rangle | < \langle T \rangle \text{'Machine_Radix} \langle k \rangle$.

17

The following <primitive function attributes> are defined for any subtype S of a floating point type <T>.

18

S'Exponent

S'Exponent denotes a function with the following specification:

19

```
function S'Exponent (<X> : <T>)
    return <universal_integer>
```

20

The function yields the normalized exponent of <X>.

21

S'Fraction

S'Fraction denotes a function with the following specification:

22

```
function S'Fraction (<X> : <T>)
    return <T>
```

23

The function yields the value $\langle X \rangle \cdot \langle T \rangle \text{'Machine_Radix}^{-\langle k \rangle}$, where $\langle k \rangle$ is the normalized exponent of $\langle X \rangle$. A zero result, which can only occur when $\langle X \rangle$ is zero, has the sign of $\langle X \rangle$.

24

S'Compose

S'Compose denotes a function with the following specification:

25

```

function S'Compose (<Fraction> : <T>;
                  <Exponent> : <universal_integer>)
    return <T>

```

26

Let $\langle v \rangle$ be the value $\langle \text{Fraction} \rangle \cdot \langle \text{T} \rangle \text{Machine_Radix}^{\langle \text{Exponent} \rangle - \langle k \rangle}$, where $\langle k \rangle$ is the normalized exponent of $\langle \text{Fraction} \rangle$. If $\langle v \rangle$ is a machine number of the type $\langle \text{T} \rangle$, or if $|\langle v \rangle| \geq \langle \text{T} \rangle \text{Model_Small}$, the function yields $\langle v \rangle$; otherwise, it yields either one of the machine numbers of the type $\langle \text{T} \rangle$ adjacent to $\langle v \rangle$. `Constraint_Error` is optionally raised if $\langle v \rangle$ is outside the base range of S . A zero result has the sign of $\langle \text{Fraction} \rangle$ when `S'Signed_Zeros` is `True`.

27

`S'Scaling`

`S'Scaling` denotes a function with the following specification:

28

```

function S'Scaling (<X> : <T>;
                  <Adjustment> : <universal_integer>)
    return <T>

```

29

Let $\langle v \rangle$ be the value $\langle X \rangle \cdot \langle T \rangle$ 'Machine_Radix $\langle \text{Adjustment} \rangle$. If $\langle v \rangle$ is a machine number of the type $\langle T \rangle$, or if $|\langle v \rangle| \geq \langle T \rangle$ 'Model_Small, the function yields $\langle v \rangle$; otherwise, it yields either one of the machine numbers of the type $\langle T \rangle$ adjacent to $\langle v \rangle$. Constraint_Error is optionally raised if $\langle v \rangle$ is outside the base range of S. A zero result has the sign of $\langle X \rangle$ when S'Signed_Zeros is True.

30
S'Floor

S'Floor denotes a function with the following specification:

31

```
function S'Floor (<X> : <T>)
  return <T>
```

32

The function yields the value $\text{floor}(\langle X \rangle)$, i.e., the largest (most positive) integral value less than or equal to $\langle X \rangle$. When $\langle X \rangle$ is zero, the result has the sign of $\langle X \rangle$; a zero result otherwise has a positive sign.

33
S'Ceiling

S'Ceiling denotes a function with the following specification:

34

```
function S'Ceiling (<X> : <T>)  
  return <T>
```

35

The function yields the value `ceiling(<X>)`, i.e., the smallest (most negative) integral value greater than or equal to `<X>`. When `<X>` is zero, the result has the sign of `<X>`; a zero result otherwise has a negative sign when `S'Signed_Zeros` is `True`.

36
S'Rounding

S'Rounding denotes a function with the following specification:

37

```
function S'Rounding (<X> : <T>)  
  return <T>
```

38

The function yields the integral value nearest to `<X>`, rounding away from zero if `<X>` lies exactly halfway between two integers. A zero result has the

sign of $\langle X \rangle$ when
S'Signed_Zeros is
True.

39

S'Unbiased_Rounding

S'Unbiased_Rounding
denotes a function
with the following
specification:

40

```
function S'Unbiased_Rounding (<X> : <T>)
return <T>
```

41

The function yields
the integral value
nearest to $\langle X \rangle$,
rounding toward the
even integer if $\langle X \rangle$
lies exactly halfway
between two integers.
A zero result has the
sign of $\langle X \rangle$ when
S'Signed_Zeros is
True.

41.1/2

S'Machine_Rounding

S'Machine_Rounding
denotes a function
with the following
specification:

41.2/2

```
function S'Machine_Rounding (<X> : <T>)
return <T>
```

41.3/2

The function yields
the integral value
nearest to $\langle X \rangle$. If
 $\langle X \rangle$ lies exactly
halfway between
two integers, one

of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of <X> when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor.

42
S'Truncation

S'Truncation denotes a function with the following specification:

43

```
function S'Truncation (<X> : <T>)  
    return <T>
```

44

The function yields the value `ceiling(<X>)` when <X> is negative, and `floor(<X>)` otherwise. A zero result has the sign of <X> when S'Signed_Zeros is True.

45
S'Remainder

S'Remainder denotes a function with the following specification:

46

```
function S'Remainder (<X>, <Y> : <T>)  
    return <T>
```


47

For nonzero $\langle Y \rangle$, let $\langle v \rangle$ be the value $\langle X \rangle - \langle n \rangle \cdot \langle Y \rangle$, where $\langle n \rangle$ is the integer nearest to the exact value of $\langle X \rangle / \langle Y \rangle$; if $|\langle n \rangle - \langle X \rangle / \langle Y \rangle| = 1/2$, then $\langle n \rangle$ is chosen to be even. If $\langle v \rangle$ is a machine number of the type $\langle T \rangle$, the function yields $\langle v \rangle$; otherwise, it yields zero. `Constraint_Error` is raised if $\langle Y \rangle$ is zero. A zero result has the sign of $\langle X \rangle$ when `S'Signed_Zeros` is `True`.

48
S'Adjacent

S'Adjacent denotes a function with the following specification:

49

```
function S'Adjacent (<X>, <Towards> : <T>)
return <T>
```

50

If $\langle \text{Towards} \rangle = \langle X \rangle$, the function yields $\langle X \rangle$; otherwise, it yields the machine number of the type $\langle T \rangle$ adjacent to $\langle X \rangle$ in the direction of $\langle \text{Towards} \rangle$, if that machine number exists. If the result would be outside the base range of

S, Constraint_Error is raised. When <T>'Signed_Zeros is True, a zero result has the sign of <X>. When <Towards> is zero, its sign has no bearing on the result.

51
S'Copy_Sign

S'Copy_Sign denotes a function with the following specification:

52

```
function S'Copy_Sign (<Value>, <Sign> : <T>)■  
    return <T>
```

53

If the value of <Value> is nonzero, the function yields a result whose magnitude is that of <Value> and whose sign is that of <Sign>; otherwise, it yields the value zero. Constraint_Error is optionally raised if the result is outside the base range of S. A zero result has the sign of <Sign> when S'Signed_Zeros is True.

54
S'Leading_Part

S'Leading_Part denotes a function with the following specification:

55

```

function S'Leading_Part (<X> : <T>;
                        <Radix_Digits> : <universal_
return <T>

```

56

Let $\langle v \rangle$ be the value $\langle T \rangle$ 'Machine_Radix $\langle k \rangle - \langle \text{Radix_Digits} \rangle$, where $\langle k \rangle$ is the normalized exponent of $\langle X \rangle$. The function yields the value

57

- $\text{floor}(\langle X \rangle / \langle v \rangle)$
 $\cdot \langle v \rangle$, when $\langle X \rangle$ is non-negative and $\langle \text{Radix_Digits} \rangle$ is positive;

58

- $\text{ceiling}(\langle X \rangle / \langle v \rangle)$
 $\cdot \langle v \rangle$, when $\langle X \rangle$ is negative and $\langle \text{Radix_Digits} \rangle$ is positive.

59

Constraint_Error is raised when $\langle \text{Radix_Digits} \rangle$ is zero or negative. A zero result, which can only occur when $\langle X \rangle$ is zero, has the sign of $\langle X \rangle$.

60

S'Machine

S'Machine denotes a function with the following specification:

61

62

```
function S'Machine (<X> : <T>)
    return <T>
```

If <X> is a machine number of the type <T>, the function yields <X>; otherwise, it yields the value obtained by rounding or truncating <X> to either one of the adjacent machine numbers of the type <T>. Constraint_Error is raised if rounding or truncating <X> to the precision of the machine numbers results in a value outside the base range of S. A zero result has the sign of <X> when S'Signed_Zeros is True.

63

The following <model-oriented attributes> are defined for any subtype S of a floating point type <T>.

64

S'Model_Mantissa

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to $\text{ceiling}(\langle d \rangle \cdot \log(10) / \log(\langle T \rangle \text{'Machine_Radix})) + 1$, where <d> is the requested decimal precision of <T>, and less than or equal to the value of

<T>'Machine_Mantissa.
See Section 21.2.2
[G.2.2], page 1105, for
further requirements
that apply to
implementations
supporting the
Numerics Annex.
The value of this at-
tribute is of the type
<universal_integer>.

65

S'Model_Emin

If the Numerics
Annex is not
supported, this
attribute yields an
implementation
defined value that
is greater than or
equal to the value of
<T>'Machine_Emin.
See Section 21.2.2
[G.2.2], page 1105, for
further requirements
that apply to
implementations
supporting the
Numerics Annex.
The value of this at-
tribute is of the type
<universal_integer>.

66

S'Model_Epsilon

Yields the value
<T>'Machine_Radix1
– <T>'Model_Mantissa.
The value of this
attribute is of the
type <universal_real>.

67

S'Model_Small

Yields the value
<T>'Machine_Radix<T>'Model_Emin
– 1. The value of

this attribute is of the type `<universal_real>`.

68
S'Model

S'Model denotes a function with the following specification:

69

```
function S'Model (<X> : <T>)
  return <T>
```

70

If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see Section 21.2.2 [G.2.2], page 1105, for the definition that applies to implementations supporting the Numerics Annex.

71
S'Safe.First

Yields the lower bound of the safe range (see Section 4.5.7 [3.5.7], page 103) of the type `<T>`. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see Section 21.2.2 [G.2.2], page 1105, for the definition that applies to implementations supporting the

Numerics Annex.
The value of this attribute is of the type `<universal_real>`.

72

`S'Safe_Last`

Yields the upper bound of the safe range (see Section 4.5.7 [3.5.7], page 103) of the type `<T>`. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see Section 21.2.2 [G.2.2], page 1105, for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type `<universal_real>`.

15.5.4 A.5.4 Attributes of Fixed Point Types

Static Semantics

1

The following `<representation-oriented>` attributes are defined for every subtype `S` of a fixed point type `<T>`.

2

`S'Machine_Radix`

Yields the radix of the hardware representation of the type `<T>`. The value of this attribute is of the type `<universal_integer>`.

3

`S'Machine_Rounds`

Yields the value `True` if rounding is

performed on inexact results of every predefined operation that yields a result of the type `<T>`; yields the value `False` otherwise. The value of this attribute is of the predefined type `Boolean`.

4

`S'Machine_Overflows`

Yields the value `True` if overflow and divide-by-zero are detected and reported by raising `Constraint_Error` for every predefined operation that yields a result of the type `<T>`; yields the value `False` otherwise. The value of this attribute is of the predefined type `Boolean`.

15.6 A.6 Input-Output

1/2

Input-output is provided through language-defined packages, each of which is a child of the root package `Ada`. The generic packages `Sequential_IO` and `Direct_IO` define input-output operations applicable to files containing elements of a given type. The generic package `Storage_IO` supports reading from and writing to an in-memory buffer. Additional operations for text input-output are supplied in the packages `Text_IO`, `Wide_Text_IO`, and `Wide_Wide_Text_IO`. Heterogeneous input-output is provided through the child packages `Streams.Stream_IO` and `Text_IO.Text_Streams` (see also Section 14.13 [13.13], page 538). The package `IO_Exceptions` defines the exceptions needed by the predefined input-output packages.

15.7 A.7 External Files and File Objects

Static Semantics

1

Values input from the external environment of the program, or output to the external environment, are considered to occupy `<external files>`. An external file can be anything external to the program that can produce a value to be read or receive a value to be written.

An external file is identified by a string (the <name>). A second string (the <form>) gives further system-dependent characteristics that may be associated with the file, such as the physical organization or access rights. The conventions governing the interpretation of such strings shall be documented.

2

Input and output operations are expressed as operations on objects of some <file type>, rather than directly in terms of the external files. In the remainder of this section, the term <file> is always used to refer to a file object; the term <external file> is used otherwise.

3

Input-output for sequential files of values of a single element type is defined by means of the generic package `Sequential_IO`. In order to define sequential input-output for a given element type, an instantiation of this generic unit, with the given type as actual parameter, has to be declared. The resulting package contains the declaration of a file type (called `File_Type`) for files of such elements, as well as the operations applicable to these files, such as the `Open`, `Read`, and `Write` procedures.

4/2

Input-output for direct access files is likewise defined by a generic package called `Direct_IO`. Input-output in human-readable form is defined by the (nongeneric) packages `Text_IO` for `Character` and `String` data, `Wide_Text_IO` for `Wide_Character` and `Wide_String` data, and `Wide_Wide_Text_IO` for `Wide_Wide_Character` and `Wide_Wide_String` data. Input-output for files containing streams of elements representing values of possibly different types is defined by means of the (nongeneric) package `Streams.Stream_IO`.

5

Before input or output operations can be performed on a file, the file first has to be associated with an external file. While such an association is in effect, the file is said to be <open>, and otherwise the file is said to be <closed>.

6

The language does not define what happens to external files after the completion of the main program and all the library tasks (in particular, if corresponding files have not been closed). The effect of input-output for access types is unspecified.

7

An open file has a <current mode>, which is a value of one of the following enumeration types:

8

```
type File_Mode is (In_File, Inout_File, Out_File); --< for Direct_IO>■
```

9

These values correspond respectively to the cases where only reading, both reading and writing, or only writing are to be performed.

10/2

```
type File_Mode is (In_File, Out_File, Append_File);
```

--< for Sequential_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Stream_IO>

11

These values correspond respectively to the cases where only reading, only writing, or only appending are to be performed.

12

The mode of a file can be changed.

13/2

Several file management operations are common to Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. These operations are described in subclause Section 15.8.2 [A.8.2], page 685, for sequential and direct files. Any additional effects concerning text input–output are described in subclause Section 15.10.2 [A.10.2], page 709.

14

The exceptions that can be propagated by the execution of an input–output subprogram are defined in the package IO_Exceptions; the situations in which they can be propagated are described following the description of the subprogram (and in clause Section 15.13 [A.13], page 752). The exceptions Storage_Error and Program_Error may be propagated. (Program_Error can only be propagated due to errors made by the caller of the subprogram.) Finally, exceptions can be propagated in certain implementation–defined situations.

NOTES

15/2

18 Each instantiation of the generic packages Sequential_IO and Direct_IO declares a different type File_Type. In the case of Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Streams.Stream_IO, the corresponding type File_Type is unique.

16

19 A bidirectional device can often be modeled as two sequential files associated with the device, one of mode In_File, and one of mode Out_File. An implementation may restrict the number of files that may be associated with a given external file.

15.8 A.8 Sequential and Direct Files

Static Semantics

1/2

Two kinds of access to external files are defined in this subclause: <sequential access> and <direct access>. The corresponding file types and the associated operations are provided by the generic packages Sequential_IO and Direct_IO. A file object to be used for sequential access is called a <sequential file>, and one to be used for direct access is called a <direct file>. Access to <stream file>s is described in Section 15.12.1 [A.12.1], page 746.

2

For sequential access, the file is viewed as a sequence of values that are transferred in the order of their appearance (as produced by the program or by the external environment). When the file is opened with mode `In_File` or `Out_File`, transfer starts respectively from or to the beginning of the file. When the file is opened with mode `Append_File`, transfer to the file starts after the last element of the file.

3

For direct access, the file is viewed as a set of elements occupying consecutive positions in linear order; a value can be transferred to or from an element of the file at any selected position. The position of an element is specified by its `<index>`, which is a number, greater than zero, of the implementation-defined integer type `Count`. The first element, if any, has index one; the index of the last element, if any, is called the `<current size>`; the current size is zero if there are no elements. The current size is a property of the external file.

4

An open direct file has a `<current index>`, which is the index that will be used by the next read or write operation. When a direct file is opened, the current index is set to one. The current index of a direct file is a property of a file object, not of an external file.

15.8.1 A.8.1 The Generic Package `Sequential_IO`

Static Semantics

1

The generic library package `Sequential_IO` has the following declaration:

2

```
with Ada.IO_Exceptions;
generic
  type Element_Type(<>) is private;
package Ada.Sequential_IO is
```

3

```
  type File_Type is limited private;
```

4

```
  type File_Mode is (In_File, Out_File, Append_File);
```

5

```
  <-- File management>
```

6

```
  procedure Create(File : in out File_Type;
                  Mode : in File_Mode := Out_File;
                  Name : in String := "";
                  Form : in String := "");
```

7

```
procedure Open (File : in out File_Type;  
               Mode : in File_Mode;  
               Name : in String;  
               Form : in String := "");
```

8

```
procedure Close (File : in out File_Type);  
procedure Delete(File : in out File_Type);  
procedure Reset (File : in out File_Type; Mode : in File_Mode);  
procedure Reset (File : in out File_Type);
```

9

```
function Mode   (File : in File_Type) return File_Mode;  
function Name   (File : in File_Type) return String;  
function Form   (File : in File_Type) return String;
```

10

```
function Is_Open(File : in File_Type) return Boolean;
```

11

```
--< Input and output operations>
```

12

```
procedure Read (File : in File_Type; Item : out Element_Type);  
procedure Write (File : in File_Type; Item : in Element_Type);
```

13

```
function End_Of_File(File : in File_Type) return Boolean;
```

14

```
--< Exceptions>
```

15

```
Status_Error : exception renames IO_Exceptions.Status_Error;  
Mode_Error   : exception renames IO_Exceptions.Mode_Error;  
Name_Error   : exception renames IO_Exceptions.Name_Error;  
Use_Error    : exception renames IO_Exceptions.Use_Error;  
Device_Error : exception renames IO_Exceptions.Device_Error;  
End_Error    : exception renames IO_Exceptions.End_Error;  
Data_Error   : exception renames IO_Exceptions.Data_Error;
```

16

```
private
  ... -- <not specified by the language>
end Ada.Sequential_IO;
```

17/2

The type `File_Type` needs finalization (see Section 8.6 [7.6], page 295) in every instantiation of `Sequential_IO`.

15.8.2 A.8.2 File Management

Static Semantics

1

The procedures and functions described in this subclause provide for the control of external files; their declarations are repeated in each of the packages for sequential, direct, text, and stream input–output. For text input–output, the procedures `Create`, `Open`, and `Reset` have additional effects described in subclause Section 15.10.2 [A.10.2], page 709.

2

```
procedure Create(File : in out File_Type;
  Mode : in File_Mode := <default_mode>;
  Name : in String := "";
  Form : in String := "");
```

3/2

Establishes a new external file, with the given name and form, and associates this external file with the given file. The given file is left open. The current mode of the given file is set to the given access mode. The default access mode is the mode `Out_File` for sequential, stream, and text input–output; it is the mode `Inout_File` for direct input–output. For direct access, the size of the created file is implementation defined.

4

A null string for `Name` specifies an external file that is not accessible after the completion of the main program (a temporary file). A null string for `Form` specifies the use of the default options of the implementation for the external file.

5

The exception `Status_Error` is propagated if the given file is already open. The exception `Name_Error` is propagated if the string given as `Name` does not allow the identification of an external file. The exception `Use_Error` is propagated if, for the specified mode, the external environment does not support creation of an external file with the given name (in the absence of `Name_Error`) and form.

6

```
procedure Open(File : in out File_Type;  
              Mode : in File_Mode;  
              Name : in String;  
              Form : in String := "");
```

7

Associates the given file with an existing external file having the given name and form, and sets the current mode of the given file to the given mode. The given file is left open.

8

The exception `Status_Error` is propagated if the given file is already open. The exception `Name_Error` is propagated if the string given as `Name` does not allow the identification of an external file; in particular, this exception is propagated if no external file with the given name exists. The exception `Use_Error` is propagated if, for the specified mode, the external environment does not support opening for an external file with the given name (in the absence of `Name_Error`) and form.

9

```
procedure Close(File : in out File_Type);
```

10

Severs the association between the given file and its associated external file. The given file is left closed. In addition, for sequential files, if the file being closed has mode `Out_File` or `Append_File`, then the last element written

since the most recent open or reset is the last element that can be read from the file. If no elements have been written and the file mode is `Out_File`, then the closed file is empty. If no elements have been written and the file mode is `Append_File`, then the closed file is unchanged.

11

The exception `Status_Error` is propagated if the given file is not open.

12

```
procedure Delete(File : in out File_Type);
```

13

Deletes the external file associated with the given file. The given file is closed, and the external file ceases to exist.

14

The exception `Status_Error` is propagated if the given file is not open. The exception `Use_Error` is propagated if deletion of the external file is not supported by the external environment.

15

```
procedure Reset(File : in out File_Type; Mode : in File_Mode);  
procedure Reset(File : in out File_Type);
```

16/2

Resets the given file so that reading from its elements can be restarted from the beginning of the external file (for modes `In_File` and `Inout_File`), and so that writing to its elements can be restarted at the beginning of the external file (for modes `Out_File` and `Inout_File`) or after the last element of the external file (for mode `Append_File`). In particular, for direct access this means that the current index is set to one. If a `Mode` parameter is supplied, the current mode of the

given file is set to the given mode. In addition, for sequential files, if the given file has mode `Out_File` or `Append_File` when `Reset` is called, the last element written since the most recent open or reset is the last element that can be read from the external file. If no elements have been written and the file mode is `Out_File`, the reset file is empty. If no elements have been written and the file mode is `Append_File`, then the reset file is unchanged.

17

The exception `Status_Error` is propagated if the file is not open. The exception `Use_Error` is propagated if the external environment does not support resetting for the external file and, also, if the external environment does not support resetting to the specified mode for the external file.

18

```
function Mode(File : in File_Type) return File_Mode;
```

19

Returns the current mode of the given file.

20

The exception `Status_Error` is propagated if the file is not open.

21

```
function Name(File : in File_Type) return String;
```

22/2

Returns a string which uniquely identifies the external file currently associated with the given file (and may thus be used in an `Open` operation).

23

The exception `Status_Error` is propagated if the given file is not open. The exception

Use_Error is propagated if the associated external file is a temporary file that cannot be opened by any name.

24

```
function Form(File : in File_Type) return String;
```

25

Returns the form string for the external file currently associated with the given file. If an external environment allows alternative specifications of the form (for example, abbreviations using default options), the string returned by the function should correspond to a full specification (that is, it should indicate explicitly all options selected, including default options).

26

The exception Status_Error is propagated if the given file is not open.

27

```
function Is_Open(File : in File_Type) return Boolean;
```

28

Returns True if the file is open (that is, if it is associated with an external file), otherwise returns False.

Implementation Permissions

29

An implementation may propagate Name_Error or Use_Error if an attempt is made to use an I/O feature that cannot be supported by the implementation due to limitations in the external environment. Any such restriction should be documented.

15.8.3 A.8.3 Sequential Input-Output Operations

Static Semantics

1

The operations available for sequential input and output are described in this subclause. The exception Status_Error is propagated if any of these operations is attempted for a file that is not open.

2

```
procedure Read(File : in File_Type; Item : out Element_Type);
```

3

Operates on a file of mode `In_File`. Reads an element from the given file, and returns the value of this element in the `Item` parameter.

4

The exception `Mode_Error` is propagated if the mode is not `In_File`. The exception `End_Error` is propagated if no more elements can be read from the given file. The exception `Data_Error` can be propagated if the element read cannot be interpreted as a value of the subtype `Element_Type` (see Section 15.13 [A.13], page 752, "Section 15.13 [A.13], page 752, Exceptions in Input-Output").

5

```
procedure Write(File : in File_Type; Item : in Element_Type);
```

6

Operates on a file of mode `Out_File` or `Append_File`. Writes the value of `Item` to the given file.

7

The exception `Mode_Error` is propagated if the mode is not `Out_File` or `Append_File`. The exception `Use_Error` is propagated if the capacity of the external file is exceeded.

8

```
function End_Of_File(File : in File_Type) return Boolean;
```

9

Operates on a file of mode `In_File`. Returns `True` if no more elements can be read from the given file; otherwise returns `False`.

10

The exception `Mode_Error` is propagated if the mode is not `In_File`.

15.8.4 A.8.4 The Generic Package Direct_IO

Static Semantics

1

The generic library package Direct_IO has the following declaration:

2

```
with Ada.IO_Exceptions;  
generic  
  type Element_Type is private;  
package Ada.Direct_IO is
```

3

```
  type File_Type is limited private;
```

4

```
  type File_Mode is (In_File, Inout_File, Out_File);  
  type Count      is range 0 .. <implementation-defined>;  
  subtype Positive_Count is Count range 1 .. Count'Last;
```

5

```
  --< File management>
```

6

```
  procedure Create(File : in out File_Type;  
                  Mode : in File_Mode := Inout_File;  
                  Name : in String := "";  
                  Form : in String := "");
```

7

```
  procedure Open (File : in out File_Type;  
                 Mode : in File_Mode;  
                 Name : in String;  
                 Form : in String := "");
```

8

```
  procedure Close (File : in out File_Type);  
  procedure Delete(File : in out File_Type);  
  procedure Reset (File : in out File_Type; Mode : in File_Mode);  
  procedure Reset (File : in out File_Type);
```

9

```
  function Mode (File : in File_Type) return File_Mode;  
  function Name (File : in File_Type) return String;
```

```

10     function Form    (File : in File_Type) return String;

11     function Is_Open(File : in File_Type) return Boolean;

12     --< Input and output operations>

13     procedure Read (File : in File_Type; Item : out Element_Type;
14                   From : in Positive_Count);
15     procedure Read (File : in File_Type; Item : out Element_Type);

16     procedure Write(File : in File_Type; Item : in  Element_Type;
17                   To  : in Positive_Count);
18     procedure Write(File : in File_Type; Item : in  Element_Type);

19     procedure Set_Index(File : in File_Type; To : in Positive_Count);■

20     function Index(File : in File_Type) return Positive_Count;
21     function Size (File : in File_Type) return Count;

22     function End_Of_File(File : in File_Type) return Boolean;

23     --< Exceptions>

24     Status_Error : exception renames IO_Exceptions.Status_Error;
25     Mode_Error   : exception renames IO_Exceptions.Mode_Error;
26     Name_Error   : exception renames IO_Exceptions.Name_Error;
27     Use_Error    : exception renames IO_Exceptions.Use_Error;
28     Device_Error : exception renames IO_Exceptions.Device_Error;
29     End_Error    : exception renames IO_Exceptions.End_Error;
30     Data_Error   : exception renames IO_Exceptions.Data_Error;

31     private

```

```
... -- <not specified by the language>
end Ada.Direct_IO;
```

20/2

The type File_Type needs finalization (see Section 8.6 [7.6], page 295) in every instantiation of Direct_IO.

15.8.5 A.8.5 Direct Input-Output Operations

Static Semantics

1

The operations available for direct input and output are described in this subclause. The exception Status_Error is propagated if any of these operations is attempted for a file that is not open.

2

```
procedure Read(File : in File_Type; Item : out Element_Type;
               From : in Positive_Count);
procedure Read(File : in File_Type; Item : out Element_Type);
```

3

Operates on a file of mode In_File or Inout_File. In the case of the first form, sets the current index of the given file to the index value given by the parameter From. Then (for both forms) returns, in the parameter Item, the value of the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

4

The exception Mode_Error is propagated if the mode of the given file is Out_File. The exception End_Error is propagated if the index to be used exceeds the size of the external file. The exception Data_Error can be propagated if the element read cannot be interpreted as a value of the subtype Element_Type (see Section 15.13 [A.13], page 752).

5

```
procedure Write(File : in File_Type; Item : in Element_Type;
               To   : in Positive_Count);
procedure Write(File : in File_Type; Item : in Element_Type);
```

6

Operates on a file of mode `Inout_File` or `Out_File`. In the case of the first form, sets the index of the given file to the index value given by the parameter `To`. Then (for both forms) gives the value of the parameter `Item` to the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

7

The exception `Mode_Error` is propagated if the mode of the given file is `In_File`. The exception `Use_Error` is propagated if the capacity of the external file is exceeded.

8

```
procedure Set_Index(File : in File_Type; To : in Positive_Count);
```

9

Operates on a file of any mode. Sets the current index of the given file to the given index value (which may exceed the current size of the file).

10

```
function Index(File : in File_Type) return Positive_Count;
```

11

Operates on a file of any mode. Returns the current index of the given file.

12

```
function Size(File : in File_Type) return Count;
```

13

Operates on a file of any mode. Returns the current size of the external file that is associated with the given file.

14

```
function End_Of_File(File : in File_Type) return Boolean;
```

15

Operates on a file of mode `In_File` or `Inout_File`. Returns `True` if the current index exceeds the size of the external file; otherwise returns `False`.

16

The exception `Mode_Error` is propagated if the mode of the given file is `Out_File`.

NOTES

17

20 `Append_File` mode is not supported for the generic package `Direct_IO`.

15.9 A.9 The Generic Package `Storage_IO`

1

The generic package `Storage_IO` provides for reading from and writing to an in-memory buffer. This generic package supports the construction of user-defined input-output packages.

Static Semantics

2

The generic library package `Storage_IO` has the following declaration:

3

```
with Ada.IO_Exceptions;
with System.Storage_Elements;
generic
  type Element_Type is private;
package Ada.Storage_IO is
  pragma Preelaborate(Storage_IO);
```

4

```
  Buffer_Size : constant System.Storage_Elements.Storage_Count :=
    <implementation-defined>;
  subtype Buffer_Type is
    System.Storage_Elements.Storage_Array(1..Buffer_Size);
```

5

```
  --< Input and output operations>
```

6

```
  procedure Read (Buffer : in Buffer_Type; Item : out Element_Type);
```

7

```
procedure Write(Buffer : out Buffer_Type; Item : in Element_Type);■
```

8

```
--< Exceptions>
```

9

```
Data_Error : exception renames IO_Exceptions.Data_Error;  
end Ada.Storage_IO;
```

10

In each instance, the constant `Buffer_Size` has a value that is the size (in storage elements) of the buffer required to represent the content of an object of subtype `Element_Type`, including any implicit levels of indirection used by the implementation. The `Read` and `Write` procedures of `Storage_IO` correspond to the `Read` and `Write` procedures of `Direct_IO` (see Section 15.8.4 [A.8.4], page 691), but with the content of the `Item` parameter being read from or written into the specified `Buffer`, rather than an external file.

NOTES

11

21 A buffer used for `Storage_IO` holds only one element at a time; an external file used for `Direct_IO` holds a sequence of elements.

15.10 A.10 Text Input-Output

Static Semantics

1

This clause describes the package `Text_IO`, which provides facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of characters grouped into lines, and as a sequence of lines grouped into pages. The specification of the package is given below in subclause Section 15.10.1 [A.10.1], page 698.

2

The facilities for file management given above, in subclauses Section 15.8.2 [A.8.2], page 685, and Section 15.8.3 [A.8.3], page 689, are available for text input-output. In place of `Read` and `Write`, however, there are procedures `Get` and `Put` that input values of suitable types from text files, and output values to them. These values are provided to the `Put` procedures, and returned by the `Get` procedures, in a parameter `Item`. Several overloaded procedures of these names exist, for different types of `Item`. These `Get` procedures analyze the input sequences of characters based on lexical elements (see Section 2) and return the corresponding values; the `Put` procedures output the given values as appropriate lexical elements. Procedures `Get` and `Put` are also available that input and output individual characters treated as character values rather than as lexical elements. Related to character input are procedures to look ahead at the next character without reading it, and to read a character "immediately" without waiting for an end-of-line to signal availability.

3

In addition to the procedures `Get` and `Put` for numeric and enumeration types of `Item` that operate on text files, analogous procedures are provided that read from and write to a parameter of type `String`. These procedures perform the same analysis and composition of character sequences as their counterparts which have a file parameter.

4

For all `Get` and `Put` procedures that operate on text files, and for many other subprograms, there are forms with and without a file parameter. Each such `Get` procedure operates on an input file, and each such `Put` procedure operates on an output file. If no file is specified, a default input file or a default output file is used.

5

At the beginning of program execution the default input and output files are the so-called standard input file and standard output file. These files are open, have respectively the current modes `In_File` and `Out_File`, and are associated with two implementation-defined external files. Procedures are provided to change the current default input file and the current default output file.

6

At the beginning of program execution a default file for program-dependent error-related text output is the so-called standard error file. This file is open, has the current mode `Out_File`, and is associated with an implementation-defined external file. A procedure is provided to change the current default error file.

7

From a logical point of view, a text file is a sequence of pages, a page is a sequence of lines, and a line is a sequence of characters; the end of a line is marked by a `<line terminator>`; the end of a page is marked by the combination of a line terminator immediately followed by a `<page terminator>`; and the end of a file is marked by the combination of a line terminator immediately followed by a page terminator and then a `<file terminator>`. Terminators are generated during output; either by calls of procedures provided expressly for that purpose; or implicitly as part of other operations, for example, when a bounded line length, a bounded page length, or both, have been specified for a file.

8

The actual nature of terminators is not defined by the language and hence depends on the implementation. Although terminators are recognized or generated by certain of the procedures that follow, they are not necessarily implemented as characters or as sequences of characters. Whether they are characters (and if so which ones) in any particular implementation need not concern a user who neither explicitly outputs nor explicitly inputs control characters. The effect of input (`Get`) or output (`Put`) of control characters (other than horizontal tabulation) is not specified by the language.

9

The characters of a line are numbered, starting from one; the number of a character is called its `<column number>`. For a line terminator, a column number is also defined: it is one more than the number of characters in the line. The lines of a page, and the pages of a file, are similarly numbered. The current column number is the column number of the next character or line terminator to be transferred. The current line number is the number of the current line. The current page number is the number of the current page. These

numbers are values of the subtype Positive_Count of the type Count (by convention, the value zero of the type Count is used to indicate special conditions).

10

```
type Count is range 0 .. <implementation-defined>;
subtype Positive_Count is Count range 1 .. Count'Last;
```

11

For an output file or an append file, a <maximum line length> can be specified and a <maximum page length> can be specified. If a value to be output cannot fit on the current line, for a specified maximum line length, then a new line is automatically started before the value is output; if, further, this new line cannot fit on the current page, for a specified maximum page length, then a new page is automatically started before the value is output. Functions are provided to determine the maximum line length and the maximum page length. When a file is opened with mode Out_File or Append_File, both values are zero: by convention, this means that the line lengths and page lengths are unbounded. (Consequently, output consists of a single line if the subprograms for explicit control of line and page structure are not used.) The constant Unbounded is provided for this purpose.

15.10.1 A.10.1 The Package Text_IO

Static Semantics

1

The library package Text_IO has the following declaration:

2

```
with Ada.IO_Exceptions;
package Ada.Text_IO is
```

3

```
type File_Type is limited private;
```

4

```
type File_Mode is (In_File, Out_File, Append_File);
```

5

```
type Count is range 0 .. <implementation-defined>;
subtype Positive_Count is Count range 1 .. Count'Last;
Unbounded : constant Count := 0; --< line and page length>
```

6

```
subtype Field      is Integer range 0 .. <implementation-defined>;
subtype Number_Base is Integer range 2 .. 16;
```

7

```
type Type_Set is (Lower_Case, Upper_Case);
```

8

```
--< File Management>
```

9

```
procedure Create (File : in out File_Type;  
                 Mode : in File_Mode := Out_File;  
                 Name : in String    := "";  
                 Form : in String    := "");
```

10

```
procedure Open  (File : in out File_Type;  
                 Mode : in File_Mode;  
                 Name : in String;  
                 Form : in String := "");
```

11

```
procedure Close (File : in out File_Type);  
procedure Delete (File : in out File_Type);  
procedure Reset (File : in out File_Type; Mode : in File_Mode);  
procedure Reset (File : in out File_Type);
```

12

```
function Mode  (File : in File_Type) return File_Mode;  
function Name  (File : in File_Type) return String;  
function Form  (File : in File_Type) return String;
```

13

```
function Is_Open (File : in File_Type) return Boolean;
```

14

```
--< Control of default input and output files>
```

15

```
procedure Set_Input (File : in File_Type);  
procedure Set_Output (File : in File_Type);  
procedure Set_Error (File : in File_Type);
```

16

```
function Standard_Input  return File_Type;  
function Standard_Output return File_Type;  
function Standard_Error  return File_Type;
```

17

```
function Current_Input    return File_Type;  
function Current_Output  return File_Type;  
function Current_Error    return File_Type;
```

18

```
type File_Access is access constant File_Type;
```

19

```
function Standard_Input  return File_Access;  
function Standard_Output return File_Access;  
function Standard_Error  return File_Access;
```

20

```
function Current_Input    return File_Access;  
function Current_Output  return File_Access;  
function Current_Error    return File_Access;
```

21/1

```
--<Buffer control>  
procedure Flush (File : in File_Type);  
procedure Flush;
```

22

```
--< Specification of line and page lengths>
```

23

```
procedure Set_Line_Length(File : in File_Type; To : in Count);  
procedure Set_Line_Length(To   : in Count);
```

24

```
procedure Set_Page_Length(File : in File_Type; To : in Count);  
procedure Set_Page_Length(To   : in Count);
```

25

```
function Line_Length(File : in File_Type) return Count;  
function Line_Length return Count;
```

26

```
function Page_Length(File : in File_Type) return Count;  
function Page_Length return Count;
```

27

```
--< Column, Line, and Page Control>
```

28

```
procedure New_Line (File      : in File_Type;  
                   Spacing   : in Positive_Count := 1);  
procedure New_Line (Spacing : in Positive_Count := 1);
```

29

```
procedure Skip_Line (File      : in File_Type;  
                   Spacing   : in Positive_Count := 1);  
procedure Skip_Line (Spacing : in Positive_Count := 1);
```

30

```
function End_Of_Line(File : in File_Type) return Boolean;  
function End_Of_Line return Boolean;
```

31

```
procedure New_Page (File : in File_Type);  
procedure New_Page;
```

32

```
procedure Skip_Page (File : in File_Type);  
procedure Skip_Page;
```

33

```
function End_Of_Page(File : in File_Type) return Boolean;  
function End_Of_Page return Boolean;
```

34

```
function End_Of_File(File : in File_Type) return Boolean;  
function End_Of_File return Boolean;
```

35

```
procedure Set_Col (File : in File_Type; To : in Positive_Count);  
procedure Set_Col (To   : in Positive_Count);
```

36

```
procedure Set_Line(File : in File_Type; To : in Positive_Count);  
procedure Set_Line(To   : in Positive_Count);
```

37

```
function Col (File : in File_Type) return Positive_Count;  
function Col  return Positive_Count;
```

38

```
function Line(File : in File_Type) return Positive_Count;  
function Line  return Positive_Count;
```

39

```
function Page(File : in File_Type) return Positive_Count;  
function Page  return Positive_Count;
```

40

```
--< Character Input-Output>
```

41

```
procedure Get(File : in File_Type; Item : out Character);  
procedure Get(Item : out Character);
```

42

```
procedure Put(File : in File_Type; Item : in Character);  
procedure Put(Item : in Character);
```

43

```
procedure Look_Ahead (File      : in File_Type;  
                      Item      : out Character;  
                      End_Of_Line : out Boolean);  
procedure Look_Ahead (Item      : out Character;  
                      End_Of_Line : out Boolean);
```

44

```
procedure Get_Immediate(File      : in File_Type;  
                        Item      : out Character);  
procedure Get_Immediate(Item      : out Character);
```

45

```
procedure Get_Immediate(File      : in File_Type;  
                        Item      : out Character;  
                        Available : out Boolean);  
procedure Get_Immediate(Item      : out Character;  
                        Available : out Boolean);
```

46

```
--< String Input-Output>
```

47

```
procedure Get(File : in File_Type; Item : out String);  
procedure Get(Item : out String);
```

48

```
procedure Put(File : in File_Type; Item : in String);  
procedure Put(Item : in String);
```

49

```
procedure Get_Line(File : in File_Type;  
                  Item : out String;  
                  Last : out Natural);  
procedure Get_Line(Item : out String; Last : out Natural);
```

49.1/2

```
function Get_Line(File : in File_Type) return String;  
function Get_Line return String;
```

50

```
procedure Put_Line(File : in File_Type; Item : in String);  
procedure Put_Line(Item : in String);
```

51

```
--< Generic packages for Input-Output of Integer Types>
```

52

```
generic  
  type Num is range <>;  
package Integer_IO is
```

53

```
  Default_Width : Field := Num'Width;  
  Default_Base  : Number_Base := 10;
```

54

```
  procedure Get(File : in File_Type;  
               Item : out Num;  
               Width : in Field := 0);  
  procedure Get(Item : out Num);
```

```

55         Width : in Field := 0);

procedure Put(File : in File_Type;
              Item : in Num;
              Width : in Field := Default_Width;
              Base : in Number_Base := Default_Base);
procedure Put(Item : in Num;
              Width : in Field := Default_Width;
              Base : in Number_Base := Default_Base);
procedure Get(From : in String;
              Item : out Num;
              Last : out Positive);
procedure Put(To : out String;
              Item : in Num;
              Base : in Number_Base := Default_Base);

56

end Integer_IO;

57

generic
  type Num is mod <>;
package Modular_IO is

58

    Default_Width : Field := Num'Width;
    Default_Base : Number_Base := 10;

59

    procedure Get(File : in File_Type;
                  Item : out Num;
                  Width : in Field := 0);
    procedure Get(Item : out Num;
                  Width : in Field := 0);

60

    procedure Put(File : in File_Type;
                  Item : in Num;
                  Width : in Field := Default_Width;
                  Base : in Number_Base := Default_Base);
    procedure Put(Item : in Num;
                  Width : in Field := Default_Width;
                  Base : in Number_Base := Default_Base);

```



```

        procedure Get(From : in String;
                     Item  : out Num;
                     Last  : out Positive);
        procedure Put(To   : out String;
                     Item  : in Num;
                     Base  : in Number_Base := Default_Base);
61
    end Modular_IO;
62
    --< Generic packages for Input-Output of Real Types>
63
    generic
        type Num is digits <>;
    package Float_IO is
64
        Default_Fore : Field := 2;
        Default_Aft  : Field := Num'Digits-1;
        Default_Exp  : Field := 3;
65

        procedure Get(File : in File_Type;
                     Item  : out Num;
                     Width : in Field := 0);
        procedure Get(Item : out Num;
                     Width : in Field := 0);
66

        procedure Put(File : in File_Type;
                     Item  : in Num;
                     Fore  : in Field := Default_Fore;
                     Aft   : in Field := Default_Aft;
                     Exp   : in Field := Default_Exp);
        procedure Put(Item : in Num;
                     Fore  : in Field := Default_Fore;
                     Aft   : in Field := Default_Aft;
                     Exp   : in Field := Default_Exp);
67

        procedure Get(From : in String;
                     Item  : out Num;

```

```
        Last : out Positive);
    procedure Put(To    : out String;
                 Item  : in Num;
                 Aft   : in Field := Default_Aft;
                 Exp   : in Field := Default_Exp);
end Float_IO;
```

68

```
generic
    type Num is delta <>;
package Fixed_IO is
```

69

```
    Default_Fore : Field := Num'Fore;
    Default_Aft  : Field := Num'Aft;
    Default_Exp  : Field := 0;
```

70

```
    procedure Get(File : in File_Type;
                  Item  : out Num;
                  Width : in Field := 0);
    procedure Get(Item  : out Num;
                  Width : in Field := 0);
```

71

```
    procedure Put(File : in File_Type;
                  Item  : in Num;
                  Fore  : in Field := Default_Fore;
                  Aft   : in Field := Default_Aft;
                  Exp   : in Field := Default_Exp);
    procedure Put(Item  : in Num;
                  Fore  : in Field := Default_Fore;
                  Aft   : in Field := Default_Aft;
                  Exp   : in Field := Default_Exp);
```

72

```
    procedure Get(From : in String;
                  Item  : out Num;
                  Last  : out Positive);
    procedure Put(To    : out String;
                  Item  : in Num;
                  Aft   : in Field := Default_Aft;
                  Exp   : in Field := Default_Exp);
end Fixed_IO;
```

73

```
generic
  type Num is delta <> digits <>;
package Decimal_IO is
```

74

```
  Default_Fore : Field := Num'Fore;
  Default_Aft  : Field := Num'Aft;
  Default_Exp  : Field := 0;
```

75

```
  procedure Get(File : in File_Type;
                Item  : out Num;
                Width : in Field := 0);
  procedure Get(Item  : out Num;
                Width : in Field := 0);
```

76

```
  procedure Put(File : in File_Type;
                Item  : in Num;
                Fore  : in Field := Default_Fore;
                Aft   : in Field := Default_Aft;
                Exp   : in Field := Default_Exp);
  procedure Put(Item  : in Num;
                Fore  : in Field := Default_Fore;
                Aft   : in Field := Default_Aft;
                Exp   : in Field := Default_Exp);
```

77

```
  procedure Get(From : in String;
                Item  : out Num;
                Last  : out Positive);
  procedure Put(To   : out String;
                Item  : in Num;
                Aft   : in Field := Default_Aft;
                Exp   : in Field := Default_Exp);
end Decimal_IO;
```

78

```
--< Generic package for Input-Output of Enumeration Types>
```

79

```
generic
```

```

    type Enum is (<>);
package Enumeration_IO is
80

    Default_Width   : Field := 0;
    Default_Setting : Type_Set := Upper_Case;
81

    procedure Get(File : in File_Type;
                  Item : out Enum);
    procedure Get(Item : out Enum);
82

    procedure Put(File : in File_Type;
                  Item : in Enum;
                  Width : in Field := Default_Width;
                  Set : in Type_Set := Default_Setting);
    procedure Put(Item : in Enum;
                  Width : in Field := Default_Width;
                  Set : in Type_Set := Default_Setting);
83

    procedure Get(From : in String;
                  Item : out Enum;
                  Last : out Positive);
    procedure Put(To : out String;
                  Item : in Enum;
                  Set : in Type_Set := Default_Setting);
end Enumeration_IO;
84

--< Exceptions>
85

Status_Error : exception renames IO_Exceptions.Status_Error;
Mode_Error   : exception renames IO_Exceptions.Mode_Error;
Name_Error   : exception renames IO_Exceptions.Name_Error;
Use_Error    : exception renames IO_Exceptions.Use_Error;
Device_Error : exception renames IO_Exceptions.Device_Error;
End_Error    : exception renames IO_Exceptions.End_Error;
Data_Error   : exception renames IO_Exceptions.Data_Error;
Layout_Error : exception renames IO_Exceptions.Layout_Error;
private
    ... -- <not specified by the language>

```

end Ada.Text_IO;

86/2

The type File_Type needs finalization (see Section 8.6 [7.6], page 295).

15.10.2 A.10.2 Text File Management

Static Semantics

1

The only allowed file modes for text files are the modes In_File, Out_File, and Append_File. The subprograms given in subclause Section 15.8.2 [A.8.2], page 685, for the control of external files, and the function End_Of_File given in subclause Section 15.8.3 [A.8.3], page 689, for sequential input–output, are also available for text files. There is also a version of End_Of_File that refers to the current default input file. For text files, the procedures have the following additional effects:

2

- For the procedures Create and Open: After a file with mode Out_File or Append_File is opened, the page length and line length are unbounded (both have the conventional value zero). After a file (of any mode) is opened, the current column, current line, and current page numbers are set to one. If the mode is Append_File, it is implementation defined whether a page terminator will separate preexisting text in the file from the new text to be written.

3

- For the procedure Close: If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator.

4

- For the procedure Reset: If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator. The current column, line, and page numbers are set to one, and the line and page lengths to Unbounded. If the new mode is Append_File, it is implementation defined whether a page terminator will separate preexisting text in the file from the new text to be written.

5

The exception Mode_Error is propagated by the procedure Reset upon an attempt to change the mode of a file that is the current default input file, the current default output file, or the current default error file.

NOTES

6

22 An implementation can define the Form parameter of Create and Open to control effects including the following:

7

- the interpretation of line and column numbers for an interactive file, and

8

- the interpretation of text formats in a file created by a foreign program.

15.10.3 A.10.3 Default Input, Output, and Error Files

Static Semantics

1

The following subprograms provide for the control of the particular default files that are used when a file parameter is omitted from a Get, Put, or other operation of text input–output described below, or when application–dependent error–related text is to be output.

2

```
procedure Set_Input(File : in File_Type);
```

3

Operates on a file of mode In_File. Sets the current default input file to File.

4

The exception Status_Error is propagated if the given file is not open. The exception Mode_Error is propagated if the mode of the given file is not In_File.

5

```
procedure Set_Output(File : in File_Type);  
procedure Set_Error (File : in File_Type);
```

6

Each operates on a file of mode Out_File or Append_File. Set_Output sets the current default output file to File. Set_Error sets the current default error file to File. The exception Status_Error is propagated if the given file is not open. The exception Mode_Error is propagated if the mode of the given file is not Out_File or Append_File.

7

```
function Standard_Input return File_Type;  
function Standard_Input return File_Access;
```

8

Returns the standard input file (see Section 15.10 [A.10], page 696), or an access value designating the standard input file, respectively.

9

```
function Standard_Output return File_Type;  
function Standard_Output return File_Access;
```

10

Returns the standard output file (see Section 15.10 [A.10], page 696) or an access value designating the standard output file, respectively.

11

```
function Standard_Error return File_Type;  
function Standard_Error return File_Access;
```

12/1

Returns the standard error file (see Section 15.10 [A.10], page 696), or an access value designating the standard error file, respectively.

13

The Form strings implicitly associated with the opening of Standard.Input, Standard.Output, and Standard.Error at the start of program execution are implementation defined.

14

```
function Current_Input return File_Type;  
function Current_Input return File_Access;
```

15

Returns the current default input file, or an access value designating the current default input file, respectively.

16

```
function Current_Output return File_Type;  
function Current_Output return File_Access;
```

17

Returns the current default output file, or an access value designating the current default output file, respectively.

18

```
function Current_Error return File_Type;  
function Current_Error return File_Access;
```

19

Returns the current default error file, or an access value designating the current default error file, respectively.

20/1

```
procedure Flush (File : in File_Type);  
procedure Flush;
```

21

The effect of Flush is the same as the corresponding subprogram in Streams.Stream_IO (see Section 15.12.1 [A.12.1], page 746). If File is not explicitly specified, Current_Output is used.

Erroneous Execution

22/1

The execution of a program is erroneous if it invokes an operation on a current default input, default output, or default error file, and if the corresponding file object is closed or no longer exists.

23/1

<This paragraph was deleted.>

NOTES

24

23 The standard input, standard output, and standard error files cannot be opened, closed, reset, or deleted, because the parameter File of the corresponding procedures has the mode in out.

24 The standard input, standard output, and standard error files are different file objects, but not necessarily different external files.

15.10.4 A.10.4 Specification of Line and Page Lengths

Static Semantics

1

The subprograms described in this subclause are concerned with the line and page structure of a file of mode `Out_File` or `Append_File`. They operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the current default output file. They provide for output of text with a specified maximum line length or page length. In these cases, line and page terminators are output implicitly and automatically when needed. When line and page lengths are unbounded (that is, when they have the conventional value zero), as in the case of a newly opened file, new lines and new pages are only started when explicitly called for.

2

In all cases, the exception `Status_Error` is propagated if the file to be used is not open; the exception `Mode_Error` is propagated if the mode of the file is not `Out_File` or `Append_File`.

3

```
procedure Set_Line_Length(File : in File_Type; To : in Count);
procedure Set_Line_Length(To   : in Count);
```

4

Sets the maximum line length of the specified output or append file to the number of characters specified by `To`. The value zero for `To` specifies an unbounded line length.

5

The exception `Use_Error` is propagated if the specified line length is inappropriate for the associated external file.

6

```
procedure Set_Page_Length(File : in File_Type; To : in Count);
procedure Set_Page_Length(To   : in Count);
```

7

Sets the maximum page length of the specified output or append file to the number of lines specified by `To`. The value zero for `To` specifies an unbounded page length.

8

The exception `Use_Error` is propagated if the specified page length is inappropriate for the associated external file.

9

```
function Line_Length(File : in File_Type) return Count;  
function Line_Length return Count;
```

10

Returns the maximum line length currently set for the specified output or append file, or zero if the line length is unbounded.

11

```
function Page_Length(File : in File_Type) return Count;  
function Page_Length return Count;
```

12

Returns the maximum page length currently set for the specified output or append file, or zero if the page length is unbounded.

15.10.5 A.10.5 Operations on Columns, Lines, and Pages

Static Semantics

1

The subprograms described in this subclause provide for explicit control of line and page structure; they operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the appropriate (input or output) current default file. The exception `Status_Error` is propagated by any of these subprograms if the file to be used is not open.

2

```
procedure New_Line(File : in File_Type; Spacing : in Positive_Count := 1);  
procedure New_Line(Spacing : in Positive_Count := 1);
```

3

Operates on a file of mode `Out_File` or `Append_File`.

4

For a `Spacing` of one: Outputs a line terminator and sets the current column number to

one. Then increments the current line number by one, except in the case that the current line number is already greater than or equal to the maximum page length, for a bounded page length; in that case a page terminator is output, the current page number is incremented by one, and the current line number is set to one.

5

For a Spacing greater than one, the above actions are performed Spacing times.

6

The exception Mode_Error is propagated if the mode is not Out_File or Append_File.

7

```
procedure Skip_Line(File : in File_Type; Spacing : in Positive_Count := 1);  
procedure Skip_Line(Spacing : in Positive_Count := 1);
```

8

Operates on a file of mode In_File.

9

For a Spacing of one: Reads and discards all characters until a line terminator has been read, and then sets the current column number to one. If the line terminator is not immediately followed by a page terminator, the current line number is incremented by one. Otherwise, if the line terminator is immediately followed by a page terminator, then the page terminator is skipped, the current page number is incremented by one, and the current line number is set to one.

10

For a Spacing greater than one, the above actions are performed Spacing times.

11

The exception Mode_Error is propagated if the mode is not In_File. The exception

End_Error is propagated if an attempt is made to read a file terminator.

12

```
function End_Of_Line(File : in File_Type) return Boolean;  
function End_Of_Line return Boolean;
```

13

Operates on a file of mode In_File. Returns True if a line terminator or a file terminator is next; otherwise returns False.

14

The exception Mode_Error is propagated if the mode is not In_File.

15

```
procedure New_Page(File : in File_Type);  
procedure New_Page;
```

16

Operates on a file of mode Out_File or Append_File. Outputs a line terminator if the current line is not terminated, or if the current page is empty (that is, if the current column and line numbers are both equal to one). Then outputs a page terminator, which terminates the current page. Adds one to the current page number and sets the current column and line numbers to one.

17

The exception Mode_Error is propagated if the mode is not Out_File or Append_File.

18

```
procedure Skip_Page(File : in File_Type);  
procedure Skip_Page;
```

19

Operates on a file of mode In_File. Reads and discards all characters and line terminators until a page terminator has been read. Then

adds one to the current page number, and sets the current column and line numbers to one.

20

The exception `Mode_Error` is propagated if the mode is not `In_File`. The exception `End_Error` is propagated if an attempt is made to read a file terminator.

21

```
function End_Of_Page(File : in File_Type) return Boolean;  
function End_Of_Page return Boolean;
```

22

Operates on a file of mode `In_File`. Returns `True` if the combination of a line terminator and a page terminator is next, or if a file terminator is next; otherwise returns `False`.

23

The exception `Mode_Error` is propagated if the mode is not `In_File`.

24

```
function End_Of_File(File : in File_Type) return Boolean;  
function End_Of_File return Boolean;
```

25

Operates on a file of mode `In_File`. Returns `True` if a file terminator is next, or if the combination of a line, a page, and a file terminator is next; otherwise returns `False`.

26

The exception `Mode_Error` is propagated if the mode is not `In_File`.

27

The following subprograms provide for the control of the current position of reading or writing in a file. In all cases, the default file is the current output file.

28

```
procedure Set_Col(File : in File_Type; To : in Positive_Count);  
procedure Set_Col(To : in Positive_Count);
```

29

If the file mode is `Out_File` or `Append_File`:

30

- If the value specified by `To` is greater than the current column number, outputs spaces, adding one to the current column number after each space, until the current column number equals the specified value. If the value specified by `To` is equal to the current column number, there is no effect. If the value specified by `To` is less than the current column number, has the effect of calling `New_Line` (with a spacing of one), then outputs $(To - 1)$ spaces, and sets the current column number to the specified value.

31

- The exception `Layout_Error` is propagated if the value specified by `To` exceeds `Line_Length` when the line length is bounded (that is, when it does not have the conventional value zero).

32

If the file mode is `In_File`:

33

- Reads (and discards) individual characters, line terminators, and page terminators, until the next character to be read has a column number that equals the value specified by `To`; there is no effect if the current column number already equals this value. Each transfer of a character or terminator maintains the current column, line, and page numbers in the same way as a `Get` procedure (see Section 15.10.6 [A.10.6], page 721). (Short lines will be skipped until a line is reached that

has a character at the specified column position.)

34

- The exception `End_Error` is propagated if an attempt is made to read a file terminator.

35

```
procedure Set_Line(File : in File_Type; To : in Positive_Count);  
procedure Set_Line(To   : in Positive_Count);
```

36

If the file mode is `Out_File` or `Append_File`:

37

- If the value specified by `To` is greater than the current line number, has the effect of repeatedly calling `New_Line` (with a spacing of one), until the current line number equals the specified value. If the value specified by `To` is equal to the current line number, there is no effect. If the value specified by `To` is less than the current line number, has the effect of calling `New_Page` followed by a call of `New_Line` with a spacing equal to $(To - 1)$.

38

- The exception `Layout_Error` is propagated if the value specified by `To` exceeds `Page_Length` when the page length is bounded (that is, when it does not have the conventional value zero).

39

If the mode is `In_File`:

40

- Has the effect of repeatedly calling `Skip_Line` (with a spacing of one), until the current line number equals the value specified by `To`; there is no effect if the

current line number already equals this value. (Short pages will be skipped until a page is reached that has a line at the specified line position.)

41

- The exception `End_Error` is propagated if an attempt is made to read a file terminator.

42

```
function Col(File : in File_Type) return Positive_Count;  
function Col return Positive_Count;
```

43

Returns the current column number.

44

The exception `Layout_Error` is propagated if this number exceeds `Count'Last`.

45

```
function Line(File : in File_Type) return Positive_Count;  
function Line return Positive_Count;
```

46

Returns the current line number.

47

The exception `Layout_Error` is propagated if this number exceeds `Count'Last`.

48

```
function Page(File : in File_Type) return Positive_Count;  
function Page return Positive_Count;
```

49

Returns the current page number.

50

The exception `Layout_Error` is propagated if this number exceeds `Count'Last`.

51

The column number, line number, or page number are allowed to exceed Count'Last (as a consequence of the input or output of sufficiently many characters, lines, or pages). These events do not cause any exception to be propagated. However, a call of Col, Line, or Page propagates the exception Layout_Error if the corresponding number exceeds Count'Last.

NOTES

52

25 A page terminator is always skipped whenever the preceding line terminator is skipped. An implementation may represent the combination of these terminators by a single character, provided that it is properly recognized on input.

15.10.6 A.10.6 Get and Put Procedures

Static Semantics

1

The procedures Get and Put for items of the type Character, String, numeric types, and enumeration types are described in subsequent subclauses. Features of these procedures that are common to most of these types are described in this subclause. The Get and Put procedures for items of type Character and String deal with individual character values; the Get and Put procedures for numeric and enumeration types treat the items as lexical elements.

2

All procedures Get and Put have forms with a file parameter, written first. Where this parameter is omitted, the appropriate (input or output) current default file is understood to be specified. Each procedure Get operates on a file of mode In_File. Each procedure Put operates on a file of mode Out_File or Append_File.

3

All procedures Get and Put maintain the current column, line, and page numbers of the specified file: the effect of each of these procedures upon these numbers is the result of the effects of individual transfers of characters and of individual output or skipping of terminators. Each transfer of a character adds one to the current column number. Each output of a line terminator sets the current column number to one and adds one to the current line number. Each output of a page terminator sets the current column and line numbers to one and adds one to the current page number. For input, each skipping of a line terminator sets the current column number to one and adds one to the current line number; each skipping of a page terminator sets the current column and line numbers to one and adds one to the current page number. Similar considerations apply to the procedures Get_Line, Put_Line, and Set_Col.

4

Several Get and Put procedures, for numeric and enumeration types, have <format> parameters which specify field lengths; these parameters are of the nonnegative subtype Field of the type Integer.

5/2

Input-output of enumeration values uses the syntax of the corresponding lexical elements.

Any Get procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators. A <blank> is defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input.

6

For a numeric type, the Get procedures have a format parameter called Width. If the value given for this parameter is zero, the Get procedure proceeds in the same manner as for enumeration types, but using the syntax of numeric literals instead of that of enumeration literals. If a nonzero value is given, then exactly Width characters are input, or the characters up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. The syntax used for numeric literals is an extended syntax that allows a leading sign (but no intervening blanks, or line or page terminators) and that also allows (for real types) an integer literal as well as forms that have digits only before the point or only after the point.

7

Any Put procedure, for an item of a numeric or an enumeration type, outputs the value of the item as a numeric literal, identifier, or character literal, as appropriate. This is preceded by leading spaces if required by the format parameters Width or Fore (as described in later subclauses), and then a minus sign for a negative value; for an enumeration type, the spaces follow instead of leading. The format given for a Put procedure is overridden if it is insufficiently wide, by using the minimum needed width.

8

Two further cases arise for Put procedures for numeric and enumeration types, if the line length of the specified output file is bounded (that is, if it does not have the conventional value zero). If the number of characters to be output does not exceed the maximum line length, but is such that they cannot fit on the current line, starting from the current column, then (in effect) New_Line is called (with a spacing of one) before output of the item. Otherwise, if the number of characters exceeds the maximum line length, then the exception Layout_Error is propagated and nothing is output.

9

The exception Status_Error is propagated by any of the procedures Get, Get_Line, Put, and Put_Line if the file to be used is not open. The exception Mode_Error is propagated by the procedures Get and Get_Line if the mode of the file to be used is not In_File; and by the procedures Put and Put_Line, if the mode is not Out_File or Append_File.

10

The exception End_Error is propagated by a Get procedure if an attempt is made to skip a file terminator. The exception Data_Error is propagated by a Get procedure if the sequence finally input is not a lexical element corresponding to the type, in particular if no characters were input; for this test, leading blanks are ignored; for an item of a numeric type, when a sign is input, this rule applies to the succeeding numeric literal. The exception Layout_Error is propagated by a Put procedure that outputs to a parameter of type String, if the length of the actual string is insufficient for the output of the item.

Examples

11

In the examples, here and in subclauses Section 15.10.8 [A.10.8], page 727, and Section 15.10.9 [A.10.9], page 731, the string quotes and the lower case letter b are not transferred: they are shown only to reveal the layout and spaces.

12

```
N : Integer;  
    ...  
Get(N);
```

13

```
<--      Characters at input      Sequence input      Value of N>  
  
<--      bb-12535b      -12535      -12535>  
<--      bb12_535e1b      12_535e1      125350>  
<--      bb12_535e;      12_535e      (none) Data_Error raised>
```

14

Example of overridden width parameter:

15

```
Put(Item => -23, Width => 2);  --<  "-23">
```

15.10.7 A.10.7 Input-Output of Characters and Strings

Static Semantics

1

For an item of type Character the following procedures are provided:

2

```
procedure Get(File : in File_Type; Item : out Character);  
procedure Get(Item : out Character);
```

3

After skipping any line terminators and any page terminators, reads the next character from the specified input file and returns the value of this character in the out parameter Item.

4

The exception End_Error is propagated if an attempt is made to skip a file terminator.

5

```
procedure Put(File : in File_Type; Item : in Character);
```

```
procedure Put(Item : in Character);
```

6

If the line length of the specified output file is bounded (that is, does not have the conventional value zero), and the current column number exceeds it, has the effect of calling `New_Line` with a spacing of one. Then, or otherwise, outputs the given character to the file.

7

```
procedure Look_Ahead (File      : in  File_Type;
                      Item      : out Character;
                      End_Of_Line : out Boolean);
procedure Look_Ahead (Item      : out Character;
                      End_Of_Line : out Boolean);
```

8/1

`Mode_Error` is propagated if the mode of the file is not `In_File`. Sets `End_Of_Line` to `True` if at end of line, including if at end of page or at end of file; in each of these cases the value of `Item` is not specified. Otherwise `End_Of_Line` is set to `False` and `Item` is set to the next character (without consuming it) from the file.

9

```
procedure Get_Immediate(File : in  File_Type;
                        Item  : out Character);
procedure Get_Immediate(Item : out Character);
```

10

Reads the next character, either control or graphic, from the specified `File` or the default input file. `Mode_Error` is propagated if the mode of the file is not `In_File`. `End_Error` is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

11

```
procedure Get_Immediate(File      : in  File_Type;
                        Item      : out Character);
```

```
                Available : out Boolean);  
procedure Get_Immediate(Item      : out Character;  
                Available : out Boolean);
```

12

If a character, either control or graphic, is available from the specified File or the default input file, then the character is read; Available is True and Item contains the value of this character. If a character is not available, then Available is False and the value of Item is not specified. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

13/2

For an item of type String the following subprograms are provided:

14

```
procedure Get(File : in File_Type; Item : out String);  
procedure Get(Item : out String);
```

15

Determines the length of the given string and attempts that number of Get operations for successive characters of the string (in particular, no operation is performed if the string is null).

16

```
procedure Put(File : in File_Type; Item : in String);  
procedure Put(Item : in String);
```

17

Determines the length of the given string and attempts that number of Put operations for successive characters of the string (in particular, no operation is performed if the string is null).

17.1/2

```
function Get_Line(File : in File_Type) return String;  
function Get_Line return String;
```

17.2/2

Returns a result string constructed by reading successive characters from the specified input file, and assigning them to successive characters of the result string. The result string has a lower bound of 1 and an upper bound of the number of characters read. Reading stops when the end of the line is met; `Skip_Line` is then (in effect) called with a spacing of 1.

17.3/2

`Constraint_Error` is raised if the length of the line exceeds `Positive'Last`; in this case, the line number and page number are unchanged, and the column number is unspecified but no less than it was before the call. The exception `End_Error` is propagated if an attempt is made to skip a file terminator.

18

```
procedure Get_Line(File : in File_Type;
                  Item : out String;
                  Last : out Natural);
procedure Get_Line(Item : out String; Last : out Natural);
```

19

Reads successive characters from the specified input file and assigns them to successive characters of the specified string. Reading stops if the end of the string is met. Reading also stops if the end of the line is met before meeting the end of the string; in this case `Skip_Line` is (in effect) called with a spacing of 1. The values of characters not assigned are not specified.

20

If characters are read, returns in `Last` the index value such that `Item>Last` is the last character assigned (the index of the first character assigned is `Item'First`). If no characters are read, returns in `Last` an index value

that is one less than Item'First. The exception End_Error is propagated if an attempt is made to skip a file terminator.

21

```
procedure Put_Line(File : in File_Type; Item : in String);  
procedure Put_Line(Item : in String);
```

22

Calls the procedure Put for the given string, and then the procedure New_Line with a spacing of one.

Implementation Advice

23

The Get_Immediate procedures should be implemented with unbuffered input. For a device such as a keyboard, input should be "available" if a key has already been typed, whereas for a disk file, input should always be available except at end of file. For a file associated with a keyboard-like device, any line-editing features of the underlying operating system should be disabled during the execution of Get_Immediate.

NOTES

24

26 Get_Immediate can be used to read a single key from the keyboard "immediately"; that is, without waiting for an end of line. In a call of Get_Immediate without the parameter Available, the caller will wait until a character is available.

25

27 In a literal string parameter of Put, the enclosing string bracket characters are not output. Each doubled string bracket character in the enclosed string is output as a single string bracket character, as a consequence of the rule for string literals (see Section 3.6 [2.6], page 42).

26

28 A string read by Get or written by Put can extend over several lines. An implementation is allowed to assume that certain external files do not contain page terminators, in which case Get_Line and Skip_Line can return as soon as a line terminator is read.

15.10.8 A.10.8 Input-Output for Integer Types

Static Semantics

1

The following procedures are defined in the generic packages Integer_IO and Modular_IO,

which have to be instantiated for the appropriate signed integer or modular type respectively (indicated by Num in the specifications).

2

Values are output as decimal or based literals, without low line characters or exponent, and, for Integer_IO, preceded by a minus sign if negative. The format (which includes any leading spaces and minus sign) can be specified by an optional field width parameter. Values of widths of fields in output formats are of the nonnegative integer subtype Field. Values of bases are of the integer subtype Number_Base.

3

```
subtype Number_Base is Integer range 2 .. 16;
```

4

The default field width and base to be used by output procedures are defined by the following variables that are declared in the generic packages Integer_IO and Modular_IO:

5

```
Default_Width : Field := Num'Width;  
Default_Base  : Number_Base := 10;
```

6

The following procedures are provided:

7

```
procedure Get(File : in File_Type; Item : out Num; Width : in Field := 0);  
procedure Get(Item : out Num; Width : in Field := 0);
```

8

If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus sign if present or (for a signed type only) a minus sign if present, then reads the longest possible sequence of characters matching the syntax of a numeric literal without a point. If a nonzero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count.

9

Returns, in the parameter Item, the value of type Num that corresponds to the sequence input.

10

The exception `Data_Error` is propagated if the sequence of characters read does not form a legal integer literal or if the value obtained is not of the subtype `Num` (for `Integer_IO`) or is not in the base range of `Num` (for `Modular_IO`).

11

```
procedure Put(File : in File_Type;
              Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);

procedure Put(Item : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);
```

12

Outputs the value of the parameter `Item` as an integer literal, with no low lines, no exponent, and no leading zeros (but a single zero for the value zero), and a preceding minus sign for a negative value.

13

If the resulting sequence of characters to be output has fewer than `Width` characters, then leading spaces are first output to make up the difference.

14

Uses the syntax for decimal literal if the parameter `Base` has the value `ten` (either explicitly or through `Default_Base`); otherwise, uses the syntax for based literal, with any letters in upper case.

15

```
procedure Get(From : in String; Item : out Num; Last : out Positive);■
```

16

Reads an integer value from the beginning of the given string, following the same rules as the Get procedure that reads an integer value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

17

The exception Data_Error is propagated if the sequence input does not have the required syntax or if the value obtained is not of the subtype Num.

18

```
procedure Put(To    : out String;
              Item  : in Num;
              Base  : in Number_Base := Default_Base);
```

19

Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using the length of the given string as the value for Width.

20

Integer_Text_IO is a library package that is a nongeneric equivalent to Text_IO.Integer_IO for the predefined type Integer:

21

```
with Ada.Text_IO;
package Ada.Integer_Text_IO is new Ada.Text_IO.Integer_IO(Integer);
```

22

For each predefined signed integer type, a nongeneric equivalent to Text_IO.Integer_IO is provided, with names such as Ada.Long_Integer_Text_IO.

Implementation Permissions

23

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

NOTES

24

29 For `Modular_IO`, execution of `Get` propagates `Data_Error` if the sequence of characters read forms an integer literal outside the range `0..Num'Last`.

Examples

25/1

<This paragraph was deleted.>

26

```
package Int_IO is new Integer_IO(Small_Int); use Int_IO;
--< default format used at instantiation,>
--< Default_Width = 4, Default_Base = 10>
```

27

```
Put(126);           --< "b126">
Put(-126, 7);      --< "bbb-126">
Put(126, Width => 13, Base => 2); --< "bbb2#1111110#">
```

15.10.9 A.10.9 Input-Output for Real Types

Static Semantics

1

The following procedures are defined in the generic packages `Float_IO`, `Fixed_IO`, and `Decimal_IO`, which have to be instantiated for the appropriate floating point, ordinary fixed point, or decimal fixed point type respectively (indicated by `Num` in the specifications).

2

Values are output as decimal literals without low line characters. The format of each value output consists of a `Fore` field, a decimal point, an `Aft` field, and (if a nonzero `Exp` parameter is supplied) the letter `E` and an `Exp` field. The two possible formats thus correspond to:

3

`Fore . Aft`

4

and to:

5

`Fore . Aft E Exp`

6

without any spaces between these fields. The `Fore` field may include leading spaces, and a minus sign for negative values. The `Aft` field includes only decimal digits (possibly with trailing zeros). The `Exp` field includes the sign (plus or minus) and the exponent (possibly with leading zeros).

7

For floating point types, the default lengths of these fields are defined by the following variables that are declared in the generic package `Float_IO`:

8

```
Default_Fore : Field := 2;  
Default_Aft  : Field := Num'Digits-1;  
Default_Exp  : Field := 3;
```

9

For ordinary or decimal fixed point types, the default lengths of these fields are defined by the following variables that are declared in the generic packages Fixed_IO and Decimal_IO, respectively:

10

```
Default_Fore : Field := Num'Fore;  
Default_Aft  : Field := Num'Aft;  
Default_Exp  : Field := 0;
```

11

The following procedures are provided:

12

```
procedure Get(File : in File_Type; Item : out Num; Width : in Field := 0);  
procedure Get(Item : out Num; Width : in Field := 0);
```

13

If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads the longest possible sequence of characters matching the syntax of any of the following (see Section 3.4 [2.4], page 39):

14

- [+|-]numeric_literal

15

- [+|-]numeral.[exponent]

16

- [+|-].numeral[exponent]

17

- [+|-]base#based_numeral#[exponent]

18

- [+|-]base#.based_numeral#[exponent]

19

If a nonzero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count.

20

Returns in the parameter Item the value of type Num that corresponds to the sequence input, preserving the sign (positive if none has been specified) of a zero value if Num is a floating point type and Num'Signed_Zeros is True.

21

The exception Data_Error is propagated if the sequence input does not have the required syntax or if the value obtained is not of the subtype Num.

22

```
procedure Put(File : in File_Type;
             Item : in Num;
             Fore : in Field := Default_Fore;
             Aft  : in Field := Default_Aft;
             Exp  : in Field := Default_Exp);

procedure Put(Item : in Num;
             Fore : in Field := Default_Fore;
             Aft  : in Field := Default_Aft;
             Exp  : in Field := Default_Exp);
```

23

Outputs the value of the parameter Item as a decimal literal with the format defined by Fore, Aft and Exp. If the value is negative, or if Num is a floating point type where Num'Signed_Zeros is True and the value is a negatively signed zero, then a minus sign is included in the integer part. If Exp has the value zero, then the integer part to be output has as many digits as are needed to represent the integer part of the value of

Item, overriding Fore if necessary, or consists of the digit zero if the value of Item has no integer part.

24

If Exp has a value greater than zero, then the integer part to be output has a single digit, which is nonzero except for the value 0.0 of Item.

25

In both cases, however, if the integer part to be output has fewer than Fore characters, including any minus sign, then leading spaces are first output to make up the difference. The number of digits of the fractional part is given by Aft, or is one if Aft equals zero. The value is rounded; a value of exactly one half in the last place is rounded away from zero.

26

If Exp has the value zero, there is no exponent part. If Exp has a value greater than zero, then the exponent part to be output has as many digits as are needed to represent the exponent part of the value of Item (for which a single digit integer part is used), and includes an initial sign (plus or minus). If the exponent part to be output has fewer than Exp characters, including the sign, then leading zeros precede the digits, to make up the difference. For the value 0.0 of Item, the exponent has the value zero.

27

```
procedure Get(From : in String; Item : out Num; Last : out Positive);■
```

28

Reads a real value from the beginning of the given string, following the same rule as the Get procedure that reads a real value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Num that corresponds

to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

29

The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the value obtained is not of the subtype Num.

30

```
procedure Put(To    : out String;
              Item  : in Num;
              Aft   : in Field := Default_Aft;
              Exp   : in Field := Default_Exp);
```

31

Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using a value for Fore such that the sequence of characters output exactly fills the string, including any leading spaces.

32

Float_Text_IO is a library package that is a nongeneric equivalent to Text_IO.Float_IO for the predefined type Float:

33

```
with Ada.Text_IO;
package Ada.Float_Text_IO is new Ada.Text_IO.Float_IO(Float);
```

34

For each predefined floating point type, a nongeneric equivalent to Text_IO.Float_IO is provided, with names such as Ada.Long_Float_Text_IO.

Implementation Permissions

35

An implementation may extend Get and Put for floating point types to support special values such as infinities and NaNs.

36

The implementation of Put need not produce an output value with greater accuracy than is supported for the base subtype. The additional accuracy, if any, of the value produced by Put when the number of requested digits in the integer and fractional parts exceeds the required accuracy is implementation defined.

37

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

NOTES

38

30 For an item with a positive value, if output to a string exactly fills the string without leading spaces, then output of the corresponding negative value will propagate `Layout_Error`.

39

31 The rules for the `Value` attribute (see Section 4.5 [3.5], page 76) and the rules for `Get` are based on the same set of formats.

Examples

40/1

<This paragraph was deleted.>

41

```
package Real_IO is new Float_IO(Real); use Real_IO;
--< default format used at instantiation, Default_Exp = 3>
```

42

```
X : Real := -123.4567; --< digits 8      (see Section 4.5.7 [3.5.7],
page 103)>
```

43

```
Put(X); <-- default format>    <"-1.2345670E+02">
Put(X, Fore => 5, Aft => 3, Exp => 2);    <-- "bbb-1.235E+2">
Put(X, 5, 3, 0);                <-- "b-123.457">
```

15.10.10 A.10.10 Input-Output for Enumeration Types

Static Semantics

1

The following procedures are defined in the generic package `Enumeration_IO`, which has to be instantiated for the appropriate enumeration type (indicated by `Enum` in the specification).

2

Values are output using either upper or lower case letters for identifiers. This is specified by the parameter `Set`, which is of the enumeration type `Type_Set`.

3

```
type Type_Set is (Lower_Case, Upper_Case);
```

4

The format (which includes any trailing spaces) can be specified by an optional field width

parameter. The default field width and letter case are defined by the following variables that are declared in the generic package Enumeration_IO:

5

```
Default_Width   : Field := 0;  
Default_Setting : Type_Set := Upper_Case;
```

6

The following procedures are provided:

7

```
procedure Get(File : in File_Type; Item : out Enum);  
procedure Get(Item : out Enum);
```

8

After skipping any leading blanks, line terminators, or page terminators, reads an identifier according to the syntax of this lexical element (lower and upper case being considered equivalent), or a character literal according to the syntax of this lexical element (including the apostrophes). Returns, in the parameter Item, the value of type Enum that corresponds to the sequence input.

9

The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype Enum.

10

```
procedure Put(File   : in File_Type;  
              Item   : in Enum;  
              Width  : in Field := Default_Width;  
              Set    : in Type_Set := Default_Setting);  
  
procedure Put(Item   : in Enum;  
              Width  : in Field := Default_Width;  
              Set    : in Type_Set := Default_Setting);
```

11

Outputs the value of the parameter Item as an enumeration literal (either an identifier or

a character literal). The optional parameter `Set` indicates whether lower case or upper case is used for identifiers; it has no effect for character literals. If the sequence of characters produced has fewer than `Width` characters, then trailing spaces are finally output to make up the difference. If `Enum` is a character type, the sequence of characters produced is as for `Enum'Image(Item)`, as modified by the `Width` and `Set` parameters.

12

```
procedure Get(From : in String; Item : out Enum; Last : out Positive);■
```

13

Reads an enumeration value from the beginning of the given string, following the same rule as the `Get` procedure that reads an enumeration value from a file, but treating the end of the string as a file terminator. Returns, in the parameter `Item`, the value of type `Enum` that corresponds to the sequence input. Returns in `Last` the index value such that `From(Last)` is the last character read.

14

The exception `Data_Error` is propagated if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype `Enum`.

15

```
procedure Put(To    : out String;
              Item  : in Enum;
              Set   : in Type_Set := Default_Setting);
```

16

Outputs the value of the parameter `Item` to the given string, following the same rule as for output to a file, using the length of the given string as the value for `Width`.

17/1

Although the specification of the generic package `Enumeration_IO` would allow instantiation

for an integer type, this is not the intended purpose of this generic package, and the effect of such instantiations is not defined by the language.

NOTES

18

32 There is a difference between Put defined for characters, and for enumeration values. Thus

19

```
Ada.Text_IO.Put('A'); --< outputs the character A>
```

20

```
package Char_IO is new Ada.Text_IO.Enumeration_IO(Character);  
Char_IO.Put('A'); --< outputs the character 'A', between apostrophes>
```

21

33 The type Boolean is an enumeration type, hence Enumeration_IO can be instantiated for this type.

15.10.11 A.10.11 Input-Output for Bounded Strings

1/2

The package Text_IO.Bounded_IO provides input–output in human–readable form for Bounded_Strings.

Static Semantics

2/2

The generic library package Text_IO.Bounded_IO has the following declaration:

3/2

```
with Ada.Strings.Bounded;  
generic  
  with package Bounded is  
    new Ada.Strings.Bounded.Generic_Bounded_Length (<>);  
package Ada.Text_IO.Bounded_IO is
```

4/2

```
  procedure Put  
    (File : in File_Type;  
     Item : in Bounded.Bounded_String);
```

5/2

```
  procedure Put  
    (Item : in Bounded.Bounded_String);
```

6/2

```
procedure Put_Line
  (File : in File_Type;
   Item : in Bounded.Bounded_String);
```

7/2

```
procedure Put_Line
  (Item : in Bounded.Bounded_String);
```

8/2

```
function Get_Line
  (File : in File_Type)
  return Bounded.Bounded_String;
```

9/2

```
function Get_Line
  return Bounded.Bounded_String;
```

10/2

```
procedure Get_Line
  (File : in File_Type; Item : out Bounded.Bounded_String);
```

11/2

```
procedure Get_Line
  (Item : out Bounded.Bounded_String);
```

12/2

```
end Ada.Text_IO.Bounded_IO;
```

13/2

For an item of type Bounded_String, the following subprograms are provided:

14/2

```
procedure Put
  (File : in File_Type;
   Item : in Bounded.Bounded_String);
```

15/2

Equivalent to Text_IO.Put (File,
Bounded.To_String(Item));

16/2

```
procedure Put
```

```

      (Item : in Bounded.Bounded_String);
17/2

      Equivalent to Text_IO.Put
      (Bounded.To_String(Item));
18/2

procedure Put_Line
  (File : in File_Type;
   Item : in Bounded.Bounded_String);
19/2

      Equivalent to Text_IO.Put_Line (File,
      Bounded.To_String(Item));
20/2

procedure Put_Line
  (Item : in Bounded.Bounded_String);
21/2

      Equivalent to Text_IO.Put_Line
      (Bounded.To_String(Item));
22/2

function Get_Line
  (File : in File_Type)
  return Bounded.Bounded_String;
23/2

      Returns Bounded.To_Bounded_String(Text_IO.Get_Line(File));
24/2

function Get_Line
  return Bounded.Bounded_String;
25/2

      Returns Bounded.To_Bounded_String(Text_IO.Get_Line);
26/2

procedure Get_Line
  (File : in File_Type; Item : out Bounded.Bounded_String);
27/2

```

Equivalent to Item := Get_Line (File);

28/2

```
procedure Get_Line
  (Item : out Bounded.Bounded_String);
```

29/2

Equivalent to Item := Get_Line;

15.10.12 A.10.12 Input-Output for Unbounded Strings

1/2

The package Text_IO.Unbounded_IO provides input–output in human–readable form for Unbounded_Strings.

Static Semantics

2/2

The library package Text_IO.Unbounded_IO has the following declaration:

3/2

```
with Ada.Strings.Unbounded;
package Ada.Text_IO.Unbounded_IO is
```

4/2

```
  procedure Put
    (File : in File_Type;
     Item : in Strings.Unbounded.Unbounded_String);
```

5/2

```
  procedure Put
    (Item : in Strings.Unbounded.Unbounded_String);
```

6/2

```
  procedure Put_Line
    (File : in File_Type;
     Item : in Strings.Unbounded.Unbounded_String);
```

7/2

```
  procedure Put_Line
    (Item : in Strings.Unbounded.Unbounded_String);
```

8/2

```
  function Get_Line
    (File : in File_Type)
    return Strings.Unbounded.Unbounded_String;
```

9/2

```
function Get_Line  
    return Strings.Unbounded.Unbounded_String;
```

10/2

```
procedure Get_Line  
    (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);
```

11/2

```
procedure Get_Line  
    (Item : out Strings.Unbounded.Unbounded_String);
```

12/2

```
end Ada.Text_IO.Unbounded_IO;
```

13/2

For an item of type Unbounded_String, the following subprograms are provided:

14/2

```
procedure Put  
    (File : in File_Type;  
     Item : in Strings.Unbounded.Unbounded_String);
```

15/2

Equivalent to Text_IO.Put (File,
Strings.Unbounded.To_String(Item));

16/2

```
procedure Put  
    (Item : in Strings.Unbounded.Unbounded_String);
```

17/2

Equivalent to Text_IO.Put
(Strings.Unbounded.To_String(Item));

18/2

```
procedure Put_Line  
    (File : in File_Type;  
     Item : in Strings.Unbounded.Unbounded_String);
```

19/2

Equivalent to Text_IO.Put_Line (File,
Strings.Unbounded.To_String(Item));

20/2

```
procedure Put_Line  
  (Item : in Strings.Unbounded.Unbounded_String);
```

21/2

Equivalent to Text_IO.Put_Line
(Strings.Unbounded.To_String(Item));

22/2

```
function Get_Line  
  (File : in File_Type)  
  return Strings.Unbounded.Unbounded_String;
```

23/2

Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line(File));

24/2

```
function Get_Line  
  return Strings.Unbounded.Unbounded_String;
```

25/2

Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line);

26/2

```
procedure Get_Line  
  (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);■
```

27/2

Equivalent to Item := Get_Line (File);

28/2

```
procedure Get_Line  
  (Item : out Strings.Unbounded.Unbounded_String);
```

29/2

Equivalent to Item := Get_Line;

15.11 A.11 Wide Text Input-Output and Wide Wide Text Input-Output

1/2

The packages `Wide_Text_IO` and `Wide_Wide_Text_IO` provide facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of wide characters (or wide wide characters) grouped into lines, and as a sequence of lines grouped into pages.

Static Semantics

2/2

The specification of package `Wide_Text_IO` is the same as that for `Text_IO`, except that in each `Get`, `Look_Ahead`, `Get_Immediate`, `Get_Line`, `Put`, and `Put_Line` subprogram, any occurrence of `Character` is replaced by `Wide_Character`, and any occurrence of `String` is replaced by `Wide_String`. Nongeneric equivalents of `Wide_Text_IO.Integer_IO` and `Wide_Text_IO.Float_IO` are provided (as for `Text_IO`) for each predefined numeric type, with names such as `Ada.Integer_Wide_Text_IO`, `Ada.Long_Integer_Wide_Text_IO`, `Ada.Float_Wide_Text_IO`, `Ada.Long_Float_Wide_Text_IO`.

3/2

The specification of package `Wide_Wide_Text_IO` is the same as that for `Text_IO`, except that in each `Get`, `Look_Ahead`, `Get_Immediate`, `Get_Line`, `Put`, and `Put_Line` subprogram, any occurrence of `Character` is replaced by `Wide_Wide_Character`, and any occurrence of `String` is replaced by `Wide_Wide_String`. Nongeneric equivalents of `Wide_Wide_Text_IO.Integer_IO` and `Wide_Wide_Text_IO.Float_IO` are provided (as for `Text_IO`) for each predefined numeric type, with names such as `Ada.Integer_Wide_Wide_Text_IO`, `Ada.Long_Integer_Wide_Wide_Text_IO`, `Ada.Float_Wide_Wide_Text_IO`, `Ada.Long_Float_Wide_Wide_Text_IO`.

4/2

The specification of package `Wide_Text_IO.Wide_Bounded_IO` is the same as that for `Text_IO.Bounded_IO`, except that any occurrence of `Bounded_String` is replaced by `Wide_Bounded_String`, and any occurrence of package `Bounded` is replaced by `Wide_Bounded`. The specification of package `Wide_Wide_Text_IO.Wide_Wide_Bounded_IO` is the same as that for `Text_IO.Bounded_IO`, except that any occurrence of `Bounded_String` is replaced by `Wide_Wide_Bounded_String`, and any occurrence of package `Bounded` is replaced by `Wide_Wide_Bounded`.

5/2

The specification of package `Wide_Text_IO.Wide_Unbounded_IO` is the same as that for `Text_IO.Unbounded_IO`, except that any occurrence of `Unbounded_String` is replaced by `Wide_Unbounded_String`, and any occurrence of package `Unbounded` is replaced by `Wide_Unbounded`. The specification of package `Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO` is the same as that for `Text_IO.Unbounded_IO`, except that any occurrence of `Unbounded_String` is replaced by `Wide_Wide_Unbounded_String`, and any occurrence of package `Unbounded` is replaced by `Wide_Wide_Unbounded`.

15.12 A.12 Stream Input-Output

1/2

The packages `Streams.Stream_IO`, `Text_IO.Text_Streams`, `Wide_Text_IO.Text_Streams`, and `Wide_Wide_Text_IO.Text_Streams` provide stream-oriented operations on files.

15.12.1 A.12.1 The Package Streams.Stream_IO

1

The subprograms in the child package Streams.Stream_IO provide control over stream files. Access to a stream file is either sequential, via a call on Read or Write to transfer an array of stream elements, or positional (if supported by the implementation for the given file), by specifying a relative index for an element. Since a stream file can be converted to a Stream_Access value, calling stream-oriented attribute subprograms of different element types with the same Stream_Access value provides heterogeneous input-output. See Section 14.13 [13.13], page 538, for a general discussion of streams.

Static Semantics

1.1/1

The elements of a stream file are stream elements. If positioning is supported for the specified external file, a current index and current size are maintained for the file as described in Section 15.8 [A.8], page 682. If positioning is not supported, a current index is not maintained, and the current size is implementation defined.

2

The library package Streams.Stream_IO has the following declaration:

3

```
with Ada.IO_Exceptions;  
package Ada.Streams.Stream_IO is
```

4

```
    type Stream_Access is access all Root_Stream_Type'Class;
```

5

```
    type File_Type is limited private;
```

6

```
    type File_Mode is (In_File, Out_File, Append_File);
```

7

```
    type Count is range 0 .. <implementation-defined>;  
    subtype Positive_Count is Count range 1 .. Count'Last;  
    -- <Index into file, in stream elements.>
```

8

```
    procedure Create (File : in out File_Type;  
                    Mode : in File_Mode := Out_File;  
                    Name : in String   := "";  
                    Form : in String   := "");
```

9

```
    procedure Open (File : in out File_Type;
```

```

Mode : in File_Mode;
Name : in String;
Form : in String := "";
10

procedure Close (File : in out File_Type);
procedure Delete (File : in out File_Type);
procedure Reset (File : in out File_Type; Mode : in File_Mode);
procedure Reset (File : in out File_Type);
11

function Mode (File : in File_Type) return File_Mode;
function Name (File : in File_Type) return String;
function Form (File : in File_Type) return String;
12

function Is_Open (File : in File_Type) return Boolean;
function End_Of_File (File : in File_Type) return Boolean;
13

function Stream (File : in File_Type) return Stream_Access;
-- <Return stream access for use with T'Input and T'Output>
14/1

<This paragraph was deleted.>
15

-- <Read array of stream elements from file>
procedure Read (File : in File_Type;
Item : out Stream_Element_Array;
Last : out Stream_Element_Offset;
From : in Positive_Count);
16

procedure Read (File : in File_Type;
Item : out Stream_Element_Array;
Last : out Stream_Element_Offset);
17/1

<This paragraph was deleted.>
18

-- <Write array of stream elements into file>

```

```

19      procedure Write (File : in File_Type;
                       Item  : in Stream_Element_Array;
                       To    : in Positive_Count);

20/1      procedure Write (File : in File_Type;
                       Item  : in Stream_Element_Array);

21      <This paragraph was deleted.>

22      -- <Operations on position within file>

23      procedure Set_Index(File : in File_Type; To : in Positive_Count);█

24      function Index(File : in File_Type) return Positive_Count;
      function Size (File : in File_Type) return Count;

25/1      procedure Set_Mode(File : in out File_Type; Mode : in File_Mode);█

26      procedure Flush(File : in File_Type);

27      -- <exceptions>
      Status_Error : exception renames IO_Exceptions.Status_Error;
      Mode_Error   : exception renames IO_Exceptions.Mode_Error;
      Name_Error   : exception renames IO_Exceptions.Name_Error;
      Use_Error    : exception renames IO_Exceptions.Use_Error;
      Device_Error : exception renames IO_Exceptions.Device_Error;
      End_Error    : exception renames IO_Exceptions.End_Error;
      Data_Error   : exception renames IO_Exceptions.Data_Error;

      private
      ... -- <not specified by the language>
      end Ada.Streams.Stream_IO;

```

27.1/2

The type File_Type needs finalization (see Section 8.6 [7.6], page 295).

28/2

The subprograms given in subclause Section 15.8.2 [A.8.2], page 685, for the control of external files (Create, Open, Close, Delete, Reset, Mode, Name, Form, and Is_Open) are available for stream files.

28.1/2

The End_Of_File function:

28.2/2

- Propagates Mode_Error if the mode of the file is not In_File;

28.3/2

- If positioning is supported for the given external file, the function returns True if the current index exceeds the size of the external file; otherwise it returns False;

28.4/2

- If positioning is not supported for the given external file, the function returns True if no more elements can be read from the given file; otherwise it returns False.

28.5/2

The Set_Mode procedure sets the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

28.6/1

The Flush procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file or changing the position. Mode_Error is propagated if the mode of the file is In_File.

29/1

The Stream function returns a Stream_Access result from a File_Type object, thus allowing the stream-oriented attributes Read, Write, Input, and Output to be used on the same file for multiple types. Stream propagates Status_Error if File is not open.

30/2

The procedures Read and Write are equivalent to the corresponding operations in the package Streams. Read propagates Mode_Error if the mode of File is not In_File. Write propagates Mode_Error if the mode of File is not Out_File or Append_File. The Read procedure with a Positive_Count parameter starts reading at the specified index. The Write procedure with a Positive_Count parameter starts writing at the specified index. For a file that supports positioning, Read without a Positive_Count parameter starts reading at the current index, and Write without a Positive_Count parameter starts writing at the current index.

30.1/1

The Size function returns the current size of the file.

31/1

The Index function returns the current index.

32

The Set_Index procedure sets the current index to the specified value.

32.1/1

If positioning is supported for the external file, the current index is maintained as follows:

32.2/1

- For Open and Create, if the Mode parameter is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one.

32.3/1

- For Reset, if the Mode parameter is Append_File, or no Mode parameter is given and the current mode is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one.

32.4/1

- For Set_Mode, if the new mode is Append_File, the current index is set to current size plus one; otherwise, the current index is unchanged.

32.5/1

- For Read and Write without a Positive_Count parameter, the current index is incremented by the number of stream elements read or written.

32.6/1

- For Read and Write with a Positive_Count parameter, the value of the current index is set to the value of the Positive_Count parameter plus the number of stream elements read or written.

33

If positioning is not supported for the given file, then a call of Index or Set_Index propagates Use_Error. Similarly, a call of Read or Write with a Positive_Count parameter propagates Use_Error.

<Paragraphs 34 through 36 were deleted.>

Erroneous Execution

36.1/1

If the File_Type object passed to the Stream function is later closed or finalized, and the stream-oriented attributes are subsequently called (explicitly or implicitly) on the Stream_Access value returned by Stream, execution is erroneous. This rule applies even if the File_Type object was opened again after it had been closed.

15.12.2 A.12.2 The Package Text_IO.Text_Streams

1

The package Text_IO.Text_Streams provides a function for treating a text file as a stream.

Static Semantics

2

The library package Text_IO.Text_Streams has the following declaration:

3

```
with Ada.Streams;  
package Ada.Text_IO.Text_Streams is  
    type Stream_Access is access all Streams.Root_Stream_Type'Class;
```

4

```
        function Stream (File : in File_Type) return Stream_Access;  
    end Ada.Text_IO.Text_Streams;
```

5

The Stream function has the same effect as the corresponding function in Streams.Stream_IO.

NOTES

6

34 The ability to obtain a stream for a text file allows Current_Input, Current_Output, and Current_Error to be processed with the functionality of streams, including the mixing of text and binary input–output, and the mixing of binary input–output for different types.

7

35 Performing operations on the stream associated with a text file does not affect the column, line, or page counts.

15.12.3 A.12.3 The Package Wide_Text_IO.Text_Streams

1

The package Wide_Text_IO.Text_Streams provides a function for treating a wide text file as a stream.

Static Semantics

2

The library package Wide_Text_IO.Text_Streams has the following declaration:

3

```
with Ada.Streams;  
package Ada.Wide_Text_IO.Text_Streams is  
    type Stream_Access is access all Streams.Root_Stream_Type'Class;
```

4

```
        function Stream (File : in File_Type) return Stream_Access;  
    end Ada.Wide_Text_IO.Text_Streams;
```

5

The Stream function has the same effect as the corresponding function in Streams.Stream_IO.

15.12.4 A.12.4 The Package Wide_Wide_Text_IO.Text_Streams

1/2

The package Wide_Wide_Text_IO.Text_Streams provides a function for treating a wide wide text file as a stream.

Static Semantics

2/2

The library package Wide_Wide_Text_IO.Text_Streams has the following declaration:

3/2

```
with Ada.Streams;  
package Ada.Wide_Wide_Text_IO.Text_Streams is  
    type Stream_Access is access all Streams.Root_Stream_Type'Class;
```

4/2

```
    function Stream (File : in File_Type) return Stream_Access;  
end Ada.Wide_Wide_Text_IO.Text_Streams;
```

5/2

The Stream function has the same effect as the corresponding function in Streams.Stream_IO.

15.13 A.13 Exceptions in Input-Output

1

The package IO_Exceptions defines the exceptions needed by the predefined input-output packages.

Static Semantics

2

The library package IO_Exceptions has the following declaration:

3

```
package Ada.IO_Exceptions is  
    pragma Pure(IO_Exceptions);
```

4

```
    Status_Error : exception;  
    Mode_Error   : exception;  
    Name_Error   : exception;  
    Use_Error    : exception;  
    Device_Error : exception;  
    End_Error    : exception;  
    Data_Error   : exception;  
    Layout_Error : exception;
```

5

```
end Ada.IO_Exceptions;
```


6

If more than one error condition exists, the corresponding exception that appears earliest in the following list is the one that is propagated.

7

The exception `Status_Error` is propagated by an attempt to operate upon a file that is not open, and by an attempt to open a file that is already open.

8

The exception `Mode_Error` is propagated by an attempt to read from, or test for the end of, a file whose current mode is `Out_File` or `Append_File`, and also by an attempt to write to a file whose current mode is `In_File`. In the case of `Text_IO`, the exception `Mode_Error` is also propagated by specifying a file whose current mode is `Out_File` or `Append_File` in a call of `Set_Input`, `Skip_Line`, `End_Of_Line`, `Skip_Page`, or `End_Of_Page`; and by specifying a file whose current mode is `In_File` in a call of `Set_Output`, `Set_Line_Length`, `Set_Page_Length`, `Line_Length`, `Page_Length`, `New_Line`, or `New_Page`.

9

The exception `Name_Error` is propagated by a call of `Create` or `Open` if the string given for the parameter `Name` does not allow the identification of an external file. For example, this exception is propagated if the string is improper, or, alternatively, if either none or more than one external file corresponds to the string.

10

The exception `Use_Error` is propagated if an operation is attempted that is not possible for reasons that depend on characteristics of the external file. For example, this exception is propagated by the procedure `Create`, among other circumstances, if the given mode is `Out_File` but the form specifies an input only device, if the parameter `Form` specifies invalid access rights, or if an external file with the given name already exists and overwriting is not allowed.

11

The exception `Device_Error` is propagated if an input–output operation cannot be completed because of a malfunction of the underlying system.

12

The exception `End_Error` is propagated by an attempt to skip (read past) the end of a file.

13

The exception `Data_Error` can be propagated by the procedure `Read` (or by the `Read` attribute) if the element read cannot be interpreted as a value of the required subtype. This exception is also propagated by a procedure `Get` (defined in the package `Text_IO`) if the input character sequence fails to satisfy the required syntax, or if the value input does not belong to the range of the required subtype.

14

The exception `Layout_Error` is propagated (in text input–output) by `Col`, `Line`, or `Page` if the value returned exceeds `Count'Last`. The exception `Layout_Error` is also propagated on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, respectively (excluding the unbounded cases). It is also propagated by an attempt to `Put` too many characters to a string.

Documentation Requirements

15

The implementation shall document the conditions under which `Name_Error`, `Use_Error` and `Device_Error` are propagated.

Implementation Permissions

16

If the associated check is too complex, an implementation need not propagate `Data_Error` as part of a procedure `Read` (or the `Read` attribute) if the value read cannot be interpreted as a value of the required subtype.

Erroneous Execution

17

If the element read by the procedure `Read` (or by the `Read` attribute) cannot be interpreted as a value of the required subtype, but this is not detected and `Data_Error` is not propagated, then the resulting value can be abnormal, and subsequent references to the value can lead to erroneous execution, as explained in Section 14.9.1 [13.9.1], page 522.

15.14 A.14 File Sharing

Dynamic Semantics

1

It is not specified by the language whether the same external file can be associated with more than one file object. If such sharing is supported by the implementation, the following effects are defined:

2

- Operations on one text file object do not affect the column, line, and page numbers of any other file object.

3/1

- <This paragraph was deleted.>

4

- For direct and stream files, the current index is a property of each file object; an operation on one file object does not affect the current index of any other file object.

5

- For direct and stream files, the current size of the file is a property of the external file.

6

All other effects are identical.

15.15 A.15 The Package `Command_Line`

1

The package `Command_Line` allows a program to obtain the values of its arguments and to set the exit status code to be returned on normal termination.

Static Semantics

2

The library package Ada.Command_Line has the following declaration:

3

```
package Ada.Command_Line is
  pragma Preelaborate(Command_Line);
```

4

```
  function Argument_Count return Natural;
```

5

```
  function Argument (Number : in Positive) return String;
```

6

```
  function Command_Name return String;
```

7

```
  type Exit_Status is <implementation-defined integer type>;
```

8

```
  Success : constant Exit_Status;
```

```
  Failure : constant Exit_Status;
```

9

```
  procedure Set_Exit_Status (Code : in Exit_Status);
```

10

```
private
```

```
  ... -- <not specified by the language>
```

```
end Ada.Command_Line;
```

11

```
function Argument_Count return Natural;
```

12

If the external execution environment supports passing arguments to a program, then `Argument_Count` returns the number of arguments passed to the program invoking the

function. Otherwise it returns 0. The meaning of "number of arguments" is implementation defined.

13

```
function Argument (Number : in Positive) return String;
```

14

If the external execution environment supports passing arguments to a program, then `Argument` returns an implementation-defined value corresponding to the argument at relative position `Number`. If `Number` is outside the range `1..Argument_Count`, then `Constraint_Error` is propagated.

15

```
function Command_Name return String;
```

16

If the external execution environment supports passing arguments to a program, then `Command_Name` returns an implementation-defined value corresponding to the name of the command invoking the program; otherwise `Command_Name` returns the null string.

16.1/1

```
type Exit_Status is <implementation-defined integer type>;
```

17

The type `Exit_Status` represents the range of exit status values supported by the external execution environment. The constants `Success` and `Failure` correspond to success and failure, respectively.

18

```
procedure Set_Exit_Status (Code : in Exit_Status);
```

19

If the external execution environment supports returning an exit status from a program, then `Set_Exit_Status` sets `Code` as the status. Normal termination of a program returns as the exit status the value most recently set by `Set_Exit_Status`, or, if no such value has been set, then the value `Success`. If a program terminates abnormally, the status set by `Set_Exit_Status` is ignored, and an implementation-defined exit status value is set.

20

If the external execution environment does not support returning an exit value from a program, then `Set_Exit_Status` does nothing.

Implementation Permissions

21

An alternative declaration is allowed for package `Command_Line` if different functionality is appropriate for the external execution environment.

NOTES

22

36 `Argument_Count`, `Argument`, and `Command_Name` correspond to the C language's `argc`, `argv[n]` (for $n > 0$) and `argv[0]`, respectively.

15.16 A.16 The Package Directories

1/2

The package `Directories` provides operations for manipulating files and directories, and their names.

Static Semantics

2/2

The library package `Directories` has the following declaration:

3/2

```
with Ada.IO_Exceptions;  
with Ada.Calendar;  
package Ada.Directories is
```

4/2

```
-- <Directory and file operations:>
```

5/2

```
function Current_Directory return String;
```

6/2

```
procedure Set_Directory (Directory : in String);
```

7/2

```
procedure Create_Directory (New_Directory : in String;  
                           Form           : in String := "");
```

8/2

```
procedure Delete_Directory (Directory : in String);
```

9/2

```
procedure Create_Path (New_Directory : in String;  
                      Form           : in String := "");
```

10/2

```
procedure Delete_Tree (Directory : in String);
```

11/2

```
procedure Delete_File (Name : in String);
```

12/2

```
procedure Rename (Old_Name, New_Name : in String);
```

13/2

```
procedure Copy_File (Source_Name,  
                    Target_Name : in String;  
                    Form        : in String := "");
```

14/2

```
-- <File and directory name operations:>
```

15/2

```
function Full_Name (Name : in String) return String;
```

16/2

```
function Simple_Name (Name : in String) return String;
```

17/2

```
function Containing_Directory (Name : in String) return String;
```

18/2

```
function Extension (Name : in String) return String;
```

19/2

```
function Base_Name (Name : in String) return String;
```

20/2

```
function Compose (Containing_Directory : in String := "";
                  Name                 : in String;
                  Extension              : in String := "") return String;■
```

21/2

```
-- <File and directory queries:>
```

22/2

```
type File_Kind is (Directory, Ordinary_File, Special_File);
```

23/2

```
type File_Size is range 0 .. <implementation-defined>;
```

24/2

```
function Exists (Name : in String) return Boolean;
```

25/2

```
function Kind (Name : in String) return File_Kind;
```

26/2

```
function Size (Name : in String) return File_Size;
```

27/2

```
function Modification_Time (Name : in String) return Ada.Calendar.Time;■
```

28/2

```
-- <Directory searching:>
```

29/2

```
type Directory_Entry_Type is limited private;
```

30/2

```
type Filter_Type is array (File_Kind) of Boolean;
```

31/2

```
type Search_Type is limited private;
```

32/2

```
procedure Start_Search (Search      : in out Search_Type;
                        Directory    : in String;
                        Pattern      : in String;
                        Filter       : in Filter_Type := (others => True));
```

33/2

```
procedure End_Search (Search : in out Search_Type);
```

34/2

```
function More_Entries (Search : in Search_Type) return Boolean;
```

35/2

```
procedure Get_Next_Entry (Search : in out Search_Type;
                          Directory_Entry : out Directory_Entry_Type);
```

36/2

```
procedure Search (
    Directory : in String;
    Pattern   : in String;
    Filter    : in Filter_Type := (others => True);
    Process   : not null access procedure (
        Directory_Entry : in Directory_Entry_Type));
```

37/2

```
-- <Operations on Directory Entries:>
```

38/2

```
function Simple_Name (Directory_Entry : in Directory_Entry_Type)
    return String;
```

39/2

```
function Full_Name (Directory_Entry : in Directory_Entry_Type)
    return String;
```

40/2

```
function Kind (Directory_Entry : in Directory_Entry_Type)
    return File_Kind;
```


41/2

```
function Size (Directory_Entry : in Directory_Entry_Type)
    return File_Size;
```

42/2

```
function Modification_Time (Directory_Entry : in Directory_Entry_Type)
    return Ada.Calendar.Time;
```

43/2

```
Status_Error : exception renames Ada.IO_Exceptions.Status_Error;
Name_Error   : exception renames Ada.IO_Exceptions.Name_Error;
Use_Error    : exception renames Ada.IO_Exceptions.Use_Error;
Device_Error : exception renames Ada.IO_Exceptions.Device_Error;
```

44/2

```
private
    -- <Not specified by the language.>
end Ada.Directories;
```

45/2

External files may be classified as directories, special files, or ordinary files. A <directory> is an external file that is a container for files on the target system. A <special file> is an external file that cannot be created or read by a predefined Ada input–output package. External files that are not special files or directories are called <ordinary files>.

46/2

A <file name> is a string identifying an external file. Similarly, a <directory name> is a string identifying a directory. The interpretation of file names and directory names is implementation–defined.

47/2

The <full name> of an external file is a full specification of the name of the file. If the external environment allows alternative specifications of the name (for example, abbreviations), the full name should not use such alternatives. A full name typically will include the names of all of the directories that contain the item. The <simple name> of an external file is the name of the item, not including any containing directory names. Unless otherwise specified, a file name or directory name parameter in a call to a predefined Ada input–output subprogram can be a full name, a simple name, or any other form of name supported by the implementation.

48/2

The <default directory> is the directory that is used if a directory or file name is not a full name (that is, when the name does not fully identify all of the containing directories).

49/2

A <directory entry> is a single item in a directory, identifying a single external file (including directories and special files).

50/2

For each function that returns a string, the lower bound of the returned value is 1.

51/2

The following file and directory operations are provided:

52/2

```
function Current_Directory return String;
```

53/2

Returns the full directory name for the current default directory. The name returned shall be suitable for a future call to Set_Directory. The exception Use_Error is propagated if a default directory is not supported by the external environment.

54/2

```
procedure Set_Directory (Directory : in String);
```

55/2

Sets the current default directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support making Directory (in the absence of Name_Error) a default directory.

56/2

```
procedure Create_Directory (New_Directory : in String;  
                           Form           : in String := "");
```

57/2

Creates a directory with name New_Directory. The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation-defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of a directory. The

exception `Use_Error` is propagated if the external environment does not support the creation of a directory with the given name (in the absence of `Name_Error`) and form.

58/2

```
procedure Delete_Directory (Directory : in String);
```

59/2

Deletes an existing empty directory with name `Directory`. The exception `Name_Error` is propagated if the string given as `Directory` does not identify an existing directory. The exception `Use_Error` is propagated if the external environment does not support the deletion of the directory (or some portion of its contents) with the given name (in the absence of `Name_Error`).

60/2

```
procedure Create_Path (New_Directory : in String;  
                      Form           : in String := "");
```

61/2

Creates zero or more directories with name `New_Directory`. Each non-existent directory named by `New_Directory` is created. For example, on a typical Unix system, `Create_Path ("/usr/me/my");` would create directory `"me"` in directory `"usr"`, then create directory `"my"` in directory `"me"`. The `Form` parameter can be used to give system-dependent characteristics of the directory; the interpretation of the `Form` parameter is implementation-defined. A null string for `Form` specifies the use of the default options of the implementation of the new directory. The exception `Name_Error` is propagated if the string given as `New_Directory` does not allow the identification of any directory. The exception `Use_Error` is propagated if the external environment does not support the creation of any directories with the given name (in the absence of `Name_Error`) and form.

62/2

```
procedure Delete_Tree (Directory : in String);
```

63/2

Deletes an existing directory with name Directory. The directory and all of its contents (possibly including other directories) are deleted. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory or some portion of its contents with the given name (in the absence of Name_Error). If Use_Error is propagated, it is unspecified whether a portion of the contents of the directory is deleted.

64/2

```
procedure Delete_File (Name : in String);
```

65/2

Deletes an existing ordinary or special file with name Name. The exception Name_Error is propagated if the string given as Name does not identify an existing ordinary or special external file. The exception Use_Error is propagated if the external environment does not support the deletion of the file with the given name (in the absence of Name_Error).

66/2

```
procedure Rename (Old_Name, New_Name : in String);
```

67/2

Renames an existing external file (including directories) with name Old_Name to New_Name. The exception Name_Error is propagated if the string given as Old_Name does not identify an existing external file. The exception Use_Error is propagated if the external environment does not support the

renaming of the file with the given name (in the absence of `Name_Error`). In particular, `Use_Error` is propagated if a file or directory already exists with name `New_Name`.

68/2

```
procedure Copy_File (Source_Name,  
                    Target_Name : in String;  
                    Form        : in String);
```

69/2

Copies the contents of the existing external file with name `Source_Name` to an external file with name `Target_Name`. The resulting external file is a duplicate of the source external file. The `Form` parameter can be used to give system-dependent characteristics of the resulting external file; the interpretation of the `Form` parameter is implementation-defined. Exception `Name_Error` is propagated if the string given as `Source_Name` does not identify an existing external ordinary or special file, or if the string given as `Target_Name` does not allow the identification of an external file. The exception `Use_Error` is propagated if the external environment does not support creating the file with the name given by `Target_Name` and form given by `Form`, or copying of the file with the name given by `Source_Name` (in the absence of `Name_Error`).

70/2

The following file and directory name operations are provided:

71/2

```
function Full_Name (Name : in String) return String;
```

72/2

Returns the full name corresponding to the file name specified by `Name`. The exception `Name_Error` is propagated if the string given as `Name` does not allow the identification of an external file (including directories and special files).

73/2

```
function Simple_Name (Name : in String) return String;
```

74/2

Returns the simple name portion of the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

75/2

```
function Containing_Directory (Name : in String) return String;
```

76/2

Returns the name of the containing directory of the external file (including directories) identified by Name. (If more than one directory can contain Name, the directory name returned is implementation-defined.) The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if the external file does not have a containing directory.

77/2

```
function Extension (Name : in String) return String;
```

78/2

Returns the extension name corresponding to Name. The extension name is a portion of a simple name (not including any separator characters), typically used to identify the file class. If the external environment does not have extension names, then the null string is returned. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file.

79/2

```
function Base_Name (Name : in String) return String;
```

80/2

Returns the base name corresponding to Name. The base name is the remainder of a simple name after removing any extension and extension separators. The exception Name.Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

81/2

```
function Compose (Containing_Directory : in String := "";  
                 Name                 : in String;  
                 Extension             : in String := "") return String;■
```

82/2

Returns the name of the external file with the specified Containing_Directory, Name, and Extension. If Extension is the null string, then Name is interpreted as a simple name; otherwise Name is interpreted as a base name. The exception Name.Error is propagated if the string given as Containing_Directory is not null and does not allow the identification of a directory, or if the string given as Extension is not null and is not a possible extension, or if the string given as Name is not a possible simple name (if Extension is null) or base name (if Extension is non-null).

83/2

The following file and directory queries and types are provided:

84/2

```
type File_Kind is (Directory, Ordinary_File, Special_File);
```

85/2

The type File_Kind represents the kind of file represented by an external file or directory.

86/2

```
type File_Size is range 0 .. <implementation-defined>;
```

87/2

The type `File_Size` represents the size of an external file.

88/2

```
function Exists (Name : in String) return Boolean;
```

89/2

Returns `True` if an external file represented by `Name` exists, and `False` otherwise. The exception `Name_Error` is propagated if the string given as `Name` does not allow the identification of an external file (including directories and special files).

90/2

```
function Kind (Name : in String) return File_Kind;
```

91/2

Returns the kind of external file represented by `Name`. The exception `Name_Error` is propagated if the string given as `Name` does not allow the identification of an existing external file.

92/2

```
function Size (Name : in String) return File_Size;
```

93/2

Returns the size of the external file represented by `Name`. The size of an external file is the number of stream elements contained in the file. If the external file is not an ordinary file, the result is implementation-defined. The exception `Name_Error` is propagated if the string given as `Name` does not allow the identification of an existing external file. The exception `Constraint_Error` is propagated if the file size is not a value of type `File_Size`.

94/2

```
function Modification_Time (Name : in String) return Ada.Calendar.Time;■
```


95/2

Returns the time that the external file represented by `Name` was most recently modified. If the external file is not an ordinary file, the result is implementation-defined. The exception `Name_Error` is propagated if the string given as `Name` does not allow the identification of an existing external file. The exception `Use_Error` is propagated if the external environment does not support reading the modification time of the file with the name given by `Name` (in the absence of `Name_Error`).

96/2

The following directory searching operations and types are provided:

97/2

```
type Directory_Entry_Type is limited private;
```

98/2

The type `Directory_Entry_Type` represents a single item in a directory. These items can only be created by the `Get_Next_Entry` procedure in this package. Information about the item can be obtained from the functions declared in this package. A default-initialized object of this type is invalid; objects returned from `Get_Next_Entry` are valid.

99/2

```
type Filter_Type is array (File_Kind) of Boolean;
```

100/2

The type `Filter_Type` specifies which directory entries are provided from a search operation. If the `Directory` component is `True`, directory entries representing directories are provided. If the `Ordinary_File` component is `True`, directory entries representing ordinary files are provided. If the `Special_File` component is `True`, directory entries representing special files are provided.

101/2

```
type Search_Type is limited private;
```

102/2

The type `Search_Type` contains the state of a directory search. A default-initialized `Search_Type` object has no entries available (function `More_Entries` returns `False`). Type `Search_Type` needs finalization (see Section 8.6 [7.6], page 295).

103/2

```
procedure Start_Search (Search      : in out Search_Type;  
                        Directory   : in String;  
                        Pattern     : in String;  
                        Filter      : in Filter_Type := (others => True));
```

104/2

Starts a search in the directory named by `Directory` for entries matching `Pattern`. `Pattern` represents a pattern for matching file names. If `Pattern` is null, all items in the directory are matched; otherwise, the interpretation of `Pattern` is implementation-defined. Only items that match `Filter` will be returned. After a successful call on `Start_Search`, the object `Search` may have entries available, but it may have no entries available if no files or directories match `Pattern` and `Filter`. The exception `Name_Error` is propagated if the string given by `Directory` does not identify an existing directory, or if `Pattern` does not allow the identification of any possible external file or directory. The exception `Use_Error` is propagated if the external environment does not support the searching of the directory with the given name (in the absence of `Name_Error`). When `Start_Search` propagates `Name_Error` or `Use_Error`, the object `Search` will have no entries available.

105/2

```
procedure End_Search (Search : in out Search_Type);
```

106/2

Ends the search represented by Search. After a successful call on End_Search, the object Search will have no entries available.

107/2

```
function More_Entries (Search : in Search_Type) return Boolean;
```

108/2

Returns True if more entries are available to be returned by a call to Get_Next_Entry for the specified search object, and False otherwise.

109/2

```
procedure Get_Next_Entry (Search : in out Search_Type;  
                          Directory_Entry : out Directory_Entry_Type);■
```

110/2

Returns the next Directory_Entry for the search described by Search that matches the pattern and filter. If no further matches are available, Status_Error is raised. It is implementation-defined as to whether the results returned by this routine are altered if the contents of the directory are altered while the Search object is valid (for example, by another program). The exception Use_Error is propagated if the external environment does not support continued searching of the directory represented by Search.

111/2

```
procedure Search (  
  Directory : in String;  
  Pattern   : in String;  
  Filter    : in Filter_Type := (others => True);  
  Process   : not null access procedure (  
    Directory_Entry : in Directory_Entry_Type));
```

112/2

Searches in the directory named by Directory for entries matching Pattern. The subprogram designated by Process is called

with each matching entry in turn. Pattern represents a pattern for matching file names. If Pattern is null, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation–defined. Only items that match Filter will be returned. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error).

113/2

```
function Simple_Name (Directory_Entry : in Directory_Entry_Type)
    return String;
```

114/2

Returns the simple external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation–defined. The exception Status_Error is propagated if Directory_Entry is invalid.

115/2

```
function Full_Name (Directory_Entry : in Directory_Entry_Type)
    return String;
```

116/2

Returns the full external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation–defined. The exception Status_Error is propagated if Directory_Entry is invalid.

117/2

```
function Kind (Directory_Entry : in Directory_Entry_Type)
    return File_Kind;
```

118/2

Returns the kind of external file represented by `Directory_Entry`. The exception `Status_Error` is propagated if `Directory_Entry` is invalid.

119/2

```
function Size (Directory_Entry : in Directory_Entry_Type)
  return File_Size;
```

120/2

Returns the size of the external file represented by `Directory_Entry`. The size of an external file is the number of stream elements contained in the file. If the external file represented by `Directory_Entry` is not an ordinary file, the result is implementation-defined. The exception `Status_Error` is propagated if `Directory_Entry` is invalid. The exception `Constraint_Error` is propagated if the file size is not a value of type `File_Size`.

121/2

```
function Modification_Time (Directory_Entry : in Directory_Entry_Type)
  return Ada.Calendar.Time;
```

122/2

Returns the time that the external file represented by `Directory_Entry` was most recently modified. If the external file represented by `Directory_Entry` is not an ordinary file, the result is implementation-defined. The exception `Status_Error` is propagated if `Directory_Entry` is invalid. The exception `Use_Error` is propagated if the external environment does not support reading the modification time of the file represented by `Directory_Entry`.

Implementation Requirements

123/2

For `Copy_File`, if `Source_Name` identifies an existing external ordinary file created by a predefined Ada input-output package, and `Target_Name` and `Form` can be used in the `Create` operation of that input-output package with mode `Out_File` without raising an exception, then `Copy_File` shall not propagate `Use_Error`.

Implementation Advice

124/2

If other information about a file (such as the owner or creation date) is available in a directory entry, the implementation should provide functions in a child package `Directories`.Information to retrieve it.

125/2

`Start_Search` and `Search` should raise `Use_Error` if `Pattern` is malformed, but not if it could represent a file in the directory but does not actually do so.

126/2

`Rename` should be supported at least when both `New_Name` and `Old_Name` are simple names and `New_Name` does not identify an existing external file.

NOTES

127/2

37 The operations `Containing_Directory`, `Full_Name`, `Simple_Name`, `Base_Name`, `Extension`, and `Compose` operate on file names, not external files. The files identified by these operations do not need to exist. `Name_Error` is raised only if the file name is malformed and cannot possibly identify a file. Of these operations, only the result of `Full_Name` depends on the current default directory; the result of the others depends only on their parameters.

128/2

38 Using access types, values of `Search_Type` and `Directory_Entry_Type` can be saved and queried later. However, another task or application can modify or delete the file represented by a `Directory_Entry_Type` value or the directory represented by a `Search_Type` value; such a value can only give the information valid at the time it is created. Therefore, long-term storage of these values is not recommended.

129/2

39 If the target system does not support directories inside of directories, then `Kind` will never return `Directory` and `Containing_Directory` will always raise `Use_Error`.

130/2

40 If the target system does not support creation or deletion of directories, then `Create_Directory`, `Create_Path`, `Delete_Directory`, and `Delete_Tree` will always propagate `Use_Error`.

131/2

41 To move a file or directory to a different location, use `Rename`. Most target systems will allow renaming of files from one directory to another. If the target file or directory might already exist, it should be deleted first.

15.17 A.17 The Package Environment_Variables

1/2

The package Environment_Variables allows a program to read or modify environment variables. Environment variables are name–value pairs, where both the name and value are strings. The definition of what constitutes an <environment variable>, and the meaning of the name and value, are implementation defined.

Static Semantics

2/2

The library package Environment_Variables has the following declaration:

3/2

```
package Ada.Environment_Variables is
  pragma Preelaborate(Environment_Variables);
```

4/2

```
  function Value (Name : in String) return String;
```

5/2

```
  function Exists (Name : in String) return Boolean;
```

6/2

```
  procedure Set (Name : in String; Value : in String);
```

7/2

```
  procedure Clear (Name : in String);
  procedure Clear;
```

8/2

```
  procedure Iterate (
    Process : not null access procedure (Name, Value : in String));
```

9/2

```
end Ada.Environment_Variables;
```

10/2

```
function Value (Name : in String) return String;
```

11/2

If the external execution environment supports environment variables, then Value returns the value of the environment variable

with the given name. If no environment variable with the given name exists, then `Constraint_Error` is propagated. If the execution environment does not support environment variables, then `Program_Error` is propagated.

12/2

```
function Exists (Name : in String) return Boolean;
```

13/2

If the external execution environment supports environment variables and an environment variable with the given name currently exists, then `Exists` returns `True`; otherwise it returns `False`.

14/2

```
procedure Set (Name : in String; Value : in String);
```

15/2

If the external execution environment supports environment variables, then `Set` first clears any existing environment variable with the given name, and then defines a single new environment variable with the given name and value. Otherwise `Program_Error` is propagated.

16/2

If implementation-defined circumstances prohibit the definition of an environment variable with the given name and value, then `Constraint_Error` is propagated.

17/2

It is implementation defined whether there exist values for which the call `Set(Name, Value)` has the same effect as `Clear (Name)`.

18/2

```
procedure Clear (Name : in String);
```

19/2

If the external execution environment supports environment variables, then `Clear` deletes all existing environment variable with the given name. Otherwise `Program_Error` is propagated.

20/2

```
procedure Clear;
```

21/2

If the external execution environment supports environment variables, then `Clear` deletes all existing environment variables. Otherwise `Program_Error` is propagated.

22/2

```
procedure Iterate (  
    Process : not null access procedure (Name, Value : in String));
```

23/2

If the external execution environment supports environment variables, then `Iterate` calls the subprogram designated by `Process` for each existing environment variable, passing the name and value of that environment variable. Otherwise `Program_Error` is propagated.

24/2

If several environment variables exist that have the same name, `Process` is called once for each such variable.

Bounded (Run-Time) Errors

25/2

It is a bounded error to call `Value` if more than one environment variable exists with the given name; the possible outcomes are that:

26/2

- one of the values is returned, and that same value is returned in subsequent calls in the absence of changes to the environment; or

27/2

- `Program_Error` is propagated.

Erroneous Execution

28/2

Making calls to the procedures Set or Clear concurrently with calls to any subprogram of package Environment_Variables, or to any instantiation of Iterate, results in erroneous execution.

29/2

Making calls to the procedures Set or Clear in the actual subprogram corresponding to the Process parameter of Iterate results in erroneous execution.

Documentation Requirements

30/2

An implementation shall document how the operations of this package behave if environment variables are changed by external mechanisms (for instance, calling operating system services).

Implementation Permissions

31/2

An implementation running on a system that does not support environment variables is permitted to define the operations of package Environment_Variables with the semantics corresponding to the case where the external execution environment does support environment variables. In this case, it shall provide a mechanism to initialize a nonempty set of environment variables prior to the execution of a partition.

Implementation Advice

32/2

If the execution environment supports subprocesses, the currently defined environment variables should be used to initialize the environment variables of a subprocess.

33/2

Changes to the environment variables made outside the control of this package should be reflected immediately in the effect of the operations of this package. Changes to the environment variables made using this package should be reflected immediately in the external execution environment. This package should not perform any buffering of the environment variables.

15.18 A.18 Containers

1/2

This clause presents the specifications of the package Containers and several child packages, which provide facilities for storing collections of elements.

2/2

A variety of sequence and associative containers are provided. Each container includes a <cursor> type. A cursor is a reference to an element within a container. Many operations on cursors are common to all of the containers. A cursor referencing an element in a container is considered to be overlapping with the container object itself.

3/2

Within this clause we provide Implementation Advice for the desired average or worst case time complexity of certain operations on a container. This advice is expressed using the Landau symbol $O(X)$. Presuming f is some function of a length parameter N and $t(N)$

is the time the operation takes (on average or worst case, as specified) for the length N , a complexity of $\langle O \rangle(f(N))$ means that there exists a finite A such that for any N , $t(N)/f(N) < A$.

4/2

If the advice suggests that the complexity should be less than $\langle O \rangle(f(N))$, then for any arbitrarily small positive real D , there should exist a positive integer M such that for all $N > M$, $t(N)/f(N) < D$.

15.18.1 A.18.1 The Package Containers

1/2

The package Containers is the root of the containers subsystem.

Static Semantics

2/2

The library package Containers has the following declaration:

3/2

```
package Ada.Containers is
  pragma Pure(Containers);
```

4/2

```
  type Hash_Type is mod <implementation-defined>;
```

5/2

```
  type Count_Type is range 0 .. <implementation-defined>;
```

6/2

```
end Ada.Containers;
```

7/2

Hash_Type represents the range of the result of a hash function. Count_Type represents the (potential or actual) number of elements of a container.

Implementation Advice

8/2

Hash_Type'Modulus should be at least 2^{**32} . Count_Type'Last should be at least $2^{**31}-1$.

15.18.2 A.18.2 The Package Containers.Vectors

1/2

The language-defined generic package Containers.Vectors provides private types Vector and Cursor, and a set of operations for each type. A vector container allows insertion and deletion at any position, but it is specifically optimized for insertion and deletion at the high end (the end with the higher index) of the container. A vector container also provides random access to its elements.

2/2

A vector container behaves conceptually as an array that expands as necessary as items are inserted. The $\langle \text{length} \rangle$ of a vector is the number of elements that the vector contains. The

<capacity> of a vector is the maximum number of elements that can be inserted into the vector prior to it being automatically expanded.

3/2

Elements in a vector container can be referred to by an index value of a generic formal type. The first element of a vector always has its index value equal to the lower bound of the formal type.

4/2

A vector container may contain <empty elements>. Empty elements do not have a specified value.

Static Semantics

5/2

The generic library package Containers.Vectors has the following declaration:

6/2

```
generic
  type Index_Type is range <>;
  type Element_Type is private;
  with function "=" (Left, Right : Element_Type)
    return Boolean is <>;
package Ada.Containers.Vectors is
  pragma Preelaborate(Vectors);
```

7/2

```
  subtype Extended_Index is
    Index_Type'Base range
      Index_Type'First-1 ..
      Index_Type'Min (Index_Type'Base'Last - 1, Index_Type'Last) + 1;
  No_Index : constant Extended_Index := Extended_Index'First;
```

8/2

```
  type Vector is tagged private;
  pragma Preelaborable_Initialization(Vector);
```

9/2

```
  type Cursor is private;
  pragma Preelaborable_Initialization(Cursor);
```

10/2

```
  Empty_Vector : constant Vector;
```

11/2

```
  No_Element : constant Cursor;
```

12/2

```
function "=" (Left, Right : Vector) return Boolean;
```

13/2

```
function To_Vector (Length : Count_Type) return Vector;
```

14/2

```
function To_Vector  
  (New_Item : Element_Type;  
   Length   : Count_Type) return Vector;
```

15/2

```
function "&" (Left, Right : Vector) return Vector;
```

16/2

```
function "&" (Left  : Vector;  
            Right : Element_Type) return Vector;
```

17/2

```
function "&" (Left  : Element_Type;  
            Right : Vector) return Vector;
```

18/2

```
function "&" (Left, Right : Element_Type) return Vector;
```

19/2

```
function Capacity (Container : Vector) return Count_Type;
```

20/2

```
procedure Reserve_Capacity (Container : in out Vector;  
                           Capacity  : in   Count_Type);
```

21/2

```
function Length (Container : Vector) return Count_Type;
```

22/2

```
procedure Set_Length (Container : in out Vector;  
                    Length     : in   Count_Type);
```

23/2

```
function Is_Empty (Container : Vector) return Boolean;
```

24/2

```
procedure Clear (Container : in out Vector);
```

25/2

```
function To_Cursor (Container : Vector;  
                   Index      : Extended_Index) return Cursor;
```

26/2

```
function To_Index (Position : Cursor) return Extended_Index;
```

27/2

```
function Element (Container : Vector;  
                 Index      : Index_Type)  
  return Element_Type;
```

28/2

```
function Element (Position : Cursor) return Element_Type;
```

29/2

```
procedure Replace_Element (Container : in out Vector;  
                           Index      : in      Index_Type;  
                           New_Item   : in      Element_Type);
```

30/2

```
procedure Replace_Element (Container : in out Vector;  
                           Position   : in      Cursor;  
                           New_item   : in      Element_Type);
```

31/2

```
procedure Query_Element  
  (Container : in Vector;  
   Index      : in Index_Type;  
   Process    : not null access procedure (Element : in Element_Type));■
```

32/2

```
procedure Query_Element  
  (Position : in Cursor;  
   Process   : not null access procedure (Element : in Element_Type));■
```

33/2

```
procedure Update_Element  
  (Container : in out Vector;
```


41/2

```
procedure Insert (Container : in out Vector;
                 Before    : in    Cursor;
                 New_Item   : in    Element_Type;
                 Position   :      out Cursor;
                 Count      : in    Count_Type := 1);
```

42/2

```
procedure Insert (Container : in out Vector;
                 Before    : in    Extended_Index;
                 Count      : in    Count_Type := 1);
```

43/2

```
procedure Insert (Container : in out Vector;
                 Before    : in    Cursor;
                 Position   :      out Cursor;
                 Count      : in    Count_Type := 1);
```

44/2

```
procedure Prepend (Container : in out Vector;
                  New_Item   : in    Vector);
```

45/2

```
procedure Prepend (Container : in out Vector;
                  New_Item   : in    Element_Type;
                  Count      : in    Count_Type := 1);
```

46/2

```
procedure Append (Container : in out Vector;
                 New_Item   : in    Vector);
```

47/2

```
procedure Append (Container : in out Vector;
                 New_Item   : in    Element_Type;
                 Count      : in    Count_Type := 1);
```

48/2

```
procedure Insert_Space (Container : in out Vector;
                       Before    : in    Extended_Index;
                       Count      : in    Count_Type := 1);
```

49/2

```
procedure Insert_Space (Container : in out Vector;
```



```
Before      : in      Cursor;
Position    :      out Cursor;
Count       : in      Count_Type := 1);
```

50/2

```
procedure Delete (Container : in out Vector;
                  Index      : in      Extended_Index;
                  Count      : in      Count_Type := 1);
```

51/2

```
procedure Delete (Container : in out Vector;
                  Position   : in out Cursor;
                  Count      : in      Count_Type := 1);
```

52/2

```
procedure Delete_First (Container : in out Vector;
                        Count      : in      Count_Type := 1);
```

53/2

```
procedure Delete_Last (Container : in out Vector;
                       Count      : in      Count_Type := 1);
```

54/2

```
procedure Reverse_Elements (Container : in out Vector);
```

55/2

```
procedure Swap (Container : in out Vector;
                I, J       : in      Index_Type);
```

56/2

```
procedure Swap (Container : in out Vector;
                I, J       : in      Cursor);
```

57/2

```
function First_Index (Container : Vector) return Index_Type;
```

58/2

```
function First (Container : Vector) return Cursor;
```

59/2

```
function First_Element (Container : Vector)
    return Element_Type;
```

60/2

```
function Last_Index (Container : Vector) return Extended_Index;
```

61/2

```
function Last (Container : Vector) return Cursor;
```

62/2

```
function Last_Element (Container : Vector)
    return Element_Type;
```

63/2

```
function Next (Position : Cursor) return Cursor;
```

64/2

```
procedure Next (Position : in out Cursor);
```

65/2

```
function Previous (Position : Cursor) return Cursor;
```

66/2

```
procedure Previous (Position : in out Cursor);
```

67/2

```
function Find_Index (Container : Vector;
                    Item       : Element_Type;
                    Index      : Index_Type := Index_Type'First)
    return Extended_Index;
```

68/2

```
function Find (Container : Vector;
              Item       : Element_Type;
              Position  : Cursor := No_Element)
    return Cursor;
```

69/2

```
function Reverse_Find_Index (Container : Vector;
                             Item      : Element_Type;
                             Index     : Index_Type := Index_Type'Last)
    return Extended_Index;
```

70/2

```
function Reverse_Find (Container : Vector;
```

```

Item      : Element_Type;
Position  : Cursor := No_Element)
return Cursor;
71/2

function Contains (Container : Vector;
                  Item       : Element_Type) return Boolean;
72/2

function Has_Element (Position : Cursor) return Boolean;
73/2

procedure Iterate
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor));■
74/2

procedure Reverse_Iterate
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor));■
75/2

generic
  with function "<" (Left, Right : Element_Type)
    return Boolean is <>;
package Generic_Sorting is
76/2

  function Is_Sorted (Container : Vector) return Boolean;
77/2

  procedure Sort (Container : in out Vector);
78/2

  procedure Merge (Target  : in out Vector;
                  Source   : in out Vector);
79/2

end Generic_Sorting;
80/2

private

```

81/2

```
... -- <not specified by the language>
```

82/2

```
end Ada.Containers.Vectors;
```

83/2

The actual function for the generic formal function "=" on `Element_Type` values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions defined to use it return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions defined to use it are unspecified.

84/2

The type `Vector` is used to represent vectors. The type `Vector` needs finalization (see Section 8.6 [7.6], page 295).

85/2

`Empty_Vector` represents the empty vector object. It has a length of 0. If an object of type `Vector` is not otherwise initialized, it is initialized to the same value as `Empty_Vector`.

86/2

`No_Element` represents a cursor that designates no element. If an object of type `Cursor` is not otherwise initialized, it is initialized to the same value as `No_Element`.

87/2

The predefined "=" operator for type `Cursor` returns `True` if both cursors are `No_Element`, or designate the same element in the same container.

88/2

Execution of the default implementation of the `Input`, `Output`, `Read`, or `Write` attribute of type `Cursor` raises `Program_Error`.

89/2

`No_Index` represents a position that does not correspond to any element. The subtype `Extended_Index` includes the indices covered by `Index_Type` plus the value `No_Index` and, if it exists, the successor to the `Index_Type'Last`.

90/2

Some operations of this generic package have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

91/2

A subprogram is said to <tamper with cursors> of a vector object <V> if:

92/2

- it inserts or deletes elements of <V>, that is, it calls the `Insert`, `Insert_Space`, `Clear`, `Delete`, or `Set_Length` procedures with <V> as a parameter; or

93/2

- it finalizes <V>; or

94/2

- it calls the Move procedure with <V> as a parameter.

95/2

A subprogram is said to <tamper with elements> of a vector object <V> if:

96/2

- it tampers with cursors of <V>; or

97/2

- it replaces one or more elements of <V>, that is, it calls the Replace_Element, Reverse_Elements, or Swap procedures or the Sort or Merge procedures of an instance of Generic_Sorting with <V> as a parameter.

98/2

```
function "=" (Left, Right : Vector) return Boolean;
```

99/2

If Left and Right denote the same vector object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise it returns True. Any exception raised during evaluation of element equality is propagated.

100/2

```
function To_Vector (Length : Count_Type) return Vector;
```

101/2

Returns a vector with a length of Length, filled with empty elements.

102/2

```
function To_Vector
```

```
(New_Item : Element_Type;  
Length   : Count_Type) return Vector;
```

103/2

Returns a vector with a length of Length,
filled with elements initialized to the value
New_Item.

104/2

```
function "&" (Left, Right : Vector) return Vector;
```

105/2

Returns a vector comprising the elements of
Left followed by the elements of Right.

106/2

```
function "&" (Left  : Vector;  
             Right : Element_Type) return Vector;
```

107/2

Returns a vector comprising the elements of
Left followed by the element Right.

108/2

```
function "&" (Left  : Element_Type;  
             Right : Vector) return Vector;
```

109/2

Returns a vector comprising the element Left
followed by the elements of Right.

110/2

```
function "&" (Left, Right : Element_Type) return Vector;
```

111/2

Returns a vector comprising the element Left
followed by the element Right.

112/2

```
function Capacity (Container : Vector) return Count_Type;
```

113/2

Returns the capacity of Container.

114/2

```
procedure Reserve_Capacity (Container : in out Vector;  
                           Capacity  : in    Count_Type);
```

115/2

Reserve_Capacity allocates new internal data structures such that the length of the resulting vector can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then copies the elements into the new data structures and deallocates the old data structures. Any exception raised during allocation is propagated and Container is not modified.

116/2

```
function Length (Container : Vector) return Count_Type;
```

117/2

Returns the number of elements in Container.

118/2

```
procedure Set_Length (Container : in out Vector;  
                    Length     : in    Count_Type);
```

119/2

If Length is larger than the capacity of Container, Set_Length calls Reserve_Capacity (Container, Length), then sets the length of the Container to Length. If Length is greater than the original length of Container, empty elements are added to Container; otherwise elements are removed from Container.

120/2

```
function Is_Empty (Container : Vector) return Boolean;
```

121/2

Equivalent to $\text{Length}(\text{Container}) = 0$.

122/2

```
procedure Clear (Container : in out Vector);
```

123/2

Removes all the elements from Container.
The capacity of Container does not change.

124/2

```
function To_Cursor (Container : Vector;  
                   Index      : Extended_Index) return Cursor;
```

125/2

If Index is not in the range First_Index (Container) .. Last_Index (Container), then No_Element is returned. Otherwise, a cursor designating the element at position Index in Container is returned.

126/2

```
function To_Index (Position : Cursor) return Extended_Index;
```

127/2

If Position is No_Element, No_Index is returned. Otherwise, the index (within its containing vector) of the element designated by Position is returned.

128/2

```
function Element (Container : Vector;  
                Index      : Index_Type)  
  return Element_Type;
```

129/2

If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Element returns the element at position Index.

130/2

```
function Element (Position : Cursor) return Element_Type;
```

131/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position.

132/2

```
procedure Replace_Element (Container : in out Vector;  
                           Index      : in    Index_Type;  
                           New_Item   : in    Element_Type);
```

133/2

If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise Replace_Element assigns the value New_Item to the element at position Index. Any exception raised during the assignment is propagated. The element at position Index is not an empty element after successful call to Replace_Element.

134/2

```
procedure Replace_Element (Container : in out Vector;  
                           Position  : in    Cursor;  
                           New_Item  : in    Element_Type);
```

135/2

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns New_Item to the element designated by Position. Any exception raised during the assignment is propagated. The element at Position is not an empty element after successful call to Replace_Element.

136/2

```
procedure Query_Element  
  (Container : in Vector;  
   Index     : in Index_Type;  
   Process   : not null access procedure (Element : in Element_Type));■
```

137/2

If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Query_Element calls Process.all with the element at position Index as the argument. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

138/2

```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type));
```

139/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.all with the element designated by Position as the argument. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

140/2

```
procedure Update_Element
  (Container : in out Vector;
   Index     : in     Index_Type;
   Process   : not null access procedure (Element : in out Element_Type));
```

141/2

If Index is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Update_Element calls Process.all with the element at position Index as the argument. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

142/2

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.all shall be unconstrained.

143/2

The element at position Index is not an empty element after successful completion of this operation.

144/2

```
procedure Update_Element
  (Container : in out Vector;
   Position  : in     Cursor;
   Process   : not null access procedure (Element : in out Element_Type));
```

145/2

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.all with the element designated by Position as the argument. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

146/2

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.all shall be unconstrained.

147/2

The element designated by Position is not an empty element after successful completion of this operation.

148/2

```
procedure Move (Target : in out Vector;
               Source  : in out Vector);
```

149/2

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target); then, each element from Source is removed from Source and inserted into Target in the original order. The

length of Source is 0 after a successful call to Move.

150/2

```
procedure Insert (Container : in out Vector;  
                 Before     : in     Extended_Index;  
                 New_Item   : in     Vector);
```

151/2

If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Length(New_Item) is 0, then Insert does nothing. Otherwise, it computes the new length <NL> as the sum of the current length and Length (New_Item); if the value of Last appropriate for length <NL> would be greater than Index_Type'Last then Constraint_Error is propagated.

152/2

If the current vector capacity is less than <NL>, Reserve_Capacity (Container, <NL>) is called to increase the vector capacity. Then Insert slides the elements in the range Before .. Last_Index (Container) up by Length(New_Item) positions, and then copies the elements of New_Item to the positions starting at Before. Any exception raised during the copying is propagated.

153/2

```
procedure Insert (Container : in out Vector;  
                 Before     : in     Cursor;  
                 New_Item   : in     Vector);
```

154/2

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, if Length(New_Item) is 0, then Insert does nothing. If Before is No_Element, then the call is equivalent to Insert (Container, Last_Index (Container) + 1, New_Item);

otherwise the call is equivalent to Insert
(Container, To_Index (Before), New_Item);

155/2

```
procedure Insert (Container : in out Vector;  
                 Before    : in    Cursor;  
                 New_Item  : in    Vector;  
                 Position  :    out Cursor);
```

156/2

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If Before equals No_Element, then let <T> be Last_Index (Container) + 1; otherwise, let <T> be To_Index (Before). Insert (Container, <T>, New_Item) is called, and then Position is set to To_Cursor (Container, <T>).

157/2

```
procedure Insert (Container : in out Vector;  
                 Before    : in    Extended_Index;  
                 New_Item  : in    Element_Type;  
                 Count     : in    Count_Type := 1);
```

158/2

Equivalent to Insert (Container, Before,
To_Vector (New_Item, Count));

159/2

```
procedure Insert (Container : in out Vector;  
                 Before    : in    Cursor;  
                 New_Item  : in    Element_Type;  
                 Count     : in    Count_Type := 1);
```

160/2

Equivalent to Insert (Container, Before,
To_Vector (New_Item, Count));

161/2

```
procedure Insert (Container : in out Vector;  
                 Before    : in    Cursor;  
                 New_Item  : in    Element_Type;
```

```
Position : out Cursor;
Count    : in    Count_Type := 1);
```

162/2

Equivalent to Insert (Container, Before,
To_Vector (New_Item, Count), Position);

163/2

```
procedure Insert (Container : in out Vector;
                 Before    : in    Extended_Index;
                 Count     : in    Count_Type := 1);
```

164/2

If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, then Insert does nothing. Otherwise, it computes the new length <NL> as the sum of the current length and Count; if the value of Last appropriate for length <NL> would be greater than Index_Type'Last then Constraint_Error is propagated.

165/2

If the current vector capacity is less than <NL>, Reserve_Capacity (Container, <NL>) is called to increase the vector capacity. Then Insert slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts elements that are initialized by default (see Section 4.3.1 [3.3.1], page 61) in the positions starting at Before.

166/2

```
procedure Insert (Container : in out Vector;
                 Before    : in    Cursor;
                 Position  : out Cursor;
                 Count     : in    Count_Type := 1);
```

167/2

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If

Before equals No_Element, then let <T> be Last_Index (Container) + 1; otherwise, let <T> be To_Index (Before). Insert (Container, <T>, Count) is called, and then Position is set to To_Cursor (Container, <T>).

168/2

```
procedure Prepend (Container : in out Vector;  
                  New_Item  : in   Vector;  
                  Count     : in   Count_Type := 1);
```

169/2

Equivalent to Insert (Container, First_Index (Container), New_Item).

170/2

```
procedure Prepend (Container : in out Vector;  
                  New_Item  : in   Element_Type;  
                  Count     : in   Count_Type := 1);
```

171/2

Equivalent to Insert (Container, First_Index (Container), New_Item, Count).

172/2

```
procedure Append (Container : in out Vector;  
                 New_Item  : in   Vector);
```

173/2

Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item).

174/2

```
procedure Append (Container : in out Vector;  
                 New_Item  : in   Element_Type;  
                 Count     : in   Count_Type := 1);
```

175/2

Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item, Count).

176/2

```
procedure Insert_Space (Container : in out Vector;
```

```
Before      : in      Extended_Index;  
Count       : in      Count_Type := 1);
```

177/2

If Before is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, then Insert_Space does nothing. Otherwise, it computes the new length <NL> as the sum of the current length and Count; if the value of Last appropriate for length <NL> would be greater than Index_Type'Last then Constraint_Error is propagated.

178/2

If the current vector capacity is less than <NL>, Reserve_Capacity (Container, <NL>) is called to increase the vector capacity. Then Insert_Space slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts empty elements in the positions starting at Before.

179/2

```
procedure Insert_Space (Container : in out Vector;  
                        Before     : in      Cursor;  
                        Position   : out Cursor;  
                        Count      : in      Count_Type := 1);
```

180/2

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. If Before equals No_Element, then let <T> be Last_Index (Container) + 1; otherwise, let <T> be To_Index (Before). Insert_Space (Container, <T>, Count) is called, and then Position is set to To_Cursor (Container, <T>).

181/2

```
procedure Delete (Container : in out Vector;  
                 Index      : in      Extended_Index;  
                 Count     : in      Count_Type := 1);
```


182/2

If Index is not in the range First_Index (Container) .. Last_Index (Container) + 1, then Constraint_Error is propagated. If Count is 0, Delete has no effect. Otherwise Delete slides the elements (if any) starting at position Index + Count down to Index. Any exception raised during element assignment is propagated.

183/2

```
procedure Delete (Container : in out Vector;  
                 Position  : in out Cursor;  
                 Count     : in     Count_Type := 1);
```

184/2

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete (Container, To_Index (Position), Count) is called, and then Position is set to No_Element.

185/2

```
procedure Delete_First (Container : in out Vector;  
                       Count     : in     Count_Type := 1);
```

186/2

Equivalent to Delete (Container, First_Index (Container), Count).

187/2

```
procedure Delete_Last (Container : in out Vector;  
                      Count     : in     Count_Type := 1);
```

188/2

If Length (Container) <= Count then Delete_Last is equivalent to Clear (Container). Otherwise it is equivalent to Delete (Container, Index_Type'Val(Index_Type'Pos(Last_Index (Container)) - Count + 1), Count).

189/2

```
procedure Reverse_Elements (Container : in out List);
```

190/2

Reorders the elements of Container in reverse order.

191/2

```
procedure Swap (Container : in out Vector;  
               I, J       : in   Index_Type);
```

192/2

If either I or J is not in the range First_Index (Container) .. Last_Index (Container), then Constraint_Error is propagated. Otherwise, Swap exchanges the values of the elements at positions I and J.

193/2

```
procedure Swap (Container : in out Vector;  
               I, J       : in   Cursor);
```

194/2

If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap exchanges the values of the elements designated by I and J.

195/2

```
function First_Index (Container : Vector) return Index_Type;
```

196/2

Returns the value Index_Type'First.

197/2

```
function First (Container : Vector) return Cursor;
```

198/2

If Container is empty, First returns No_Element. Otherwise, it returns a

cursor that designates the first element in Container.

199/2

```
function First_Element (Container : Vector) return Element_Type;
```

200/2

Equivalent to Element (Container, First_Index (Container)).

201/2

```
function Last_Index (Container : Vector) return Extended_Index;
```

202/2

If Container is empty, Last_Index returns No_Index. Otherwise, it returns the position of the last element in Container.

203/2

```
function Last (Container : Vector) return Cursor;
```

204/2

If Container is empty, Last returns No_Element. Otherwise, it returns a cursor that designates the last element in Container.

205/2

```
function Last_Element (Container : Vector) return Element_Type;
```

206/2

Equivalent to Element (Container, Last_Index (Container)).

207/2

```
function Next (Position : Cursor) return Cursor;
```

208/2

If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the element

with index `To_Index (Position) + 1` in the same vector as `Position`.

209/2

```
procedure Next (Position : in out Cursor);
```

210/2

Equivalent to `Position := Next (Position)`.

211/2

```
function Previous (Position : Cursor) return Cursor;
```

212/2

If `Position` equals `No_Element` or designates the first element of the container, then `Previous` returns the value `No_Element`. Otherwise, it returns a cursor that designates the element with index `To_Index (Position) - 1` in the same vector as `Position`.

213/2

```
procedure Previous (Position : in out Cursor);
```

214/2

Equivalent to `Position := Previous (Position)`.

215/2

```
function Find_Index (Container : Vector;  
                    Item      : Element_Type;  
                    Index     : Index_Type := Index_Type'First)  
return Extended_Index;
```

216/2

Searches the elements of `Container` for an element equal to `Item` (using the generic formal equality operator). The search starts at position `Index` and proceeds towards `Last_Index (Container)`. If no equal element is found, then `Find_Index` returns `No_Index`. Otherwise, it returns the index of the first equal element encountered.

217/2

```
function Find (Container : Vector;  
              Item      : Element_Type;  
              Position  : Cursor := No_Element)  
  return Cursor;
```

218/2

If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the first element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the last element of Container. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

219/2

```
function Reverse_Find_Index (Container : Vector;  
                             Item      : Element_Type;  
                             Index     : Index_Type := Index_Type'Last)█  
  return Extended_Index;
```

220/2

Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index or, if Index is greater than Last_Index (Container), at position Last_Index (Container). It proceeds towards First_Index (Container). If no equal element is found, then Reverse_Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered.

221/2

```
function Reverse_Find (Container : Vector;  
                      Item      : Element_Type;  
                      Position  : Cursor := No_Element)  
  return Cursor;
```

222/2

If Position is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise Reverse_Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the last element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the first element of Container. If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

223/2

```
function Contains (Container : Vector;  
                  Item      : Element_Type) return Boolean;
```

224/2

Equivalent to Has_Element (Find (Container, Item)).

225/2

```
function Has_Element (Position : Cursor) return Boolean;
```

226/2

Returns True if Position designates an element, and returns False otherwise.

227/2

```
procedure Iterate  
(Container : in Vector;  
 Process   : not null access procedure (Position : in Cursor));
```

228/2

Invokes Process.all with a cursor that designates each element in Container, in index order. Program_Error is propagated if Process.all tampers with the cursors of Container. Any exception raised by Process is propagated.

229/2

```
procedure Reverse_Iterate
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor));
```

230/2

Iterates over the elements in Container as per Iterate, except that elements are traversed in reverse index order.

231/2

The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.

232/2

```
function Is_Sorted (Container : Vector) return Boolean;
```

233/2

Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

234/2

```
procedure Sort (Container : in out Vector);
```

235/2

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

236/2

```
procedure Merge (Target  : in out Vector;
                 Source  : in out Vector);
```

237/2

Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

Bounded (Run-Time) Errors

238/2

Reading the value of an empty element by calling Element, Query_Element, Update_Element, Swap, Is_Sorted, Sort, Merge, "=", Find, or Reverse_Find is a bounded error. The implementation may treat the element as having any normal value (see Section 14.9.1 [13.9.1], page 522) of the element type, or raise Constraint_Error or Program_Error before modifying the vector.

239/2

Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

240/2

A Cursor value is <ambiguous> if any of the following have occurred since it was created:

241/2

- Insert, Insert_Space, or Delete has been called on the vector that contains the element the cursor designates with an index value (or a cursor designating an element at such an index value) less than or equal to the index value of the element designated by the cursor; or

242/2

- The vector that contains the element it designates has been passed to the Sort or Merge procedures of an instance of Generic_Sorting, or to the Reverse_Elements procedure.

243/2

It is a bounded error to call any subprogram other than "=" or Has_Element declared in Containers.Vectors with an ambiguous (but not invalid, see below) cursor parameter. Possible results are:

244/2

- The cursor may be treated as if it were No_Element;

245/2

- The cursor may designate some element in the vector (but not necessarily the element that it originally designated);

246/2

- Constraint_Error may be raised; or

247/2

- Program_Error may be raised.

Erroneous Execution

248/2

A Cursor value is <invalid> if any of the following have occurred since it was created:

249/2

- The vector that contains the element it designates has been finalized;

250/2

- The vector that contains the element it designates has been used as the Source or Target of a call to Move; or

251/2

- The element it designates has been deleted.

252/2

The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Vectors is called with an invalid cursor parameter.

Implementation Requirements

253/2

No storage associated with a vector object shall be lost upon assignment or scope exit.

254/2

The execution of an assignment_statement for a vector shall have the effect of copying the elements from the source vector object to the target vector object.

Implementation Advice

255/2

Containers.Vectors should be implemented similarly to an array. In particular, if the length of a vector is <N>, then

256/2

- the worst-case time complexity of Element should be $O(\log N)$;

257

- the worst-case time complexity of Append with Count=1 when <N> is less than the capacity of the vector should be $O(\log N)$; and

258/2

- the worst–case time complexity of Prepend with Count=1 and Delete_First with Count=1 should be $O(N \log N)$.

259/2

The worst–case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be $O(N^2)$, and the average time complexity should be better than $O(N^2)$.

260/2

Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements.

261/2

Move should not copy elements, and should minimize copying of internal data structures.

262/2

If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation.

NOTES

263/2

42 All elements of a vector occupy locations in the internal array. If a sparse container is required, a Hashed_Map should be used rather than a vector.

264/2

43 If `Index_Type'Base'First = Index_Type'First` an instance of `Ada.Containers.Vectors` will raise `Constraint_Error`. A value below `Index_Type'First` is required so that an empty vector has a meaningful value of `Last_Index`.

15.18.3 A.18.3 The Package Containers.Doubly_Linked_Lists

1/2

The language–defined generic package Containers.Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. A list container is optimized for insertion and deletion at any position.

2/2

A doubly–linked list container object manages a linked list of internal <nodes>, each of which contains an element and pointers to the next (successor) and previous (predecessor) internal nodes. A cursor designates a particular node within a list (and by extension the element contained in that node). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved in the container.

3/2

The <length> of a list is the number of elements it contains.

Static Semantics

4/2

The generic library package Containers.Doubly_Linked_Lists has the following declaration:

5/2

```
generic
  type Element_Type is private;
  with function "=" (Left, Right : Element_Type)
    return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists is
  pragma Preelaborate(Doubly_Linked_Lists);
```

6/2

```
  type List is tagged private;
  pragma Preelaborable_Initialization(List);
```

7/2

```
  type Cursor is private;
  pragma Preelaborable_Initialization(Cursor);
```

8/2

```
  Empty_List : constant List;
```

9/2

```
  No_Element : constant Cursor;
```

10/2

```
  function "=" (Left, Right : List) return Boolean;
```

11/2

```
  function Length (Container : List) return Count_Type;
```

12/2

```
  function Is_Empty (Container : List) return Boolean;
```

13/2

```
  procedure Clear (Container : in out List);
```

14/2

```
  function Element (Position : Cursor)
    return Element_Type;
```

15/2

```
  procedure Replace_Element (Container : in out List;
```

```
Position : in Cursor;
New_Item : in Element_Type);
```

16/2

```
procedure Query_Element
(Position : in Cursor;
 Process : not null access procedure (Element : in Element_Type));■
```

17/2

```
procedure Update_Element
(Container : in out List;
 Position : in Cursor;
 Process : not null access procedure
 (Element : in out Element_Type));
```

18/2

```
procedure Move (Target : in out List;
 Source : in out List);
```

19/2

```
procedure Insert (Container : in out List;
 Before : in Cursor;
 New_Item : in Element_Type;
 Count : in Count_Type := 1);
```

20/2

```
procedure Insert (Container : in out List;
 Before : in Cursor;
 New_Item : in Element_Type;
 Position : out Cursor;
 Count : in Count_Type := 1);
```

21/2

```
procedure Insert (Container : in out List;
 Before : in Cursor;
 Position : out Cursor;
 Count : in Count_Type := 1);
```

22/2

```
procedure Prepend (Container : in out List;
 New_Item : in Element_Type;
 Count : in Count_Type := 1);
```

23/2

```
procedure Append (Container : in out List;
                 New_Item  : in   Element_Type;
                 Count     : in   Count_Type := 1);
```

24/2

```
procedure Delete (Container : in out List;
                 Position  : in out Cursor;
                 Count     : in   Count_Type := 1);
```

25/2

```
procedure Delete_First (Container : in out List;
                       Count     : in   Count_Type := 1);
```

26/2

```
procedure Delete_Last (Container : in out List;
                      Count     : in   Count_Type := 1);
```

27/2

```
procedure Reverse_Elements (Container : in out List);
```

28/2

```
procedure Swap (Container : in out List;
               I, J       : in   Cursor);
```

29/2

```
procedure Swap_Links (Container : in out List;
                     I, J       : in   Cursor);
```

30/2

```
procedure Splice (Target   : in out List;
                 Before    : in   Cursor;
                 Source     : in out List);
```

31/2

```
procedure Splice (Target   : in out List;
                 Before    : in   Cursor;
                 Source     : in out List;
                 Position  : in out Cursor);
```

32/2

```
procedure Splice (Container: in out List;
```

```

                                Before  : in    Cursor;
                                Position : in    Cursor);
33/2

function First (Container : List) return Cursor;
34/2

function First_Element (Container : List)
    return Element_Type;
35/2

function Last (Container : List) return Cursor;
36/2

function Last_Element (Container : List)
    return Element_Type;
37/2

function Next (Position : Cursor) return Cursor;
38/2

function Previous (Position : Cursor) return Cursor;
39/2

procedure Next (Position : in out Cursor);
40/2

procedure Previous (Position : in out Cursor);
41/2

function Find (Container : List;
              Item       : Element_Type;
              Position   : Cursor := No_Element)
    return Cursor;
42/2

function Reverse_Find (Container : List;
                     Item       : Element_Type;
                     Position   : Cursor := No_Element)
    return Cursor;
43/2

function Contains (Container : List;

```

```

Item      : Element_Type) return Boolean;
44/2

function Has_Element (Position : Cursor) return Boolean;
45/2

procedure Iterate
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor));■
46/2

procedure Reverse_Iterate
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor));■
47/2

generic
  with function "<" (Left, Right : Element_Type)
    return Boolean is <>;
package Generic_Sorting is
48/2

  function Is_Sorted (Container : List) return Boolean;
49/2

  procedure Sort (Container : in out List);
50/2

  procedure Merge (Target  : in out List;
                  Source  : in out List);
51/2

end Generic_Sorting;
52/2

private
53/2

  ... -- <not specified by the language>
54/2

end Ada.Containers.Doubly_Linked_Lists;

```

55/2

The actual function for the generic formal function "=" on `Element_Type` values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions `Find`, `Reverse_Find`, and "=" on list values return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions `Find`, `Reverse_Find`, and "=" on list values are unspecified.

56/2

The type `List` is used to represent lists. The type `List` needs finalization (see Section 8.6 [7.6], page 295).

57/2

`Empty_List` represents the empty `List` object. It has a length of 0. If an object of type `List` is not otherwise initialized, it is initialized to the same value as `Empty_List`.

58/2

`No_Element` represents a cursor that designates no element. If an object of type `Cursor` is not otherwise initialized, it is initialized to the same value as `No_Element`.

59/2

The predefined "=" operator for type `Cursor` returns `True` if both cursors are `No_Element`, or designate the same element in the same container.

60/2

Execution of the default implementation of the `Input`, `Output`, `Read`, or `Write` attribute of type `Cursor` raises `Program_Error`.

61/2

Some operations of this generic package have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

62/2

A subprogram is said to <tamper with cursors> of a list object <L> if:

63/2

- it inserts or deletes elements of <L>, that is, it calls the `Insert`, `Clear`, `Delete`, or `Delete_Last` procedures with <L> as a parameter; or

64/2

- it reorders the elements of <L>, that is, it calls the `Splice`, `Swap_Links`, or `Reverse_Elements` procedures or the `Sort` or `Merge` procedures of an instance of `Generic_Sorting` with <L> as a parameter; or

65/2

- it finalizes <L>; or

66/2

- it calls the Move procedure with <L> as a parameter.

67/2

A subprogram is said to <tamper with elements> of a list object <L> if:

68/2

- it tampers with cursors of <L>; or

69/2

- it replaces one or more elements of <L>, that is, it calls the Replace_Element or Swap procedures with <L> as a parameter.

70/2

```
function "=" (Left, Right : List) return Boolean;
```

71/2

If Left and Right denote the same list object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise it returns True. Any exception raised during evaluation of element equality is propagated.

72/2

```
function Length (Container : List) return Count_Type;
```

73/2

Returns the number of elements in Container.

74/2

```
function Is_Empty (Container : List) return Boolean;
```

75/2

Equivalent to Length (Container) = 0.

76/2

```
procedure Clear (Container : in out List);
```

77/2

Removes all the elements from Container.

78/2

```
function Element (Position : Cursor) return Element_Type;
```

79/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position.

80/2

```
procedure Replace_Element (Container : in out List;
                           Position  : in   Cursor;
                           New_Item  : in   Element_Type);
```

81/2

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns the value New_Item to the element designated by Position.

82/2

```
procedure Query_Element
(Position : in Cursor;
 Process  : not null access procedure (Element : in Element_Type));
```

83/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.all with the element designated by Position as the argument. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

84/2

```
procedure Update_Element
```

```
(Container : in out List;  
Position  : in      Cursor;  
Process   : not null access procedure (Element : in out Element_Type));■
```

85/2

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.all with the element designated by Position as the argument. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

86/2

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.all shall be unconstrained.

87/2

```
procedure Move (Target : in out List;  
               Source  : in out List);
```

88/2

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then, the nodes in Source are moved to Target (in the original order). The length of Target is set to the length of Source, and the length of Source is set to 0.

89/2

```
procedure Insert (Container : in out List;  
                Before     : in      Cursor;  
                New_Item   : in      Element_Type;  
                Count      : in      Count_Type := 1);
```

90/2

If Before is not No_Element, and does not designate an element in Container, then Program_Error is propagated. Otherwise, Insert

inserts `Count` copies of `New_Item` prior to the element designated by `Before`. If `Before` equals `No_Element`, the new elements are inserted after the last node (if any). Any exception raised during allocation of internal storage is propagated, and `Container` is not modified.

91/2

```
procedure Insert (Container : in out List;
                 Before    : in    Cursor;
                 New_Item  : in    Element_Type;
                 Position  :      out Cursor;
                 Count     : in    Count_Type := 1);
```

92/2

If `Before` is not `No_Element`, and does not designate an element in `Container`, then `Program_Error` is propagated. Otherwise, `Insert` allocates `Count` copies of `New_Item`, and inserts them prior to the element designated by `Before`. If `Before` equals `No_Element`, the new elements are inserted after the last element (if any). `Position` designates the first newly-inserted element. Any exception raised during allocation of internal storage is propagated, and `Container` is not modified.

93/2

```
procedure Insert (Container : in out List;
                 Before    : in    Cursor;
                 Position  :      out Cursor;
                 Count     : in    Count_Type := 1);
```

94/2

If `Before` is not `No_Element`, and does not designate an element in `Container`, then `Program_Error` is propagated. Otherwise, `Insert` inserts `Count` new elements prior to the element designated by `Before`. If `Before` equals `No_Element`, the new elements are inserted after the last node (if any). The new elements are initialized by default (see Section 4.3.1 [3.3.1], page 61). Any exception raised during

allocation of internal storage is propagated,
and Container is not modified.

95/2

```
procedure Prepend (Container : in out List;  
                  New_Item  : in   Element_Type;  
                  Count     : in   Count_Type := 1);
```

96/2

Equivalent to Insert (Container, First (Container), New_Item, Count).

97/2

```
procedure Append (Container : in out List;  
                 New_Item  : in   Element_Type;  
                 Count     : in   Count_Type := 1);
```

98/2

Equivalent to Insert (Container, No_Element, New_Item, Count).

99/2

```
procedure Delete (Container : in out List;  
                 Position  : in out Cursor;  
                 Count     : in   Count_Type := 1);
```

100/2

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise Delete removes (from Container) Count elements starting at the element designated by Position (or all of the elements starting at Position if there are fewer than Count elements starting at Position). Finally, Position is set to No_Element.

101/2

```
procedure Delete_First (Container : in out List;  
                       Count     : in   Count_Type := 1);
```

102/2

Equivalent to Delete (Container, First (Container), Count).

103/2

```
procedure Delete_Last (Container : in out List;  
                      Count      : in      Count_Type := 1);
```

104/2

If Length (Container) <= Count then Delete_Last is equivalent to Clear (Container). Otherwise it removes the last Count nodes from Container.

105/2

```
procedure Reverse_Elements (Container : in out List);
```

106/2

Reorders the elements of Container in reverse order.

107/2

```
procedure Swap (Container : in out List;  
              I, J       : in      Cursor);
```

108/2

If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap exchanges the values of the elements designated by I and J.

109/2

```
procedure Swap_Links (Container : in out List;  
                    I, J       : in      Cursor);
```

110/2

If either I or J is No_Element, then Constraint_Error is propagated. If either I or J do not designate an element in Container, then Program_Error is propagated. Otherwise, Swap_Links exchanges the nodes designated by I and J.

111/2

```
procedure Splice (Target    : in out List;  
                 Before    : in    Cursor;  
                 Source    : in out List);
```

112/2

If Before is not No_Element, and does not designate an element in Target, then Program_Error is propagated. Otherwise, if Source denotes the same object as Target, the operation has no effect. Otherwise, Splice reorders elements such that they are removed from Source and moved to Target, immediately prior to Before. If Before equals No_Element, the nodes of Source are spliced after the last node of Target. The length of Target is incremented by the number of nodes in Source, and the length of Source is set to 0.

113/2

```
procedure Splice (Target    : in out List;  
                 Before    : in    Cursor;  
                 Source    : in out List;  
                 Position  : in out Cursor);
```

114/2

If Position is No_Element then Constraint_Error is propagated. If Before does not equal No_Element, and does not designate an element in Target, then Program_Error is propagated. If Position does not equal No_Element, and does not designate a node in Source, then Program_Error is propagated. If Source denotes the same object as Target, then there is no effect if Position equals Before, else the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. In both cases, Position and the length of Target are unchanged. Otherwise the element designated by Position is removed from Source and moved to Target,

immediately prior to Before, or, if Before equals No_Element, after the last element of Target. The length of Target is incremented, the length of Source is decremented, and Position is updated to represent an element in Target.

115/2

```
procedure Splice (Container: in out List;  
                 Before   : in   Cursor;  
                 Position  : in   Cursor);
```

116/2

If Position is No_Element then Constraint_Error is propagated. If Before does not equal No_Element, and does not designate an element in Container, then Program_Error is propagated. If Position does not equal No_Element, and does not designate a node in Container, then Program_Error is propagated. If Position equals Before there is no effect. Otherwise, the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. The length of Container is unchanged.

117/2

```
function First (Container : List) return Cursor;
```

118/2

If Container is empty, First returns the value No_Element. Otherwise it returns a cursor that designates the first node in Container.

119/2

```
function First_Element (Container : List) return Element_Type;
```

120/2

Equivalent to Element (First (Container)).

121/2

```
function Last (Container : List) return Cursor;
```


122/2

If Container is empty, Last returns the value No_Element. Otherwise it returns a cursor that designates the last node in Container.

123/2

```
function Last_Element (Container : List) return Element_Type;
```

124/2

Equivalent to Element (Last (Container)).

125/2

```
function Next (Position : Cursor) return Cursor;
```

126/2

If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the successor of the element designated by Position.

127/2

```
function Previous (Position : Cursor) return Cursor;
```

128/2

If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the predecessor of the element designated by Position.

129/2

```
procedure Next (Position : in out Cursor);
```

130/2

Equivalent to Position := Next (Position).

131/2

```
procedure Previous (Position : in out Cursor);
```

132/2

Equivalent to `Position := Previous (Position)`.

133/2

```
function Find (Container : List;
              Item       : Element_Type;
              Position   : Cursor := No_Element)
return Cursor;
```

134/2

If `Position` is not `No_Element`, and does not designate an element in `Container`, then `Program_Error` is propagated. `Find` searches the elements of `Container` for an element equal to `Item` (using the generic formal equality operator). The search starts at the element designated by `Position`, or at the first element if `Position` equals `No_Element`. It proceeds towards `Last (Container)`. If no equal element is found, then `Find` returns `No_Element`. Otherwise, it returns a cursor designating the first equal element encountered.

135/2

```
function Reverse_Find (Container : List;
                     Item       : Element_Type;
                     Position   : Cursor := No_Element)
return Cursor;
```

136/2

If `Position` is not `No_Element`, and does not designate an element in `Container`, then `Program_Error` is propagated. `Reverse_Find` searches the elements of `Container` for an element equal to `Item` (using the generic formal equality operator). The search starts at the element designated by `Position`, or at the last element if `Position` equals `No_Element`. It proceeds towards `First (Container)`. If no equal element is found, then `Reverse_Find` returns `No_Element`. Otherwise, it returns a cursor designating the first equal element encountered.

137/2

```
function Contains (Container : List;  
                 Item      : Element_Type) return Boolean;
```

138/2

Equivalent to Find (Container, Item) /=
No_Element.

139/2

```
function Has_Element (Position : Cursor) return Boolean;
```

140/2

Returns True if Position designates an element,
and returns False otherwise.

141/2

```
procedure Iterate  
(Container : in List;  
 Process   : not null access procedure (Position : in Cursor));
```

142/2

Iterate calls Process.all with a cursor that
designates each node in Container, starting
with the first node and moving the cursor
as per the Next function. Program_Error is
propagated if Process.all tampers with the
cursors of Container. Any exception raised
by Process.all is propagated.

143/2

```
procedure Reverse_Iterate  
(Container : in List;  
 Process   : not null access procedure (Position : in Cursor));
```

144/2

Iterates over the nodes in Container as per
Iterate, except that elements are traversed in
reverse order, starting with the last node and
moving the cursor as per the Previous function.

145/2

The actual function for the generic formal function "<" of Generic_Sorting is expected to

return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. How many times the subprograms of Generic_Sorting call "<" is unspecified.

146/2

```
function Is_Sorted (Container : List) return Boolean;
```

147/2

Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

148/2

```
procedure Sort (Container : in out List);
```

149/2

Reorders the nodes of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. The sort is stable. Any exception raised during evaluation of "<" is propagated.

150/2

```
procedure Merge (Target  : in out List;  
                Source   : in out List);
```

151/2

Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

Bounded (Run-Time) Errors

152/2

Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

Erroneous Execution

153/2

A Cursor value is <invalid> if any of the following have occurred since it was created:

154/2

- The list that contains the element it designates has been finalized;

155/2

- The list that contains the element it designates has been used as the Source or Target of a call to Move; or

156/2

- The element it designates has been deleted.

157/2

The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Doubly_Linked_Lists is called with an invalid cursor parameter.

Implementation Requirements

158/2

No storage associated with a doubly-linked List object shall be lost upon assignment or scope exit.

159/2

The execution of an assignment_statement for a list shall have the effect of copying the elements from the source list object to the target list object.

Implementation Advice

160/2

Containers.Doubly_Linked_Lists should be implemented similarly to a linked list. In particular, if <N> is the length of a list, then the worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 should be $O(\log \langle N \rangle)$.

161/2

The worst-case time complexity of a call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should be $O(\langle N \rangle^2)$, and the average time complexity should be better than $O(\langle N \rangle^2)$.

162/2

Move should not copy elements, and should minimize copying of internal data structures.

163/2

If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation.

NOTES

164/2

44 Sorting a list never copies elements, and is a stable sort (equal elements remain in the original order). This is different than sorting an array or vector, which may need to copy elements, and is probably not a stable sort.

15.18.4 A.18.4 Maps

1/2

The language-defined generic packages `Containers.Hashing` and `Containers.Ordered_Maps` provide private types `Map` and `Cursor`, and a set of operations for each type. A map container allows an arbitrary type to be used as a key to find the element associated with that key. A hashed map uses a hash function to organize the keys, while an ordered map orders the keys per a specified relation.

2/2

This section describes the declarations that are common to both kinds of maps. See Section 15.18.5 [A.18.5], page 839, for a description of the semantics specific to `Containers.Hashing` and Section 15.18.6 [A.18.6], page 846, for a description of the semantics specific to `Containers.Ordered_Maps`.

Static Semantics

3/2

The actual function for the generic formal function "=" on `Element_Type` values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on map values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on map values are unspecified.

4/2

The type `Map` is used to represent maps. The type `Map` needs finalization (see Section 8.6 [7.6], page 295).

5/2

A map contains pairs of keys and elements, called `<nodes>`. Map cursors designate nodes, but also can be thought of as designating an element (the element contained in the node) for consistency with the other containers. There exists an equivalence relation on keys, whose definition is different for hashed maps and ordered maps. A map never contains two or more nodes with equivalent keys. The `<length>` of a map is the number of nodes it contains.

6/2

Each nonempty map has two particular nodes called the `<first node>` and the `<last node>` (which may be the same). Each node except for the last node has a `<successor node>`. If there are no other intervening operations, starting with the first node and repeatedly going to the successor node will visit each node in the map exactly once until the last node is reached. The exact definition of these terms is different for hashed maps and ordered maps.

7/2

Some operations of these generic packages have access-to-subprogram parameters. To

ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

8/2

A subprogram is said to <tamper with cursors> of a map object <M> if:

9/2

- it inserts or deletes elements of <M>, that is, it calls the Insert, Include, Clear, Delete, or Exclude procedures with <M> as a parameter; or

10/2

- it finalizes <M>; or

11/2

- it calls the Move procedure with <M> as a parameter; or

12/2

- it calls one of the operations defined to tamper with the cursors of <M>.

13/2

A subprogram is said to <tamper with elements> of a map object <M> if:

14/2

- it tampers with cursors of <M>; or

15/2

- it replaces one or more elements of <M>, that is, it calls the Replace or Replace_Element procedures with <M> as a parameter.

16/2

Empty_Map represents the empty Map object. It has a length of 0. If an object of type Map is not otherwise initialized, it is initialized to the same value as Empty_Map.

17/2

No_Element represents a cursor that designates no node. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

18/2

The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

19/2

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

20/2

```
function "=" (Left, Right : Map) return Boolean;
```

21/2

If Left and Right denote the same map object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each key <K> in Left, the function returns False if:

22/2

- a key equivalent to <K> is not present in Right; or

23/2

- the element associated with <K> in Left is not equal to the element associated with <K> in Right (using the generic formal equality operator for elements).

24/2

If the function has not returned a result after checking all of the keys, it returns True. Any exception raised during evaluation of key equivalence or element equality is propagated.

25/2

```
function Length (Container : Map) return Count_Type;
```

26/2

Returns the number of nodes in Container.

27/2

```
function Is_Empty (Container : Map) return Boolean;
```

28/2

Equivalent to $\text{Length}(\text{Container}) = 0$.

29/2

```
procedure Clear (Container : in out Map);
```


30/2

Removes all the nodes from Container.

31/2

```
function Key (Position : Cursor) return Key_Type;
```

32/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Key returns the key component of the node designated by Position.

33/2

```
function Element (Position : Cursor) return Element_Type;
```

34/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element component of the node designated by Position.

35/2

```
procedure Replace_Element (Container : in out Map;
                           Position  : in   Cursor;
                           New_Item  : in   Element_Type);
```

36/2

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Replace_Element assigns New_Item to the element of the node designated by Position.

37/2

```
procedure Query_Element
(Position : in Cursor;
 Process  : not null access procedure (Key      : in Key_Type;
                                       Element  : in Element_Type));
```

38/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.all with the key and element from the node designated by Position as the arguments. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

39/2

```
procedure Update_Element
  (Container : in out Map;
   Position  : in      Cursor;
   Process   : not null access procedure (Key      : in      Key_Type;
                                          Element : in out Element_Type));
```

40/2

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise Update_Element calls Process.all with the key and element from the node designated by Position as the arguments. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

41/2

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.all shall be unconstrained.

42/2

```
procedure Move (Target : in out Map;
               Source  : in out Map);
```

43/2

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first calls Clear (Target). Then, each node from Source is removed from Source and inserted into Target. The length of Source is 0 after a successful call to Move.

44/2

```
procedure Insert (Container : in out Map;
                 Key       : in   Key_Type;
                 New_Item  : in   Element_Type;
                 Position  :      out Cursor;
                 Inserted  :      out Boolean);
```

45/2

Insert checks if a node with a key equivalent to `Key` is already present in `Container`. If a match is found, `Inserted` is set to `False` and `Position` designates the element with the matching key. Otherwise, `Insert` allocates a new node, initializes it to `Key` and `New_Item`, and adds it to `Container`; `Inserted` is set to `True` and `Position` designates the newly-inserted node. Any exception raised during allocation is propagated and `Container` is not modified.

46/2

```
procedure Insert (Container : in out Map;
                 Key       : in   Key_Type;
                 Position  :      out Cursor;
                 Inserted  :      out Boolean);
```

47/2

`Insert` inserts `Key` into `Container` as per the five-parameter `Insert`, with the difference that an element initialized by default (see Section 4.3.1 [3.3.1], page 61) is inserted.

48/2

```
procedure Insert (Container : in out Map;
                 Key       : in   Key_Type;
                 New_Item  : in   Element_Type);
```

49/2

`Insert` inserts `Key` and `New_Item` into `Container` as per the five-parameter `Insert`, with the difference that if a node with a key equivalent to `Key` is already in the map, then `Constraint_Error` is propagated.

50/2

```
procedure Include (Container : in out Map;  
                  Key        : in    Key_Type;  
                  New_Item   : in    Element_Type);
```

51/2

Include inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then this operation assigns Key and New_Item to the matching node. Any exception raised during assignment is propagated.

52/2

```
procedure Replace (Container : in out Map;  
                  Key        : in    Key_Type;  
                  New_Item   : in    Element_Type);
```

53/2

Replace checks if a node with a key equivalent to Key is present in Container. If a match is found, Replace assigns Key and New_Item to the matching node; otherwise, Constraint_Error is propagated.

54/2

```
procedure Exclude (Container : in out Map;  
                  Key        : in    Key_Type);
```

55/2

Exclude checks if a node with a key equivalent to Key is present in Container. If a match is found, Exclude removes the node from the map.

56/2

```
procedure Delete (Container : in out Map;  
                  Key        : in    Key_Type);
```

57/2

Delete checks if a node with a key equivalent to Key is present in Container. If a match

is found, Delete removes the node from the map; otherwise, Constraint_Error is propagated.

58/2

```
procedure Delete (Container : in out Map;  
                 Position  : in out Cursor);
```

59/2

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete removes the node designated by Position from the map. Position is set to No_Element on return.

60/2

```
function First (Container : Map) return Cursor;
```

61/2

If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first node in Container.

62/2

```
function Next (Position  : Cursor) return Cursor;
```

63/2

Returns a cursor that designates the successor of the node designated by Position. If Position designates the last node, then No_Element is returned. If Position equals No_Element, then No_Element is returned.

64/2

```
procedure Next (Position  : in out Cursor);
```

65/2

Equivalent to Position := Next (Position).

66/2

```
function Find (Container : Map;
```

67/2 Key : Key_Type) return Cursor;

If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if a node with a key equivalent to Key is present in Container. If a match is found, a cursor designating the matching node is returned; otherwise, No_Element is returned.

68/2

```
function Element (Container : Map;
                  Key        : Key_Type) return Element_Type;
```

69/2

Equivalent to Element (Find (Container, Key)).

70/2

```
function Contains (Container : Map;
                  Key        : Key_Type) return Boolean;
```

71/2

Equivalent to Find (Container, Key) /= No_Element.

72/2

```
function Has_Element (Position : Cursor) return Boolean;
```

73/2

Returns True if Position designates a node, and returns False otherwise.

74/2

```
procedure Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor));
```

75/2

Iterate calls Process.all with a cursor that designates each node in Container, starting with the first node and moving the cursor according to the successor relation.

Program_Error is propagated if Process.all tampers with the cursors of Container. Any exception raised by Process.all is propagated.

Erroneous Execution

76/2

A Cursor value is <invalid> if any of the following have occurred since it was created:

77/2

- The map that contains the node it designates has been finalized;

78/2

- The map that contains the node it designates has been used as the Source or Target of a call to Move; or

79/2

- The node it designates has been deleted from the map.

80/2

The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Hashable_Maps or Containers.Ordered_Maps is called with an invalid cursor parameter.

Implementation Requirements

81/2

No storage associated with a Map object shall be lost upon assignment or scope exit.

82/2

The execution of an assignment_statement for a map shall have the effect of copying the elements from the source map object to the target map object.

Implementation Advice

83/2

Move should not copy elements, and should minimize copying of internal data structures.

84/2

If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation.

15.18.5 A.18.5 The Package Containers.Hashable_Maps

Static Semantics

1/2

The generic library package Containers.Hashable_Maps has the following declaration:

2/2

```
generic
  type Key_Type is private;
  type Element_Type is private;
  with function Hash (Key : Key_Type) return Hash_Type;
```

```
with function Equivalent_Keys (Left, Right : Key_Type)
  return Boolean;
with function "=" (Left, Right : Element_Type)
  return Boolean is <>;
package Ada.Containers.Hashing is
  pragma Preelaborate(Hashing);
```

3/2

```
type Map is tagged private;
pragma Preelaborable_Initialization(Map);
```

4/2

```
type Cursor is private;
pragma Preelaborable_Initialization(Cursor);
```

5/2

```
Empty_Map : constant Map;
```

6/2

```
No_Element : constant Cursor;
```

7/2

```
function "=" (Left, Right : Map) return Boolean;
```

8/2

```
function Capacity (Container : Map) return Count_Type;
```

9/2

```
procedure Reserve_Capacity (Container : in out Map;
                             Capacity : in      Count_Type);
```

10/2

```
function Length (Container : Map) return Count_Type;
```

11/2

```
function Is_Empty (Container : Map) return Boolean;
```

12/2

```
procedure Clear (Container : in out Map);
```

13/2

```
function Key (Position : Cursor) return Key_Type;
```


14/2

```
function Element (Position : Cursor) return Element_Type;
```

15/2

```
procedure Replace_Element (Container : in out Map;
                           Position  : in   Cursor;
                           New_Item  : in   Element_Type);
```

16/2

```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Key      : in Key_Type;
                                         Element  : in Element_Type));
```

17/2

```
procedure Update_Element
  (Container : in out Map;
   Position  : in   Cursor;
   Process   : not null access procedure
     (Key      : in   Key_Type;
      Element  : in out Element_Type));
```

18/2

```
procedure Move (Target : in out Map;
               Source  : in out Map);
```

19/2

```
procedure Insert (Container : in out Map;
                 Key        : in   Key_Type;
                 New_Item   : in   Element_Type;
                 Position   : out Cursor;
                 Inserted   : out Boolean);
```

20/2

```
procedure Insert (Container : in out Map;
                 Key        : in   Key_Type;
                 Position   : out Cursor;
                 Inserted   : out Boolean);
```

21/2

```
procedure Insert (Container : in out Map;
                 Key        : in   Key_Type;
```

```

                New_Item : in      Element_Type);
22/2

procedure Include (Container : in out Map;
                  Key        : in      Key_Type;
                  New_Item   : in      Element_Type);
23/2

procedure Replace (Container : in out Map;
                  Key        : in      Key_Type;
                  New_Item   : in      Element_Type);
24/2

procedure Exclude (Container : in out Map;
                  Key        : in      Key_Type);
25/2

procedure Delete (Container : in out Map;
                  Key        : in      Key_Type);
26/2

procedure Delete (Container : in out Map;
                  Position   : in out Cursor);
27/2

function First (Container : Map)
  return Cursor;
28/2

function Next (Position : Cursor) return Cursor;
29/2

procedure Next (Position : in out Cursor);
30/2

function Find (Container : Map;
              Key        : Key_Type)
  return Cursor;
31/2

function Element (Container : Map;
                  Key        : Key_Type)

```

```

        return Element_Type;
32/2

function Contains (Container : Map;
                  Key       : Key_Type) return Boolean;
33/2

function Has_Element (Position : Cursor) return Boolean;
34/2

function Equivalent_Keys (Left, Right : Cursor)
    return Boolean;
35/2

function Equivalent_Keys (Left  : Cursor;
                        Right  : Key_Type)
    return Boolean;
36/2

function Equivalent_Keys (Left  : Key_Type;
                        Right  : Cursor)
    return Boolean;
37/2

procedure Iterate
    (Container : in Map;
     Process   : not null access procedure (Position : in Cursor));■
38/2

private
39/2

    ... -- <not specified by the language>
40/2

end Ada.Containers.Hashed_Maps;
41/2

```

An object of type Map contains an expandable hash table, which is used to provide direct access to nodes. The <capacity> of an object of type Map is the maximum number of nodes that can be inserted into the hash table prior to it being automatically expanded.

42/2
Two keys <K1> and <K2> are defined to be <equivalent> if Equivalent_Keys (<K1>, <K2>) returns True.

43/2

The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular key value. For any two equivalent key values, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.

44/2

The actual function for the generic formal function Equivalent_Keys on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Keys behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Keys, and how many times they call it, is unspecified.

45/2

If the value of a key stored in a node of a map is changed other than by an operation in this package such that at least one of Hash or Equivalent_Keys give different results, the behavior of this package is unspecified.

46/2

Which nodes are the first node and the last node of a map, and which node is the successor of a given node, are unspecified, other than the general semantics described in Section 15.18.4 [A.18.4], page 830.

47/2

```
function Capacity (Container : Map) return Count_Type;
```

48/2

Returns the capacity of Container.

49/2

```
procedure Reserve_Capacity (Container : in out Map;  
                           Capacity : in    Count_Type);
```

50/2

Reserve_Capacity allocates a new hash table such that the length of the resulting map can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the nodes in Container onto the new hash table. It replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during

allocation is propagated and Container is not modified.

51/2

Reserve_Capacity tampers with the cursors of Container.

52/2

```
procedure Clear (Container : in out Map);
```

53/2

In addition to the semantics described in Section 15.18.4 [A.18.4], page 830, Clear does not affect the capacity of Container.

54/2

```
procedure Insert (Container : in out Map;
                 Key       : in   Key_Type;
                 New_Item  : in   Element_Type;
                 Position  :      out Cursor;
                 Inserted  :      out Boolean);
```

55/2

In addition to the semantics described in Section 15.18.4 [A.18.4], page 830, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value.

56/2

```
function Equivalent_Keys (Left, Right : Cursor)
  return Boolean;
```

57/2

Equivalent to Equivalent_Keys (Key (Left), Key (Right)).

58/2

```
function Equivalent_Keys (Left  : Cursor;
                          Right : Key_Type) return Boolean;
```

59/2

Equivalent to Equivalent_Keys (Key (Left),
Right).

60/2

```
function Equivalent_Keys (Left : Key_Type;  
                          Right : Cursor) return Boolean;
```

61/2

Equivalent to Equivalent_Keys (Left, Key
(Right)).

Implementation Advice

62/2

If $\langle N \rangle$ is the length of a map, the average time complexity of the subprograms Element, Insert, Include, Replace, Delete, Exclude and Find that take a key parameter should be $\langle O \rangle(\log \langle N \rangle)$. The average time complexity of the subprograms that take a cursor parameter should be $\langle O \rangle(1)$. The average time complexity of Reserve_Capacity should be $\langle O \rangle(\langle N \rangle)$.

15.18.6 A.18.6 The Package Containers.Ordered_Maps

Static Semantics

1/2

The generic library package Containers.Ordered_Maps has the following declaration:

2/2

```
generic  
  type Key_Type is private;  
  type Element_Type is private;  
  with function "<" (Left, Right : Key_Type) return Boolean is <>;  
  with function "=" (Left, Right : Element_Type) return Boolean is <>;  
package Ada.Containers.Ordered_Maps is  
  pragma Preelaborate(Ordered_Maps);
```

3/2

```
function Equivalent_Keys (Left, Right : Key_Type) return Boolean;■
```

4/2

```
type Map is tagged private;  
pragma Preelaborable_Initialization(Map);
```

5/2

```
type Cursor is private;  
pragma Preelaborable_Initialization(Cursor);
```

6/2

```
Empty_Map : constant Map;
```

7/2

```
No_Element : constant Cursor;
```

8/2

```
function "=" (Left, Right : Map) return Boolean;
```

9/2

```
function Length (Container : Map) return Count_Type;
```

10/2

```
function Is_Empty (Container : Map) return Boolean;
```

11/2

```
procedure Clear (Container : in out Map);
```

12/2

```
function Key (Position : Cursor) return Key_Type;
```

13/2

```
function Element (Position : Cursor) return Element_Type;
```

14/2

```
procedure Replace_Element (Container : in out Map;  
                           Position  : in   Cursor;  
                           New_Item  : in   Element_Type);
```

15/2

```
procedure Query_Element  
  (Position : in Cursor;  
   Process  : not null access procedure (Key      : in Key_Type;  
                                         Element  : in Element_Type));
```

16/2

```
procedure Update_Element  
  (Container : in out Map;  
   Position  : in   Cursor;  
   Process   : not null access procedure  
              (Key      : in   Key_Type;
```

```

Element : in out Element_Type));
17/2

procedure Move (Target : in out Map;
                Source : in out Map);
18/2

procedure Insert (Container : in out Map;
                 Key        : in    Key_Type;
                 New_Item   : in    Element_Type;
                 Position   :      out Cursor;
                 Inserted   :      out Boolean);
19/2

procedure Insert (Container : in out Map;
                 Key        : in    Key_Type;
                 Position   :      out Cursor;
                 Inserted   :      out Boolean);
20/2

procedure Insert (Container : in out Map;
                 Key        : in    Key_Type;
                 New_Item   : in    Element_Type);
21/2

procedure Include (Container : in out Map;
                  Key        : in    Key_Type;
                  New_Item   : in    Element_Type);
22/2

procedure Replace (Container : in out Map;
                  Key        : in    Key_Type;
                  New_Item   : in    Element_Type);
23/2

procedure Exclude (Container : in out Map;
                  Key        : in    Key_Type);
24/2

procedure Delete (Container : in out Map;
                  Key        : in    Key_Type);
25/2

procedure Delete (Container : in out Map;

```



```
                Position : in out Cursor);
26/2
    procedure Delete_First (Container : in out Map);
27/2
    procedure Delete_Last (Container : in out Map);
28/2
    function First (Container : Map) return Cursor;
29/2
    function First_Element (Container : Map) return Element_Type;
30/2
    function First_Key (Container : Map) return Key_Type;
31/2
    function Last (Container : Map) return Cursor;
32/2
    function Last_Element (Container : Map) return Element_Type;
33/2
    function Last_Key (Container : Map) return Key_Type;
34/2
    function Next (Position : Cursor) return Cursor;
35/2
    procedure Next (Position : in out Cursor);
36/2
    function Previous (Position : Cursor) return Cursor;
37/2
    procedure Previous (Position : in out Cursor);
38/2
    function Find (Container : Map;
                  Key       : Key_Type) return Cursor;
```

39/2

```
function Element (Container : Map;  
                 Key       : Key_Type) return Element_Type;
```

40/2

```
function Floor (Container : Map;  
              Key       : Key_Type) return Cursor;
```

41/2

```
function Ceiling (Container : Map;  
                Key       : Key_Type) return Cursor;
```

42/2

```
function Contains (Container : Map;  
                 Key       : Key_Type) return Boolean;
```

43/2

```
function Has_Element (Position : Cursor) return Boolean;
```

44/2

```
function "<" (Left, Right : Cursor) return Boolean;
```

45/2

```
function ">" (Left, Right : Cursor) return Boolean;
```

46/2

```
function "<" (Left : Cursor; Right : Key_Type) return Boolean;
```

47/2

```
function ">" (Left : Cursor; Right : Key_Type) return Boolean;
```

48/2

```
function "<" (Left : Key_Type; Right : Cursor) return Boolean;
```

49/2

```
function ">" (Left : Key_Type; Right : Cursor) return Boolean;
```

50/2

```
procedure Iterate  
  (Container : in Map;  
   Process   : not null access procedure (Position : in Cursor));
```

51/2

```
procedure Reverse_Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor));
```

52/2

```
private
```

53/2

```
... -- <not specified by the language>
```

54/2

```
end Ada.Containers.Ordered_Maps;
```

55/2

Two keys <K1> and <K2> are <equivalent> if both <K1> < <K2> and <K2> < <K1> return False, using the generic formal "<" operator for keys. Function Equivalent_Keys returns True if Left and Right are equivalent, and False otherwise.

56/2

The actual function for the generic formal function "<" on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive. If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

57/2

If the value of a key stored in a map is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.

58/2

The first node of a nonempty map is the one whose key is less than the key of all the other nodes in the map. The last node of a nonempty map is the one whose key is greater than the key of all the other elements in the map. The successor of a node is the node with the smallest key that is larger than the key of the given node. The predecessor of a node is the node with the largest key that is smaller than the key of the given node. All comparisons are done using the generic formal "<" operator for keys.

59/2

```
procedure Delete_First (Container : in out Map);
```

60/2

If Container is empty, Delete_First has no effect. Otherwise the node designated by First (Container) is removed from Container.

Delete_First tampers with the cursors of Container.

61/2

```
procedure Delete_Last (Container : in out Map);
```

62/2

If Container is empty, Delete_Last has no effect. Otherwise the node designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container.

63/2

```
function First_Element (Container : Map) return Element_Type;
```

64/2

Equivalent to Element (First (Container)).

65/2

```
function First_Key (Container : Map) return Key_Type;
```

66/2

Equivalent to Key (First (Container)).

67/2

```
function Last (Container : Map) return Cursor;
```

68/2

Returns a cursor that designates the last node in Container. If Container is empty, returns No_Element.

69/2

```
function Last_Element (Container : Map) return Element_Type;
```

70/2

Equivalent to Element (Last (Container)).

71/2

```
function Last_Key (Container : Map) return Key_Type;
```

72/2

Equivalent to Key (Last (Container)).

73/2

```
function Previous (Position : Cursor) return Cursor;
```

74/2

If Position equals No_Element, then Previous returns No_Element. Otherwise Previous returns a cursor designating the node that precedes the one designated by Position. If Position designates the first element, then Previous returns No_Element.

75/2

```
procedure Previous (Position : in out Cursor);
```

76/2

Equivalent to Position := Previous (Position).

77/2

```
function Floor (Container : Map;  
               Key       : Key_Type) return Cursor;
```

78/2

Floor searches for the last node whose key is not greater than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise No_Element is returned.

79/2

```
function Ceiling (Container : Map;  
                 Key       : Key_Type) return Cursor;
```

80/2

Ceiling searches for the first node whose key is not less than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise No_Element is returned.

81/2

```
function "<" (Left, Right : Cursor) return Boolean;
```

82/2

Equivalent to Key (Left) < Key (Right).

83/2

```
function ">" (Left, Right : Cursor) return Boolean;
```

84/2

Equivalent to Key (Right) < Key (Left).

85/2

```
function "<" (Left : Cursor; Right : Key_Type) return Boolean;
```

86/2

Equivalent to Key (Left) < Right.

87/2

```
function ">" (Left : Cursor; Right : Key_Type) return Boolean;
```

88/2

Equivalent to Right < Key (Left).

89/2

```
function "<" (Left : Key_Type; Right : Cursor) return Boolean;
```

90/2

Equivalent to Left < Key (Right).

91/2

```
function ">" (Left : Key_Type; Right : Cursor) return Boolean;
```

92/2

Equivalent to Key (Right) < Left.

93/2

```
procedure Reverse_Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor));
```

94/2

Iterates over the nodes in Container as per Iterate, with the difference that the nodes are traversed in predecessor order, starting with the last node.

Implementation Advice

95/2

If $\langle N \rangle$ is the length of a map, then the worst-case time complexity of the Element, Insert, Include, Replace, Delete, Exclude and Find operations that take a key parameter should be $\langle O \rangle((\log \langle N \rangle)^{**2})$ or better. The worst-case time complexity of the subprograms that take a cursor parameter should be $\langle O \rangle(1)$.

15.18.7 A.18.7 Sets

1/2

The language-defined generic packages Containers.Hashing_Sets and Containers.Ordered_Sets provide private types Set and Cursor, and a set of operations for each type. A set container allows elements of an arbitrary type to be stored without duplication. A hashed set uses a hash function to organize elements, while an ordered set orders its element per a specified relation.

2/2

This section describes the declarations that are common to both kinds of sets. See Section 15.18.8 [A.18.8], page 867, for a description of the semantics specific to Containers.Hashing_Sets and Section 15.18.9 [A.18.9], page 876, for a description of the semantics specific to Containers.Ordered_Sets.

Static Semantics

3/2

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on set values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on set values are unspecified.

4/2

The type Set is used to represent sets. The type Set needs finalization (see Section 8.6 [7.6], page 295).

5/2

A set contains elements. Set cursors designate elements. There exists an equivalence relation on elements, whose definition is different for hashed sets and ordered sets. A set never contains two or more equivalent elements. The $\langle \text{length} \rangle$ of a set is the number of elements it contains.

6/2

Each nonempty set has two particular elements called the $\langle \text{first element} \rangle$ and the $\langle \text{last element} \rangle$ (which may be the same). Each element except for the last element has a $\langle \text{successor element} \rangle$. If there are no other intervening operations, starting with the first element and repeatedly going to the successor element will visit each element in the set exactly once

until the last element is reached. The exact definition of these terms is different for hashed sets and ordered sets.

7/2

Some operations of these generic packages have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. In particular, some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced.

8/2

A subprogram is said to <tamper with cursors> of a set object <S> if:

9/2

- it inserts or deletes elements of <S>, that is, it calls the Insert, Include, Clear, Delete, Exclude, or Replace_Element procedures with <S> as a parameter; or

10/2

- it finalizes <S>; or

11/2

- it calls the Move procedure with <S> as a parameter; or

12/2

- it calls one of the operations defined to tamper with cursors of <S>.

13/2

A subprogram is said to <tamper with elements> of a set object <S> if:

14/2

- it tampers with cursors of <S>.

15/2

Empty_Set represents the empty Set object. It has a length of 0. If an object of type Set is not otherwise initialized, it is initialized to the same value as Empty_Set.

16/2

No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

17/2

The predefined "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

18/2

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

19/2

```
function "=" (Left, Right : Set) return Boolean;
```

20/2

If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element <E> in Left, the function returns False if an element equal to <E> (using the generic formal equality operator) is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equality is propagated.

21/2

```
function Equivalent_Sets (Left, Right : Set) return Boolean;
```

22/2

If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element <E> in Left, the function returns False if an element equivalent to <E> is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equivalence is propagated.

23/2

```
function To_Set (New_Item : Element_Type) return Set;
```

24/2

Returns a set containing the single element New_Item.

25/2

```
function Length (Container : Set) return Count_Type;
```

26/2

Returns the number of elements in Container.

27/2

```
function Is_Empty (Container : Set) return Boolean;
```

28/2

Equivalent to $\text{Length}(\text{Container}) = 0$.

29/2

```
procedure Clear (Container : in out Set);
```

30/2

Removes all the elements from Container.

31/2

```
function Element (Position : Cursor) return Element_Type;
```

32/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Element returns the element designated by Position.

33/2

```
procedure Replace_Element (Container : in out Set;  
                           Position  : in   Cursor;  
                           New_Item  : in   Element_Type);
```

34/2

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. If an element equivalent to New_Item is already present in Container at a position other than Position, Program_Error is propagated. Otherwise, Replace_Element assigns New_Item to the element designated by Position. Any exception raised by the assignment is propagated.

35/2

```
procedure Query_Element  
  (Position : in Cursor;
```

Process : not null access procedure (Element : in Element_Type));■

36/2

If Position equals No_Element, then Constraint_Error is propagated. Otherwise, Query_Element calls Process.all with the element designated by Position as the argument. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated.

37/2

```
procedure Move (Target : in out Set;  
               Source  : in out Set);
```

38/2

If Target denotes the same object as Source, then Move has no effect. Otherwise, Move first clears Target. Then, each element from Source is removed from Source and inserted into Target. The length of Source is 0 after a successful call to Move.

39/2

```
procedure Insert (Container : in out Set;  
                 New_Item  : in   Element_Type;  
                 Position  :    out Cursor;  
                 Inserted  :    out Boolean);
```

40/2

Insert checks if an element equivalent to New_Item is already present in Container. If a match is found, Inserted is set to False and Position designates the matching element. Otherwise, Insert adds New_Item to Container; Inserted is set to True and Position designates the newly-inserted element. Any exception raised during allocation is propagated and Container is not modified.

41/2

```
procedure Insert (Container : in out Set;
```

42/2 `New_Item : in Element_Type);`

Insert inserts `New_Item` into `Container` as per the four-parameter `Insert`, with the difference that if an element equivalent to `New_Item` is already in the set, then `Constraint_Error` is propagated.

43/2

```
procedure Include (Container : in out Set;
                  New_Item  : in   Element_Type);
```

44/2

`Include` inserts `New_Item` into `Container` as per the four-parameter `Insert`, with the difference that if an element equivalent to `New_Item` is already in the set, then it is replaced. Any exception raised during assignment is propagated.

45/2

```
procedure Replace (Container : in out Set;
                  New_Item  : in   Element_Type);
```

46/2

`Replace` checks if an element equivalent to `New_Item` is already in the set. If a match is found, that element is replaced with `New_Item`; otherwise, `Constraint_Error` is propagated.

47/2

```
procedure Exclude (Container : in out Set;
                  Item       : in   Element_Type);
```

48/2

`Exclude` checks if an element equivalent to `Item` is present in `Container`. If a match is found, `Exclude` removes the element from the set.

49/2

```
procedure Delete (Container : in out Set;
```

Item : in Element_Type);

50/2

Delete checks if an element equivalent to Item is present in Container. If a match is found, Delete removes the element from the set; otherwise, Constraint_Error is propagated.

51/2

```
procedure Delete (Container : in out Set;  
                 Position  : in out Cursor);
```

52/2

If Position equals No_Element, then Constraint_Error is propagated. If Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Delete removes the element designated by Position from the set. Position is set to No_Element on return.

53/2

```
procedure Union (Target : in out Set;  
                Source  : in   Set);
```

54/2

Union inserts into Target the elements of Source that are not equivalent to some element already in Target.

55/2

```
function Union (Left, Right : Set) return Set;
```

56/2

Returns a set comprising all of the elements of Left, and the elements of Right that are not equivalent to some element of Left.

57/2

```
procedure Intersection (Target : in out Set;  
                       Source  : in   Set);
```

58/2

Union deletes from Target the elements of Target that are not equivalent to some element of Source.

59/2

```
function Intersection (Left, Right : Set) return Set;
```

60/2

Returns a set comprising all the elements of Left that are equivalent to the some element of Right.

61/2

```
procedure Difference (Target : in out Set;  
                    Source : in    Set);
```

62/2

If Target denotes the same object as Source, then Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source.

63/2

```
function Difference (Left, Right : Set) return Set;
```

64/2

Returns a set comprising the elements of Left that are not equivalent to some element of Right.

65/2

```
procedure Symmetric_Difference (Target : in out Set;  
                               Source : in    Set);
```

66/2

If Target denotes the same object as Source, then Symmetric_Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source, and inserts into Target the elements of Source that are not equivalent to some element of Target.

67/2

```
function Symmetric_Difference (Left, Right : Set) return Set;
```

68/2

Returns a set comprising the elements of Left that are not equivalent to some element of Right, and the elements of Right that are not equivalent to some element of Left.

69/2

```
function Overlap (Left, Right : Set) return Boolean;
```

70/2

If an element of Left is equivalent to some element of Right, then Overlap returns True. Otherwise it returns False.

71/2

```
function Is_Subset (Subset : Set;  
                   Of_Set : Set) return Boolean;
```

72/2

If an element of Subset is not equivalent to some element of Of_Set, then Is_Subset returns False. Otherwise it returns True.

73/2

```
function First (Container : Set) return Cursor;
```

74/2

If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first element in Container.

75/2

```
function Next (Position : Cursor) return Cursor;
```

76/2

Returns a cursor that designates the successor of the element designated by Position. If Position designates the last element, then

No_Element is returned. If Position equals No_Element, then No_Element is returned.

77/2

```
procedure Next (Position : in out Cursor);
```

78/2

Equivalent to Position := Next (Position).

79/2

Equivalent to Find (Container, Item) /= No_Element.

80/2

```
function Find (Container : Set;  
              Item      : Element_Type) return Cursor;
```

81/2

If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if an element equivalent to Item is present in Container. If a match is found, a cursor designating the matching element is returned; otherwise, No_Element is returned.

82/2

```
function Contains (Container : Set;  
                 Item      : Element_Type) return Boolean;
```

83/2

```
function Has_Element (Position : Cursor) return Boolean;
```

84/2

Returns True if Position designates an element, and returns False otherwise.

85/2

```
procedure Iterate  
  (Container : in Set;  
   Process   : not null access procedure (Position : in Cursor));
```

86/2

95/2

If Position equals No_Element, then Constraint_Error is propagated; if Position does not designate an element in Container, then Program_Error is propagated. Otherwise, Update_Element_Preserving_Key uses Key to save the key value <K> of the element designated by Position. Update_Element_Preserving_Key then calls Process.all with that element as the argument. Program_Error is propagated if Process.all tampers with the elements of Container. Any exception raised by Process.all is propagated. After Process.all returns, Update_Element_Preserving_Key checks if <K> determines the same equivalence class as that for the new element; if not, the element is removed from the set and Program_Error is propagated.

96/2

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.all shall be unconstrained.

Erroneous Execution

97/2

A Cursor value is <invalid> if any of the following have occurred since it was created:

98/2

- The set that contains the element it designates has been finalized;

99/2

- The set that contains the element it designates has been used as the Source or Target of a call to Move; or

100/2

- The element it designates has been deleted from the set.

101/2

The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Has_Hashed_Sets or Containers.Ordered_Sets is called with an invalid cursor parameter.

Implementation Requirements

102/2

No storage associated with a Set object shall be lost upon assignment or scope exit.

103/2

The execution of an assignment_statement for a set shall have the effect of copying the elements from the source set object to the target set object.

Implementation Advice

104/2

Move should not copy elements, and should minimize copying of internal data structures.

105/2

If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation.

15.18.8 A.18.8 The Package Containers.Hash_Sets

Static Semantics

1/2

The generic library package Containers.Hash_Sets has the following declaration:

2/2

```
generic
  type Element_Type is private;
  with function Hash (Element : Element_Type) return Hash_Type;
  with function Equivalent_Elements (Left, Right : Element_Type)
    return Boolean;
  with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Hash_Sets is
  pragma Preelaborate(Hash_Sets);
```

3/2

```
  type Set is tagged private;
  pragma Preelaborable_Initialization(Set);
```

4/2

```
  type Cursor is private;
  pragma Preelaborable_Initialization(Cursor);
```

5/2

```
  Empty_Set : constant Set;
```

6/2

```
  No_Element : constant Cursor;
```

7/2

```
  function "=" (Left, Right : Set) return Boolean;
```

8/2

```
function Equivalent_Sets (Left, Right : Set) return Boolean;
```

9/2

```
function To_Set (New_Item : Element_Type) return Set;
```

10/2

```
function Capacity (Container : Set) return Count_Type;
```

11/2

```
procedure Reserve_Capacity (Container : in out Set;  
                           Capacity  : in   Count_Type);
```

12/2

```
function Length (Container : Set) return Count_Type;
```

13/2

```
function Is_Empty (Container : Set) return Boolean;
```

14/2

```
procedure Clear (Container : in out Set);
```

15/2

```
function Element (Position : Cursor) return Element_Type;
```

16/2

```
procedure Replace_Element (Container : in out Set;  
                          Position  : in   Cursor;  
                          New_Item  : in   Element_Type);
```

17/2

```
procedure Query_Element  
  (Position : in Cursor;  
   Process  : not null access procedure (Element : in Element_Type));■
```

18/2

```
procedure Move (Target : in out Set;  
              Source  : in out Set);
```

19/2

```
procedure Insert (Container : in out Set);
```

```
New_Item : in Element_Type;
Position : out Cursor;
Inserted : out Boolean);
```

20/2

```
procedure Insert (Container : in out Set;
New_Item : in Element_Type);
```

21/2

```
procedure Include (Container : in out Set;
New_Item : in Element_Type);
```

22/2

```
procedure Replace (Container : in out Set;
New_Item : in Element_Type);
```

23/2

```
procedure Exclude (Container : in out Set;
Item : in Element_Type);
```

24/2

```
procedure Delete (Container : in out Set;
Item : in Element_Type);
```

25/2

```
procedure Delete (Container : in out Set;
Position : in out Cursor);
```

26/2

```
procedure Union (Target : in out Set;
Source : in Set);
```

27/2

```
function Union (Left, Right : Set) return Set;
```

28/2

```
function "or" (Left, Right : Set) return Set renames Union;
```

29/2

```
procedure Intersection (Target : in out Set;
Source : in Set);
```

30/2

```
function Intersection (Left, Right : Set) return Set;
```

31/2

```
function "and" (Left, Right : Set) return Set renames Intersection;■
```

32/2

```
procedure Difference (Target : in out Set;  
                    Source : in Set);
```

33/2

```
function Difference (Left, Right : Set) return Set;
```

34/2

```
function "-" (Left, Right : Set) return Set renames Difference;
```

35/2

```
procedure Symmetric_Difference (Target : in out Set;  
                               Source : in Set);
```

36/2

```
function Symmetric_Difference (Left, Right : Set) return Set;
```

37/2

```
function "xor" (Left, Right : Set) return Set  
renames Symmetric_Difference;
```

38/2

```
function Overlap (Left, Right : Set) return Boolean;
```

39/2

```
function Is_Subset (Subset : Set;  
                  Of_Set : Set) return Boolean;
```

40/2

```
function First (Container : Set) return Cursor;
```

41/2

```
function Next (Position : Cursor) return Cursor;
```

42/2

```
procedure Next (Position : in out Cursor);
```

43/2

```
function Find (Container : Set;  
              Item      : Element_Type) return Cursor;
```

44/2

```
function Contains (Container : Set;  
                  Item      : Element_Type) return Boolean;
```

45/2

```
function Has_Element (Position : Cursor) return Boolean;
```

46/2

```
function Equivalent_Elements (Left, Right : Cursor)  
  return Boolean;
```

47/2

```
function Equivalent_Elements (Left  : Cursor;  
                              Right : Element_Type)  
  return Boolean;
```

48/2

```
function Equivalent_Elements (Left  : Element_Type;  
                              Right : Cursor)  
  return Boolean;
```

49/2

```
procedure Iterate  
  (Container : in Set;  
   Process   : not null access procedure (Position : in Cursor));
```

50/2

```
generic  
  type Key_Type (<>) is private;  
  with function Key (Element : Element_Type) return Key_Type;  
  with function Hash (Key : Key_Type) return Hash_Type;  
  with function Equivalent_Keys (Left, Right : Key_Type)  
    return Boolean;  
package Generic_Keys is
```

51/2

```
function Key (Position : Cursor) return Key_Type;
```

52/2

```
function Element (Container : Set;
                 Key       : Key_Type)
  return Element_Type;
```

53/2

```
procedure Replace (Container : in out Set;
                  Key       : in   Key_Type;
                  New_Item  : in   Element_Type);
```

54/2

```
procedure Exclude (Container : in out Set;
                  Key       : in   Key_Type);
```

55/2

```
procedure Delete (Container : in out Set;
                  Key       : in   Key_Type);
```

56/2

```
function Find (Container : Set;
              Key       : Key_Type)
  return Cursor;
```

57/2

```
function Contains (Container : Set;
                  Key       : Key_Type)
  return Boolean;
```

58/2

```
procedure Update_Element_Preserving_Key
  (Container : in out Set;
   Position  : in   Cursor;
   Process   : not null access procedure
               (Element : in out Element_Type));
```

59/2

```
end Generic_Keys;
```


60/2

```
private
```

61/2

```
... -- <not specified by the language>
```

62/2

```
end Ada.Containers.Hashing_Sets;
```

63/2

An object of type Set contains an expandable hash table, which is used to provide direct access to elements. The <capacity> of an object of type Set is the maximum number of elements that can be inserted into the hash table prior to it being automatically expanded.

64/2

Two elements <E1> and <E2> are defined to be <equivalent> if Equivalent_Elements (<E1>, <E2>) returns True.

65/2

The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular element value. For any two equivalent elements, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.

66/2

The actual function for the generic formal function Equivalent_Elements is expected to return the same value each time it is called with a particular pair of Element values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Elements behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Elements, and how many times they call it, is unspecified.

67/2

If the value of an element stored in a set is changed other than by an operation in this package such that at least one of Hash or Equivalent_Elements give different results, the behavior of this package is unspecified.

68/2

Which elements are the first element and the last element of a set, and which element is the successor of a given element, are unspecified, other than the general semantics described in Section 15.18.7 [A.18.7], page 855.

69/2

```
function Capacity (Container : Set) return Count_Type;
```

70/2

Returns the capacity of Container.

71/2

```
procedure Reserve_Capacity (Container : in out Set;  
                             Capacity  : in    Count_Type);
```

72/2

Reserve_Capacity allocates a new hash table such that the length of the resulting set can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the elements in Container onto the new hash table. It replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified.

73/2

Reserve_Capacity tampers with the cursors of Container.

74/2

```
procedure Clear (Container : in out Set);
```

75/2

In addition to the semantics described in Section 15.18.7 [A.18.7], page 855, Clear does not affect the capacity of Container.

76/2

```
procedure Insert (Container : in out Set;  
                 New_Item  : in    Element_Type;  
                 Position  :      out Cursor;  
                 Inserted  :      out Boolean);
```

77/2

In addition to the semantics described in Section 15.18.7 [A.18.7], page 855, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to

increase the capacity of Container to some larger value.

78/2

```
function First (Container : Set) return Cursor;
```

79/2

If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first hashed element in Container.

80/2

```
function Equivalent_Elements (Left, Right : Cursor)
    return Boolean;
```

81/2

Equivalent to Equivalent_Elements (Element (Left), Element (Right)).

82/2

```
function Equivalent_Elements (Left : Cursor;
    Right : Element_Type) return Boolean;
```

83/2

Equivalent to Equivalent_Elements (Element (Left), Right).

84/2

```
function Equivalent_Elements (Left : Element_Type;
    Right : Cursor) return Boolean;
```

85/2

Equivalent to Equivalent_Elements (Left, Element (Right)).

86/2

For any element <E>, the actual function for the generic formal function Generic_Keys.Hash is expected to be such that Hash (<E>) = Generic_Keys.Hash (Key (<E>)). If the actuals for Key or Generic_Keys.Hash behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Generic_Keys.Hash, and how many times they call it, is unspecified.

87/2

For any two elements <E1> and <E2>, the boolean values Equivalent_Elements (<E1>, <E2>)

and Equivalent_Keys (Key (<E1>), Key (<E2>)) are expected to be equal. If the actuals for Key or Equivalent_Keys behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Equivalent_Keys, and how many times they call it, is unspecified.

Implementation Advice

88/2

If <N> is the length of a set, the average time complexity of the subprograms Insert, Include, Replace, Delete, Exclude and Find that take an element parameter should be <O>(log <N>). The average time complexity of the subprograms that take a cursor parameter should be <O>(1). The average time complexity of Reserve_Capacity should be <O>(<N>).

15.18.9 A.18.9 The Package Containers.Ordered_Sets

Static Semantics

1/2

The generic library package Containers.Ordered_Sets has the following declaration:

2/2

```
generic
  type Element_Type is private;
  with function "<" (Left, Right : Element_Type) return Boolean is <>;
  with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets is
  pragma Preelaborate(Ordered_Sets);
```

3/2

```
function Equivalent_Elements (Left, Right : Element_Type) return Boolean;
```

4/2

```
type Set is tagged private;
pragma Preelaborable_Initialization(Set);
```

5/2

```
type Cursor is private;
pragma Preelaborable_Initialization(Cursor);
```

6/2

```
Empty_Set : constant Set;
```

7/2

```
No_Element : constant Cursor;
```

8/2

```
function "=" (Left, Right : Set) return Boolean;
```

9/2

```
function Equivalent_Sets (Left, Right : Set) return Boolean;
```

10/2

```
function To_Set (New_Item : Element_Type) return Set;
```

11/2

```
function Length (Container : Set) return Count_Type;
```

12/2

```
function Is_Empty (Container : Set) return Boolean;
```

13/2

```
procedure Clear (Container : in out Set);
```

14/2

```
function Element (Position : Cursor) return Element_Type;
```

15/2

```
procedure Replace_Element (Container : in out Set;  
                           Position  : in   Cursor;  
                           New_Item  : in   Element_Type);
```

16/2

```
procedure Query_Element  
  (Position : in Cursor;  
   Process  : not null access procedure (Element : in Element_Type));
```

17/2

```
procedure Move (Target : in out Set;  
               Source  : in out Set);
```

18/2

```
procedure Insert (Container : in out Set;  
                 New_Item  : in   Element_Type;  
                 Position  :    out Cursor;  
                 Inserted  :    out Boolean);
```

19/2

```
procedure Insert (Container : in out Set;  
                 New_Item  : in   Element_Type);
```

20/2

```
procedure Include (Container : in out Set;  
                  New_Item  : in   Element_Type);
```

21/2

```
procedure Replace (Container : in out Set;  
                  New_Item  : in   Element_Type);
```

22/2

```
procedure Exclude (Container : in out Set;  
                  Item      : in   Element_Type);
```

23/2

```
procedure Delete (Container : in out Set;  
                 Item      : in   Element_Type);
```

24/2

```
procedure Delete (Container : in out Set;  
                 Position  : in out Cursor);
```

25/2

```
procedure Delete_First (Container : in out Set);
```

26/2

```
procedure Delete_Last (Container : in out Set);
```

27/2

```
procedure Union (Target : in out Set;  
                Source  : in   Set);
```

28/2

```
function Union (Left, Right : Set) return Set;
```

29/2

```
function "or" (Left, Right : Set) return Set renames Union;
```

30/2

```
procedure Intersection (Target : in out Set;  
                       Source  : in   Set);
```

31/2

```
function Intersection (Left, Right : Set) return Set;
```

32/2

```
function "and" (Left, Right : Set) return Set renames Intersection;■
```

33/2

```
procedure Difference (Target : in out Set;  
                     Source : in    Set);
```

34/2

```
function Difference (Left, Right : Set) return Set;
```

35/2

```
function "-" (Left, Right : Set) return Set renames Difference;
```

36/2

```
procedure Symmetric_Difference (Target : in out Set;  
                               Source : in    Set);
```

37/2

```
function Symmetric_Difference (Left, Right : Set) return Set;
```

38/2

```
function "xor" (Left, Right : Set) return Set renames  
             Symmetric_Difference;
```

39/2

```
function Overlap (Left, Right : Set) return Boolean;
```

40/2

```
function Is_Subset (Subset : Set;  
                  Of_Set : Set) return Boolean;
```

41/2

```
function First (Container : Set) return Cursor;
```

42/2

```
function First_Element (Container : Set) return Element_Type;
```

43/2

```
function Last (Container : Set) return Cursor;
```

44/2

```
function Last_Element (Container : Set) return Element_Type;
```

45/2

```
function Next (Position : Cursor) return Cursor;
```

46/2

```
procedure Next (Position : in out Cursor);
```

47/2

```
function Previous (Position : Cursor) return Cursor;
```

48/2

```
procedure Previous (Position : in out Cursor);
```

49/2

```
function Find (Container : Set;  
              Item      : Element_Type)  
  return Cursor;
```

50/2

```
function Floor (Container : Set;  
              Item      : Element_Type)  
  return Cursor;
```

51/2

```
function Ceiling (Container : Set;  
                Item      : Element_Type)  
  return Cursor;
```

52/2

```
function Contains (Container : Set;  
                 Item      : Element_Type) return Boolean;
```

53/2

```
function Has_Element (Position : Cursor) return Boolean;
```


54/2

```
function "<" (Left, Right : Cursor) return Boolean;
```

55/2

```
function ">" (Left, Right : Cursor) return Boolean;
```

56/2

```
function "<" (Left : Cursor; Right : Element_Type)
  return Boolean;
```

57/2

```
function ">" (Left : Cursor; Right : Element_Type)
  return Boolean;
```

58/2

```
function "<" (Left : Element_Type; Right : Cursor)
  return Boolean;
```

59/2

```
function ">" (Left : Element_Type; Right : Cursor)
  return Boolean;
```

60/2

```
procedure Iterate
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor));
```

61/2

```
procedure Reverse_Iterate
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor));
```

62/2

```
generic
  type Key_Type (<>) is private;
  with function Key (Element : Element_Type) return Key_Type;
  with function "<" (Left, Right : Key_Type)
    return Boolean is <>;
package Generic_Keys is
```

63/2

```
  function Equivalent_Keys (Left, Right : Key_Type)
```

```

        return Boolean;
64/2

function Key (Position : Cursor) return Key_Type;
65/2

function Element (Container : Set;
                  Key       : Key_Type)
    return Element_Type;
66/2

procedure Replace (Container : in out Set;
                  Key       : in   Key_Type;
                  New_Item  : in   Element_Type);
67/2

procedure Exclude (Container : in out Set;
                  Key       : in   Key_Type);
68/2

procedure Delete (Container : in out Set;
                  Key       : in   Key_Type);
69/2

function Find (Container : Set;
              Key       : Key_Type)
    return Cursor;
70/2

function Floor (Container : Set;
               Key       : Key_Type)
    return Cursor;
71/2

function Ceiling (Container : Set;
                 Key       : Key_Type)
    return Cursor;
72/2

function Contains (Container : Set;
                  Key       : Key_Type) return Boolean;

```

73/2

```
procedure Update_Element_Preserving_Key
  (Container : in out Set;
   Position  : in     Cursor;
   Process   : not null access procedure
               (Element : in out Element_Type));
```

74/2

```
end Generic_Keys;
```

75/2

```
private
```

76/2

```
... -- <not specified by the language>
```

77/2

```
end Ada.Containers.Ordered_Sets;
```

78/2

Two elements <E1> and <E2> are <equivalent> if both <E1> < <E2> and <E2> < <E1> return False, using the generic formal "<" operator for elements. Function Equivalent_Elements returns True if Left and Right are equivalent, and False otherwise.

79/2

The actual function for the generic formal function "<" on Element_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive. If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

80/2

If the value of an element stored in a set is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.

81/2

The first element of a nonempty set is the one which is less than all the other elements in the set. The last element of a nonempty set is the one which is greater than all the other elements in the set. The successor of an element is the smallest element that is larger than the given element. The predecessor of an element is the largest element that is smaller than the given element. All comparisons are done using the generic formal "<" operator for elements.

82/2

```
procedure Delete_First (Container : in out Set);
```

83/2

If Container is empty, Delete_First has no effect. Otherwise the element designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container.

84/2

```
procedure Delete_Last (Container : in out Set);
```

85/2

If Container is empty, Delete_Last has no effect. Otherwise the element designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container.

86/2

```
function First_Element (Container : Set) return Element_Type;
```

87/2

Equivalent to Element (First (Container)).

88/2

```
function Last (Container : Set) return Cursor;
```

89/2

Returns a cursor that designates the last element in Container. If Container is empty, returns No_Element.

90/2

```
function Last_Element (Container : Set) return Element_Type;
```

91/2

Equivalent to Element (Last (Container)).

92/2

```
function Previous (Position : Cursor) return Cursor;
```

93/2

If Position equals No_Element, then Previous returns No_Element. Otherwise Previous returns a cursor designating the element that precedes the one designated by Position. If Position designates the first element, then Previous returns No_Element.

94/2

```
procedure Previous (Position : in out Cursor);
```

95/2

Equivalent to Position := Previous (Position).

96/2

```
function Floor (Container : Set;  
               Item      : Element_Type) return Cursor;
```

97/2

Floor searches for the last element which is not greater than Item. If such an element is found, a cursor that designates it is returned. Otherwise No_Element is returned.

98/2

```
function Ceiling (Container : Set;  
                 Item      : Element_Type) return Cursor;
```

99/2

Ceiling searches for the first element which is not less than Item. If such an element is found, a cursor that designates it is returned. Otherwise No_Element is returned.

100/2

```
function "<" (Left, Right : Cursor) return Boolean;
```

101/2

Equivalent to Element (Left) < Element (Right).

102/2

```
function ">" (Left, Right : Cursor) return Boolean;
```

103/2

Equivalent to Element (Right) < Element
(Left).

104/2

```
function "<" (Left : Cursor; Right : Element_Type) return Boolean;
```

105/2

Equivalent to Element (Left) < Right.

106/2

```
function ">" (Left : Cursor; Right : Element_Type) return Boolean;
```

107/2

Equivalent to Right < Element (Left).

108/2

```
function "<" (Left : Element_Type; Right : Cursor) return Boolean;
```

109/2

Equivalent to Left < Element (Right).

110/2

```
function ">" (Left : Element_Type; Right : Cursor) return Boolean;
```

111/2

Equivalent to Element (Right) < Left.

112/2

```
procedure Reverse_Iterate  
  (Container : in Set;  
   Process   : not null access procedure (Position : in Cursor));
```

113/2

Iterates over the elements in Container as per Iterate, with the difference that the elements are traversed in predecessor order, starting with the last element.

114/2

For any two elements <E1> and <E2>, the boolean values (<E1> < <E2>) and (Key(<E1> < Key(<E2>)) are expected to be equal. If the actuals for Key or Generic_Keys."<" behave

in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Key and Generic_Keys."<", and how many times the functions are called, is unspecified.

115/2

In addition to the semantics described in Section 15.18.7 [A.18.7], page 855, the subprograms in package Generic_Keys named Floor and Ceiling, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key subprogram parameter is compared to elements in the container using the Key and "<" generic formal functions. The function named Equivalent_Keys in package Generic_Keys returns True if both Left < Right and Right < Left return False using the generic formal "<" operator, and returns True otherwise.

Implementation Advice

116/2

If <N> is the length of a set, then the worst–case time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations that take an element parameter should be <O>((log <N>)**2) or better. The worst–case time complexity of the subprograms that take a cursor parameter should be <O>(1).

15.18.10 A.18.10 The Package Containers.Indefinite_Vectors

1/2

The language–defined generic package Containers.Indefinite_Vectors provides a private type Vector and a set of operations. It provides the same operations as the package Containers.Vectors (see Section 15.18.2 [A.18.2], page 779), with the difference that the generic formal Element_Type is indefinite.

Static Semantics

2/2

The declaration of the generic library package Containers.Indefinite_Vectors has the same contents as Containers.Vectors except:

3/2

- The generic formal Element_Type is indefinite.

4/2

- The procedures with the profiles:

5/2

```
procedure Insert (Container : in out Vector;  
                 Before    : in    Extended_Index;  
                 Count     : in    Count_Type := 1);
```

6

```
procedure Insert (Container : in out Vector;  
                 Before    : in    Cursor;  
                 Position  :      out Cursor;
```

```
Count      : in      Count_Type := 1);
```

7/2

are omitted.

8/2

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

15.18.11 A.18.11 The Package Containers.Indefinite_Doubly_Linked_Lists

1/2

The language-defined generic package Containers.Indefinite_Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. It provides the same operations as the package Containers.Doubly_Linked_Lists (see Section 15.18.3 [A.18.3], page 810), with the difference that the generic formal Element_Type is indefinite.

Static Semantics

2/2

The declaration of the generic library package Containers.Indefinite_Doubly_Linked_Lists has the same contents as Containers.Doubly_Linked_Lists except:

3/2

- The generic formal Element_Type is indefinite.

4/2

- The procedure with the profile:

5/2

```
procedure Insert (Container : in out List;  
                 Before    : in      Cursor;  
                 Position  :      out Cursor;  
                 Count     : in      Count_Type := 1);
```

6/2

is omitted.

7/2

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

15.18.12 A.18.12 The Package Containers.Indefinite_Hashed_Maps

1/2

The language–defined generic package Containers.Indefinite_Hashed_Maps provides a map with the same operations as the package Containers.Hashed_Maps (see Section 15.18.5 [A.18.5], page 839), with the difference that the generic formal types Key_Type and Element_Type are indefinite.

Static Semantics

2/2

The declaration of the generic library package Containers.Indefinite_Hashed_Maps has the same contents as Containers.Hashed_Maps except:

3/2

- The generic formal Key_Type is indefinite.

4/2

- The generic formal Element_Type is indefinite.

5/2

- The procedure with the profile:

6/2

```
procedure Insert (Container : in out Map;  
                 Key       : in   Key_Type;  
                 Position  :    out Cursor;  
                 Inserted  :    out Boolean);
```

7/2

is omitted.

8/2

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

15.18.13 A.18.13 The Package Containers.Indefinite_Ordered_Maps

1/2

The language–defined generic package Containers.Indefinite_Ordered_Maps provides a map with the same operations as the package Containers.Ordered_Maps (see Section 15.18.6 [A.18.6], page 846), with the difference that the generic formal types Key_Type and Element_Type are indefinite.

Static Semantics

2/2

The declaration of the generic library package `Containers.Indefinite_Ordered_Maps` has the same contents as `Containers.Ordered_Maps` except:

3/2

- The generic formal `Key_Type` is indefinite.

4/2

- The generic formal `Element_Type` is indefinite.

5/2

- The procedure with the profile:

6/2

```
procedure Insert (Container : in out Map;
                  Key       : in   Key_Type;
                  Position  :      out Cursor;
                  Inserted  :      out Boolean);
```

7/2

is omitted.

8/2

- The actual `Element` parameter of access subprogram `Process of Update_Element` may be constrained even if `Element_Type` is unconstrained.

15.18.14 A.18.14 The Package `Containers.Indefinite_Hashed_Sets`

1/2

The language-defined generic package `Containers.Indefinite_Hashed_Sets` provides a set with the same operations as the package `Containers.Hashed_Sets` (see Section 15.18.8 [A.18.8], page 867), with the difference that the generic formal type `Element_Type` is indefinite.

Static Semantics

2/2

The declaration of the generic library package `Containers.Indefinite_Hashed_Sets` has the same contents as `Containers.Hashed_Sets` except:

3/2

- The generic formal `Element_Type` is indefinite.

4/2

- The actual `Element` parameter of access subprogram `Process of Update_Element-Preserving_Key` may be constrained even if `Element_Type` is unconstrained.

15.18.15 A.18.15 The Package Containers.Indefinite_Ordered_Sets

1/2

The language-defined generic package Containers.Indefinite_Ordered_Sets provides a set with the same operations as the package Containers.Ordered_Sets (see Section 15.18.9 [A.18.9], page 876), with the difference that the generic formal type Element_Type is indefinite.

Static Semantics

2/2

The declaration of the generic library package Containers.Indefinite_Ordered_Sets has the same contents as Containers.Ordered_Sets except:

3/2

- The generic formal Element_Type is indefinite.

4/2

- The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

15.18.16 A.18.16 Array Sorting

1/2

The language-defined generic procedures Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort provide sorting on arbitrary array types.

Static Semantics

2/2

The generic library procedure Containers.Generic_Array_Sort has the following declaration:

3/2

```
generic
  type Index_Type is (<>);
  type Element_Type is private;
  type Array_Type is array (Index_Type range <>) of Element_Type;
  with function "<" (Left, Right : Element_Type)
    return Boolean is <>;
procedure Ada.Containers.Generic_Array_Sort (Container : in out Array_Type);
pragma Pure(Ada.Containers.Generic_Array_Sort);
```

4/2

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

5/2

The actual function for the generic formal function "<" of `Generic_Array_Sort` is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify `Container`. If the actual for "<" behaves in some other manner, the behavior of the instance of `Generic_Array_Sort` is unspecified. How many times `Generic_Array_Sort` calls "<" is unspecified.

6/2

The generic library procedure `Containers.Generic_Constrained_Array_Sort` has the following declaration:

7/2

```
generic
  type Index_Type is (<>);
  type Element_Type is private;
  type Array_Type is array (Index_Type) of Element_Type;
  with function "<" (Left, Right : Element_Type)
    return Boolean is <>;
procedure Ada.Containers.Generic_Constrained_Array_Sort
  (Container : in out Array_Type);
pragma Pure(Ada.Containers.Generic_Constrained_Array_Sort);
```

8/2

Reorders the elements of `Container` such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

9/2

The actual function for the generic formal function "<" of `Generic_Constrained_Array_Sort` is expected to return the same value each time it is called with a particular pair of element values. It should define a strict ordering relationship, that is, be irreflexive, asymmetric, and transitive; it should not modify `Container`. If the actual for "<" behaves in some

other manner, the behavior of the instance of `Generic_Constrained_Array_Sort` is unspecified. How many times `Generic_Constrained_Array_Sort` calls "`<`" is unspecified.

Implementation Advice

10/2

The worst–case time complexity of a call on an instance of `Containers.Generic_Array_Sort` or `Containers.Generic_Constrained_Array_Sort` should be $O(N^2)$ or better, and the average time complexity should be better than $O(N^2)$, where N is the length of the Container parameter.

11/2

`Containers.Generic_Array_Sort` and `Containers.Generic_Constrained_Array_Sort` should minimize copying of elements.

16 Annex B Interface to Other Languages

1

This Annex describes features for writing mixed-language programs. General interface support is presented first; then specific support for C, COBOL, and Fortran is defined, in terms of language interface packages for each of these languages.

16.1 B.1 Interfacing Pragmas

1

A pragma Import is used to import an entity defined in a foreign language into an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada. In contrast, a pragma Export is used to export an Ada entity to a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language. The pragmas Import and Export are intended primarily for objects and subprograms, although implementations are allowed to support other entities.

2

A pragma Convention is used to specify that an Ada entity should use the conventions of another language. It is intended primarily for types and "callback" subprograms. For example, "pragma Convention(Fortran, Matrix);" implies that Matrix should be represented according to the conventions of the supported Fortran implementation, namely column-major order.

3

A pragma Linker_Options is used to specify the system linker parameters needed when a given compilation unit is included in a partition.

Syntax

4

An <interfacing pragma> is a representation pragma that is one of the pragmas Import, Export, or Convention. Their forms, together with that of the related pragma Linker_Options, are as follows:

5

```
pragma Import(  
    [Convention =>] <convention_>identifier, [Entity =>] local_name  
    [, [External_Name =>] <string_>expression] [, [Link_Name =>]  
    <string_>expression]);
```

6

```
pragma Export(  
    [Convention =>] <convention_>identifier, [Entity =>] local_name  
    [, [External_Name =>] <string_>expression] [, [Link_Name =>]  
    <string_>expression]);
```

7

```
pragma Convention([Convention =>] <convention_>identifier,[Entity  
=>] local_name);
```

8

```
pragma Linker_Options(<string_>expression);
```

9

A pragma Linker_Options is allowed only at the place of a declarative_item.

9.1/1

For pragmas Import and Export, the argument for Link_Name shall not be given without the <pragma_argument_>identifier unless the argument for External_Name is given.

Name Resolution Rules

10

The expected type for a <string_>expression in an interfacing pragma or in pragma Linker_Options is String.

Legality Rules

11

The <convention_>identifier of an interfacing pragma shall be the name of a <convention>. The convention names are implementation defined, except for certain language-defined ones, such as Ada and Intrinsic, as explained in Section 7.3.1 [6.3.1], page 263, "Section 7.3.1 [6.3.1], page 263, Conformance Rules". Additional convention names generally represent the calling conventions of foreign languages, language implementations, or specific run-time models. The convention of a callable entity is its <calling convention>.

12

If <L> is a <convention_>identifier for a language, then a type T is said to be <compatible with convention L>, (alternatively, is said to be an <L-compatible type>) if any of the following conditions are met:

13

- T is declared in a language interface package corresponding to <L> and is defined to be <L>-compatible (see Section 16.3 [B.3], page 901, Section 16.3.1 [B.3.1], page 915, Section 16.3.2 [B.3.2], page 922, Section 16.4 [B.4], page 931, Section 16.5 [B.5], page 945),

14

- Convention <L> has been specified for T in a pragma Convention, and T is <eligible for convention L>; that is:

15

- T is an array type with either an unconstrained or statically-constrained

first subtype, and its component type is
<L>-compatible,

16

- T is a record type that has no discriminants and that only has components with statically-constrained subtypes, and each component type is <L>-compatible,

17

- T is an access-to-object type, and its designated type is <L>-compatible,

18

- T is an access-to-subprogram type, and its designated profile's parameter and result types are all <L>-compatible.

19

- T is derived from an <L>-compatible type,

20

- The implementation permits T as an <L>-compatible type.

21

If pragma Convention applies to a type, then the type shall either be compatible with or eligible for the convention specified in the pragma.

22

A pragma Import shall be the completion of a declaration. Notwithstanding any rule to the contrary, a pragma Import may serve as the completion of any kind of (explicit) declaration if supported by an implementation for that kind of declaration. If a completion is a pragma Import, then it shall appear in the same declarative_part, package_specification, task_definition or protected_definition as the declaration. For a library unit, it shall appear in the same compilation, before any subsequent compilation_units other than pragmas. If the local_name denotes more than one entity, then the pragma Import is the completion of all of them.

23

An entity specified as the Entity argument to a pragma Import (or pragma Export) is said to be <imported> (respectively, <exported>).

24

The declaration of an imported object shall not include an explicit initialization expression. Default initializations are not performed.

25

The type of an imported or exported object shall be compatible with the convention specified in the corresponding pragma.

26

For an imported or exported subprogram, the result and parameter types shall each be compatible with the convention specified in the corresponding pragma.

27

The external name and link name <string>_expressions of a pragma Import or Export, and the <string>-expression of a pragma Linker_Options, shall be static.

Static Semantics

28

Import, Export, and Convention pragmas are representation pragmas that specify the <convention> aspect of representation. In addition, Import and Export pragmas specify the <imported> and <exported> aspects of representation, respectively.

29

An interfacing pragma is a program unit pragma when applied to a program unit (see Section 11.1.5 [10.1.5], page 407).

30

An interfacing pragma defines the convention of the entity denoted by the local_name. The convention represents the calling convention or representation convention of the entity. For an access-to-subprogram type, it represents the calling convention of designated subprograms. In addition:

31

- A pragma Import specifies that the entity is defined externally (that is, outside the Ada program).

32

- A pragma Export specifies that the entity is used externally.

33

- A pragma Import or Export optionally specifies an entity's external name, link name, or both.

34

An <external name> is a string value for the name used by a foreign language program either for an entity that an Ada program imports, or for referring to an entity that an Ada program exports.

35

A <link name> is a string value for the name of an exported or imported entity, based on the conventions of the foreign language's compiler in interfacing with the system's linker tool.

36

The meaning of link names is implementation defined. If neither a link name nor the

Address attribute of an imported or exported entity is specified, then a link name is chosen in an implementation—defined manner, based on the external name if one is specified.

37

Pragma `Linker_Options` has the effect of passing its string argument as a parameter to the system linker (if one exists), if the immediately enclosing compilation unit is included in the partition being linked. The interpretation of the string argument, and the way in which the string arguments from multiple `Linker_Options` pragmas are combined, is implementation defined.

Dynamic Semantics

38

Notwithstanding what this International Standard says elsewhere, the elaboration of a declaration denoted by the `local_name` of a pragma `Import` does not create the entity. Such an elaboration has no other effect than to allow the defining name to denote the external entity.

Erroneous Execution

38.1/2

It is the programmer's responsibility to ensure that the use of interfacing pragmas does not violate Ada semantics; otherwise, program execution is erroneous.

Implementation Advice

39

If an implementation supports pragma `Export` to a given language, then it should also allow the main subprogram to be written in that language. It should support some mechanism for invoking the elaboration of the Ada library units included in the system, and for invoking the finalization of the environment task. On typical systems, the recommended mechanism is to provide two subprograms whose link names are "adainit" and "adafinal". `Adainit` should contain the elaboration code for library units. `Adafinal` should contain the finalization code. These subprograms should have no effect the second and subsequent time they are called.

40

Automatic elaboration of preelaborated packages should be provided when pragma `Export` is supported.

41

For each supported convention `<L>` other than `Intrinsic`, an implementation should support `Import` and `Export` pragmas for objects of `<L>`—compatible types and for subprograms, and pragma `Convention` for `<L>`—eligible types and for subprograms, presuming the other language has corresponding features. Pragma `Convention` need not be supported for scalar types.

NOTES

42

1 Implementations may place restrictions on interfacing pragmas; for example, requiring each exported entity to be declared at the library level.

43

2 A pragma Import specifies the conventions for accessing external entities. It is possible that the actual entity is written in assembly language, but reflects the conventions of a particular language. For example, pragma Import(Ada, ...) can be used to interface to an assembly language routine that obeys the Ada compiler's calling conventions.

44

3 To obtain "call-back" to an Ada subprogram from a foreign language environment, pragma Convention should be specified both for the access-to-subprogram type and the specific subprogram(s) to which 'Access is applied.

45

4 It is illegal to specify more than one of Import, Export, or Convention for a given entity.

46

5 The local_name in an interfacing pragma can denote more than one entity in the case of overloading. Such a pragma applies to all of the denoted entities.

47

6 See also Section 14.8 [13.8], page 518, "Section 14.8 [13.8], page 518, Machine Code Insertions".

48

7 If both External_Name and Link_Name are specified for an Import or Export pragma, then the External_Name is ignored.

49/2

<This paragraph was deleted.>

Examples

50

<Example of interfacing pragmas:>

51

```
package Fortran_Library is
  function Sqrt (X : Float) return Float;
  function Exp  (X : Float) return Float;
private
  pragma Import(Fortran, Sqrt);
  pragma Import(Fortran, Exp);
end Fortran_Library;
```

16.2 B.2 The Package Interfaces

1

Package Interfaces is the parent of several library packages that declare types and other entities useful for interfacing to foreign languages. It also contains some implementation-defined types that are useful across more than one language (in particular for interfacing to assembly language).

Static Semantics

2

The library package Interfaces has the following skeletal declaration:

3

```
package Interfaces is
  pragma Pure(Interfaces);

  type Integer_<n> is range -2**(<n>-1) .. 2**(<n>-1) - 1;  --<2's complement>

  type Unsigned_<n> is mod 2**<n>;

  function Shift_Left (Value : Unsigned_<n>; Amount : Natural)
    return Unsigned_<n>;
  function Shift_Right (Value : Unsigned_<n>; Amount : Natural)
    return Unsigned_<n>;
  function Shift_Right_Arithmetic (Value : Unsigned_<n>; Amount : Natural)
    return Unsigned_<n>;
  function Rotate_Left (Value : Unsigned_<n>; Amount : Natural)
    return Unsigned_<n>;
  function Rotate_Right (Value : Unsigned_<n>; Amount : Natural)
    return Unsigned_<n>;
  ...
end Interfaces;
```

Implementation Requirements

7

An implementation shall provide the following declarations in the visible part of package Interfaces:

8

- Signed and modular integer types of <n> bits, if supported by the target architecture, for each <n> that is at least the size of a storage element and that is a factor of the word size. The names of these types are of the form Integer_<n> for the signed types, and Unsigned_<n> for the modular types;

9

- For each such modular type in Interfaces, shifting and rotating subprograms as specified in the declaration of Interfaces above. These subprograms are Intrinsic. They operate on a bit-by-bit basis, using the binary representation of the value of the operands to yield a binary representation for the result. The Amount parameter gives the number of bits by which to shift or rotate. For shifting, zero bits are shifted in, except in the case of Shift_Right_Arithmetic, where one bits are shifted in if Value is at least half the modulus.

10

- Floating point types corresponding to each floating point format fully supported by the hardware.

10.1/2

Support for interfacing to any foreign language is optional. However, an implementation shall not provide any attribute, library unit, or pragma having the same name as an attribute, library unit, or pragma (respectively) specified in the following clauses of this Annex unless the provided construct is either as specified in those clauses or is more limited in capability than that required by those clauses. A program that attempts to use an unsupported capability of this Annex shall either be identified by the implementation before run time or shall raise an exception at run time.

Implementation Permissions

11

An implementation may provide implementation-defined library units that are children of Interfaces, and may add declarations to the visible part of Interfaces in addition to the ones defined above.

11.1/2

A child package of package Interfaces with the name of a convention may be provided independently of whether the convention is supported by the pragma Convention and vice versa. Such a child package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in Interfaces.

Implementation Advice

12/2

<This paragraph was deleted.>

13

An implementation supporting an interface to C, COBOL, or Fortran should provide the corresponding package or packages described in the following clauses.

16.3 B.3 Interfacing with C and C++

1/2

The facilities relevant to interfacing with the C language and the corresponding subset of the C++ language are the package Interfaces.C and its children; support for the Import, Export,

and Convention pragmas with <convention>_identifier C; and support for the Convention pragma with <convention>_identifier C.Pass-By-Copy.

2/2

The package Interfaces.C contains the basic types, constants and subprograms that allow an Ada program to pass scalars and strings to C and C++ functions. When this clause mentions a C entity, the reference also applies to the corresponding entity in C++.

Static Semantics

3

The library package Interfaces.C has the following declaration:

4

```
package Interfaces.C is
  pragma Pure(C);
```

5

```
<-- Declarations based on C's <limits.h>>
```

6

```
CHAR_BIT  : constant := <implementation-defined>; <-- typically 8>
SCHAR_MIN : constant := <implementation-defined>; <-- typically -128>
SCHAR_MAX : constant := <implementation-defined>; <-- typically 127>
UCHAR_MAX : constant := <implementation-defined>; <-- typically 255>
```

7

```
<-- Signed and Unsigned Integers>
type int    is range <implementation-defined>;
type short is range <implementation-defined>;
type long  is range <implementation-defined>;
```

8

```
type signed_char is range SCHAR_MIN .. SCHAR_MAX;
for signed_char'Size use CHAR_BIT;
```

9

```
type unsigned      is mod <implementation-defined>;
type unsigned_short is mod <implementation-defined>;
type unsigned_long  is mod <implementation-defined>;
```

10

```
type unsigned_char is mod (UCHAR_MAX+1);
for unsigned_char'Size use CHAR_BIT;
```

11

```
subtype plain_char is <implementation-defined>;
```

12

```
type ptrdiff_t is range <implementation-defined>;
```

13

```
type size_t is mod <implementation-defined>;
```

14

```
<-- Floating Point>
```

15

```
type C_float is digits <implementation-defined>;
```

16

```
type double is digits <implementation-defined>;
```

17

```
type long_double is digits <implementation-defined>;
```

18

```
<-- Characters and Strings >
```

19

```
type char is <<implementation-defined character type>>;
```

20/1

```
nul : constant char := <implementation-defined>;
```

21

```
function To_C (Item : in Character) return char;
```

22

```
function To_Ada (Item : in char) return Character;
```

23

```
type char_array is array (size_t range <>) of aliased char;  
pragma Pack(char_array);  
for char_array'Component_Size use CHAR_BIT;
```

24

```
function Is_Nul_Terminated (Item : in char_array) return Boolean;■
```

25

```
function To_C (Item      : in String;
               Append_Nul : in Boolean := True)
  return char_array;
```

26

```
function To_Ada (Item      : in char_array;
                 Trim_Nul  : in Boolean := True)
  return String;
```

27

```
procedure To_C (Item      : in String;
                Target    : out char_array;
                Count     : out size_t;
                Append_Nul : in Boolean := True);
```

28

```
procedure To_Ada (Item      : in char_array;
                  Target    : out String;
                  Count     : out Natural;
                  Trim_Nul  : in Boolean := True);
```

29

```
<-- Wide Character and Wide String>
```

30/1

```
type wchar_t is <<implementation-defined character type>>;
```

31/1

```
wide_nul : constant wchar_t := <implementation-defined>;
```

32

```
function To_C (Item : in Wide_Character) return wchar_t;
function To_Ada (Item : in wchar_t      ) return Wide_Character;■
```

33

```
type wchar_array is array (size_t range <>) of aliased wchar_t;
```

34

```
pragma Pack(wchar_array);
```


35

```
function Is_Nul_Terminated (Item : in wchar_array) return Boolean;■
```

36

```
function To_C (Item      : in Wide_String;
               Append_Nul : in Boolean := True)
  return wchar_array;
```

37

```
function To_Ada (Item      : in wchar_array;
                Trim_Nul  : in Boolean := True)
  return Wide_String;
```

38

```
procedure To_C (Item      : in Wide_String;
               Target     : out wchar_array;
               Count      : out size_t;
               Append_Nul : in Boolean := True);
```

39

```
procedure To_Ada (Item      : in wchar_array;
                 Target     : out Wide_String;
                 Count      : out Natural;
                 Trim_Nul  : in Boolean := True);
```

39.1/2

```
-- <ISO/IEC 10646:2003 compatible types defined by ISO/IEC TR 19769:2004.>■
```

39.2/2

```
type char16_t is <<implementation-defined character type>>;
```

39.3/2

```
char16_nul : constant char16_t := <implementation-defined>;
```

39.4/2

```
function To_C (Item : in Wide_Character) return char16_t;
function To_Ada (Item : in char16_t) return Wide_Character;
```

39.5/2

```
type char16_array is array (size_t range <>) of aliased char16_t;■
```

39.6/2

```
pragma Pack(char16_array);
```

39.7/2

```
function Is_Nul_Terminated (Item : in char16_array) return Boolean;
function To_C (Item          : in Wide_String;
               Append_Nul   : in Boolean := True)
  return char16_array;
```

39.8/2

```
function To_Ada (Item      : in char16_array;
                 Trim_Nul  : in Boolean := True)
  return Wide_String;
```

39.9/2

```
procedure To_C (Item      : in Wide_String;
                Target    : out char16_array;
                Count     : out size_t;
                Append_Nul : in Boolean := True);
```

39.10/2

```
procedure To_Ada (Item      : in char16_array;
                  Target    : out Wide_String;
                  Count     : out Natural;
                  Trim_Nul  : in Boolean := True);
```

39.11/2

```
type char32_t is <<implementation-defined character type>>;
```

39.12/2

```
char32_nul : constant char32_t := <implementation-defined>;
```

39.13/2

```
function To_C (Item : in Wide_Wide_Character) return char32_t;
function To_Ada (Item : in char32_t) return Wide_Wide_Character;
```

39.14/2

```
type char32_array is array (size_t range <>) of aliased char32_t;
```

39.15/2

```
pragma Pack(char32_array);
```

39.16/2

```
function Is_Nul_Terminated (Item : in char32_array) return Boolean;
function To_C (Item          : in Wide_Wide_String;
               Append_Nul   : in Boolean := True)
  return char32_array;
```

39.17/2

```
function To_Ada (Item      : in char32_array;
                 Trim_Nul  : in Boolean := True)
  return Wide_Wide_String;
```

39.18/2

```
procedure To_C (Item          : in Wide_Wide_String;
                Target        : out char32_array;
                Count         : out size_t;
                Append_Nul   : in Boolean := True);
```

39.19/2

```
procedure To_Ada (Item      : in char32_array;
                  Target     : out Wide_Wide_String;
                  Count      : out Natural;
                  Trim_Nul   : in Boolean := True);
```

40

```
Terminator_Error : exception;
```

41

```
end Interfaces.C;
```

42

Each of the types declared in `Interfaces.C` is C-compatible.

43/2

The types `int`, `short`, `long`, `unsigned`, `ptrdiff_t`, `size_t`, `double`, `char`, `wchar_t`, `char16_t`, and `char32_t` correspond respectively to the C types having the same names. The types `signed_char`, `unsigned_short`, `unsigned_long`, `unsigned_char`, `C_float`, and `long_double` correspond respectively to the C types `signed char`, `unsigned short`, `unsigned long`, `unsigned char`, `float`, and `long double`.

44

The type of the subtype `plain_char` is either `signed_char` or `unsigned_char`, depending on the C implementation.

45

```
function To_C (Item : in Character) return char;
```

```
function To_Ada (Item : in char      ) return Character;
```

46

The functions To_C and To_Ada map between the Ada type Character and the C type char.

47

```
function Is_Nul_Terminated (Item : in char_array) return Boolean;
```

48

The result of Is_Nul_Terminated is True if Item contains nul, and is False otherwise.

49

```
function To_C (Item : in String; Append_Nul : in Boolean := True) return char_array;
```

```
function To_Ada (Item : in char_array; Trim_Nul : in Boolean := True) return String;
```

50/2

The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True. If Append_Nul is False and Item'Length is 0, then To_C propagates Constraint_Error.

51

The result of To_Ada is a String whose length is Item'Length (if Trim_Nul is False) or the length of the slice of Item preceding the first nul (if Trim_Nul is True). The lower bound of the result is 1. If Trim_Nul is False, then for each component Item(I) the corresponding component in the result is To_Ada applied to Item(I). If Trim_Nul is True, then for each component Item(I) before the first nul the corresponding component in the result is To_Ada applied to Item(I). The function

propagates Terminator_Error if Trim_Nul is True and Item does not contain nul.

52

```
procedure To_C (Item      : in String;
                Target    : out char_array;
                Count     : out size_t;
                Append_Nul : in Boolean := True);

procedure To_Ada (Item      : in char_array;
                 Target     : out String;
                 Count      : out Natural;
                 Trim_Nul   : in Boolean := True);
```

53

For procedure To_C, each element of Item is converted (via the To_C function) to a char, which is assigned to the corresponding element of Target. If Append_Nul is True, nul is then assigned to the next element of Target. In either case, Count is set to the number of Target elements assigned. If Target is not long enough, Constraint_Error is propagated.

54

For procedure To_Ada, each element of Item (if Trim_Nul is False) or each element of Item preceding the first nul (if Trim_Nul is True) is converted (via the To_Ada function) to a Character, which is assigned to the corresponding element of Target. Count is set to the number of Target elements assigned. If Target is not long enough, Constraint_Error is propagated. If Trim_Nul is True and Item does not contain nul, then Terminator_Error is propagated.

55

```
function Is_Nul_Terminated (Item : in wchar_array) return Boolean;
```

56

The result of Is_Nul_Terminated is True if Item contains wide_nul, and is False otherwise.

57

```
function To_C (Item : in Wide_Character) return wchar_t;  
function To_Ada (Item : in wchar_t ) return Wide_Character;
```

58

To_C and To_Ada provide the mappings between the Ada and C wide character types.

59

```
function To_C (Item      : in Wide_String;  
              Append_Nul : in Boolean := True)  
  return wchar_array;  
  
function To_Ada (Item      : in wchar_array;  
               Trim_Nul  : in Boolean := True)  
  return Wide_String;  
  
procedure To_C (Item      : in Wide_String;  
              Target    : out wchar_array;  
              Count     : out size_t;  
              Append_Nul : in Boolean := True);  
  
procedure To_Ada (Item      : in wchar_array;  
                Target    : out Wide_String;  
                Count     : out Natural;  
                Trim_Nul  : in Boolean := True);
```

60

The To_C and To_Ada subprograms that convert between Wide_String and wchar_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that wide_nul is used instead of nul.

60.1/2

```
function Is_Nul_Terminated (Item : in char16_array) return Boolean;
```

60.2/2

The result of Is_Nul_Terminated is True if Item contains char16_nul, and is False otherwise.

60.3/2

```
function To_C (Item : in Wide_Character) return char16_t;  
function To_Ada (Item : in char16_t ) return Wide_Character;
```

60.4/2

To_C and To_Ada provide mappings between the Ada and C 16-bit character types.

60.5/2

```
function To_C (Item          : in Wide_String;  
              Append_Nul   : in Boolean := True)  
  return char16_array;  
  
function To_Ada (Item       : in char16_array;  
               Trim_Nul    : in Boolean := True)  
  return Wide_String;  
  
procedure To_C (Item          : in Wide_String;  
              Target        : out char16_array;  
              Count        : out size_t;  
              Append_Nul    : in Boolean := True);  
  
procedure To_Ada (Item       : in char16_array;  
                Target      : out Wide_String;  
                Count       : out Natural;  
                Trim_Nul    : in Boolean := True);
```

60.6/2

The To_C and To_Ada subprograms that convert between Wide_String and char16_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char16_nul is used instead of nul.

60.7/2

```
function Is_Nul_Terminated (Item : in char32_array) return Boolean;
```

60.8/2

The result of Is_Nul_Terminated is True if Item contains char16_nul, and is False otherwise.

60.9/2

```
function To_C (Item : in Wide_Wide_Character) return char32_t;  
function To_Ada (Item : in char32_t ) return Wide_Wide_Character;
```

60.10/2

To_C and To_Ada provide mappings between the Ada and C 32-bit character types.

60.11/2

```
function To_C (Item          : in Wide_Wide_String;  
              Append_Nul   : in Boolean := True)  
  return char32_array;  
  
function To_Ada (Item       : in char32_array;  
               Trim_Nul    : in Boolean := True)  
  return Wide_Wide_String;  
  
procedure To_C (Item          : in Wide_Wide_String;  
              Target         : out char32_array;  
              Count          : out size_t;  
              Append_Nul    : in Boolean := True);  
  
procedure To_Ada (Item       : in char32_array;  
               Target       : out Wide_Wide_String;  
               Count        : out Natural;  
               Trim_Nul     : in Boolean := True);
```

60.12/2

The To_C and To_Ada subprograms that convert between Wide_Wide_String and char32_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char32_nul is used instead of nul.

60.13/1

A Convention pragma with <convention>_identifier C_Pass_By_Copy shall only be applied to a type.

60.14/2

The eligibility rules in Section 16.1 [B.1], page 894, do not apply to convention C_Pass_By_Copy. Instead, a type T is eligible for convention C_Pass_By_Copy if T is an unchecked union type or if T is a record type that has no discriminants and that only has components with statically constrained subtypes, and each component is C-compatible.

60.15/1

If a type is C_Pass_By_Copy-compatible then it is also C-compatible.

Implementation Requirements

61/1

An implementation shall support pragma Convention with a C <convention>_identifier for a C-eligible type (see Section 16.1 [B.1], page 894). An implementation shall support pragma Convention with a C_Pass_By_Copy <convention>_identifier for a C_Pass_By_Copy-eligible type.

Implementation Permissions

62

An implementation may provide additional declarations in the C interface packages.

Implementation Advice

62.1/2

The constants nul, wide_nul, char16_nul, and char32_nul should have a representation of zero.

63

An implementation should support the following interface correspondences between Ada and C.

64

- An Ada procedure corresponds to a void-returning C function.

65

- An Ada function corresponds to a non-void C function.

66

- An Ada in scalar parameter is passed as a scalar argument to a C function.

67

- An Ada in parameter of an access-to-object type with designated type T is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.

68

- An Ada access T parameter, or an Ada out or in out parameter of an elementary type T, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T. In the case of an elementary out or in out parameter, a pointer to a temporary copy is used to preserve by-copy semantics.

68.1/2

- An Ada parameter of a (record) type T of convention C_Pass_By_Copy, of mode in, is passed as a t argument to a C function, where t is the C struct corresponding to the Ada type T.

69/2

- An Ada parameter of a record type T, of any mode, other than an in parameter of a type of convention C_Pass_By_Copy, is passed as a t* argument to a C function, where t is the C struct corresponding to the Ada type T.

70

- An Ada parameter of an array type with component type T, of any mode, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.

71

- An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification.

71.1/2

An Ada parameter of a private type is passed as specified for the full view of the type.

NOTES

72

8 Values of type char_array are not implicitly terminated with nul. If a char_array is to be passed as a parameter to an imported C function requiring nul termination, it is the programmer's responsibility to obtain this effect.

73

9 To obtain the effect of C's sizeof(item_type), where Item_Type is the corresponding Ada type, evaluate the expression: size_t(Item_Type'Size/CHAR_BIT).

74/2

<This paragraph was deleted.>

75

10 A C function that takes a variable number of arguments can correspond to several Ada subprograms, taking various specific numbers and types of parameters.

Examples

76

<Example of using the Interfaces.C package:>

77

```
<--Calling the C Library Function strcpy>  
with Interfaces.C;
```

```

78      procedure Test is
          package C renames Interfaces.C;
          use type C.char_array;
          <-- Call <string.h>strcpy:>
          <-- C definition of strcpy: char *strcpy(char *s1, const char *s2);>
          <-- This function copies the string pointed to by s2 (including the termina
          <-- into the array pointed to by s1. If copying takes place between object
          <-- the behavior is undefined. The strcpy function returns the value of s1

          <-- Note: since the C function's return value is of no interest, the Ada inter
          procedure Strcpy (Target : out C.char_array;
                          Source : in C.char_array);

79      pragma Import(C, Strcpy, "strcpy");

80
          Chars1 : C.char_array(1..20);
          Chars2 : C.char_array(1..20);

81
          begin
              Chars2(1..6) := "qwert" & C.nul;

82
              Strcpy(Chars1, Chars2);

83
              <-- Now Chars1(1..6) = "qwert" & C.Nul>

84
          end Test;

```

16.3.1 B.3.1 The Package Interfaces.C.Strings

1
The package Interfaces.C.Strings declares types and subprograms allowing an Ada program to allocate, reference, update, and free C-style strings. In particular, the private type chars_ptr corresponds to a common use of "char *" in C programs, and an object of this type can be passed to a subprogram to which pragma Import(C,...) has been applied, and for which "char *" is the type of the argument of the C function.

Static Semantics

2
The library package Interfaces.C.Strings has the following declaration:

3

```
package Interfaces.C.Strings is
  pragma Preelaborate(Strings);
```

4

```
  type char_array_access is access all char_array;
```

5/2

```
  type chars_ptr is private;
  pragma Preelaborable_Initialization(chars_ptr);
```

6/2

```
  type chars_ptr_array is array (size_t range <>) of aliased chars_ptr;■
```

7

```
  Null_Ptr : constant chars_ptr;
```

8

```
  function To_Chars_Ptr (Item      : in char_array_access;
                        Nul_Check : in Boolean := False)
    return chars_ptr;
```

9

```
  function New_Char_Array (Chars  : in char_array) return chars_ptr;■
```

10

```
  function New_String (Str : in String) return chars_ptr;
```

11

```
  procedure Free (Item : in out chars_ptr);
```

12

```
  Dereference_Error : exception;
```

13

```
  function Value (Item : in chars_ptr) return char_array;
```

14

```
  function Value (Item : in chars_ptr; Length : in size_t)
    return char_array;
```

15

```
function Value (Item : in chars_ptr) return String;
```

16

```
function Value (Item : in chars_ptr; Length : in size_t)
  return String;
```

17

```
function Strlen (Item : in chars_ptr) return size_t;
```

18

```
procedure Update (Item    : in chars_ptr;
                 Offset  : in size_t;
                 Chars   : in char_array;
                 Check   : in Boolean := True);
```

19

```
procedure Update (Item    : in chars_ptr;
                 Offset  : in size_t;
                 Str     : in String;
                 Check   : in Boolean := True);
```

20

```
Update_Error : exception;
```

21

```
private
  ... -- <not specified by the language>
end Interfaces.C.Strings;
```

22

The type `chars_ptr` is C-compatible and corresponds to the use of C's "char *" for a pointer to the first char in a char array terminated by nul. When an object of type `chars_ptr` is declared, its value is by default set to `Null_Ptr`, unless the object is imported (see Section 16.1 [B.1], page 894).

23

```
function To_Chars_Ptr (Item      : in char_array_access;
                     Nul_Check : in Boolean := False)
  return chars_ptr;
```

24/1

If `Item` is null, then `To_Chars_Ptr` returns `Null_Ptr`. If `Item` is not null, `Nul_Check` is `True`, and `Item.all` does not contain `nul`, then the function propagates `Terminator_Error`; otherwise `To_Chars_Ptr` performs a pointer conversion with no allocation of memory.

25

```
function New_Char_Array (Chars : in char_array) return chars_ptr;
```

26

This function returns a pointer to an allocated object initialized to `Chars(Chars'First .. Index) & nul`, where

27

- `Index = Chars'Last` if `Chars` does not contain `nul`, or

28

- `Index` is the smallest `size_t` value `I` such that `Chars(I+1) = nul`.

28.1

`Storage_Error` is propagated if the allocation fails.

29

```
function New_String (Str : in String) return chars_ptr;
```

30

This function is equivalent to `New_Char_Array(To_C(Str))`.

31

```
procedure Free (Item : in out chars_ptr);
```

32

If `Item` is `Null_Ptr`, then `Free` has no effect. Otherwise, `Free` releases the storage occupied by `Value(Item)`, and resets `Item` to `Null_Ptr`.

33

```
function Value (Item : in chars_ptr) return char_array;
```

34

If Item = Null_Ptr then Value propagates Dereference_Error. Otherwise Value returns the prefix of the array of chars pointed to by Item, up to and including the first nul. The lower bound of the result is 0. If Item does not point to a nul-terminated string, then execution of Value is erroneous.

35

```
function Value (Item : in chars_ptr; Length : in size_t)
  return char_array;
```

36/1

If Item = Null_Ptr then Value propagates Dereference_Error. Otherwise Value returns the shorter of two arrays, either the first Length chars pointed to by Item, or Value(Item). The lower bound of the result is 0. If Length is 0, then Value propagates Constraint_Error.

37

```
function Value (Item : in chars_ptr) return String;
```

38

Equivalent to To_Ada(Value(Item),
Trim_Nul=>True).

39

```
function Value (Item : in chars_ptr; Length : in size_t)
  return String;
```

40/1

Equivalent to To_Ada(Value(Item, Length)
& nul, Trim_Nul=>True).

41

```
function Strlen (Item : in chars_ptr) return size_t;
```

42

Returns $\langle \text{Val} \rangle' \text{Length} - 1$ where $\langle \text{Val} \rangle = \text{Value}(\text{Item})$; propagates `Dereference_Error` if `Item = Null_Ptr`.

43

```

procedure Update (Item   : in chars_ptr;
                  Offset : in size_t;
                  Chars  : in char_array;
                  Check   : Boolean := True);

```

44/1

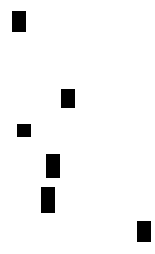
If `Item = Null_Ptr`, then `Update` propagates `Dereference_Error`. Otherwise, this procedure updates the value pointed to by `Item`, starting at position `Offset`, using `Chars` as the data to be copied into the array. Overwriting the nul terminator, and skipping with the `Offset` past the nul terminator, are both prevented if `Check` is `True`, as follows:

45

- Let $N = \text{Strlen}(\text{Item})$. If `Check` is `True`, then:

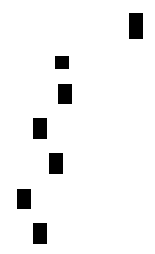
46

- If $\text{Offset} + \text{Chars}' \text{Length} > N$, propagate `Update_Error`.

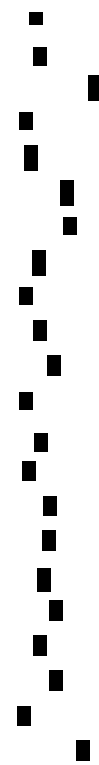


47

- Otherwise, overwrite the data in the



array pointed to by Item, starting at the character at position Offset, with the data in Chars.



48

- If Check is False, then processing is as above, but with no check that $\text{Offset} + \text{Chars}'\text{Length} > N$.

49

```
procedure Update (Item   : in chars_ptr;  
                  Offset : in size_t;  
                  Str    : in String;  
                  Check  : in Boolean := True);
```

50/2

Equivalent to `Update(Item, Offset, To_C(Str, Append_Nul => False), Check)`.
Erroneous Execution

51

Execution of any of the following is erroneous if the Item parameter is not null_ptr and Item does not point to a nul-terminated array of chars.

52

- a Value function not taking a Length parameter,

53

- the Free procedure,

54

- the Strlen function.

55

Execution of Free(X) is also erroneous if the chars_ptr X was not returned by New_Char_Array or New_String.

56

Reading or updating a freed char_array is erroneous.

57

Execution of Update is erroneous if Check is False and a call with Check equal to True would have propagated Update_Error.

NOTES

58

11 New_Char_Array and New_String might be implemented either through the allocation function from the C environment ("malloc") or through Ada dynamic memory allocation ("new"). The key points are

59

- the returned value (a chars_ptr) is represented as a C "char *" so that it may be passed to C functions;

60

- the allocated object should be freed by the programmer via a call of Free, not by a called C function.

16.3.2 B.3.2 The Generic Package Interfaces.C.Pointers

1

The generic package Interfaces.C.Pointers allows the Ada programmer to perform C-style operations on pointers. It includes an access type Pointer, Value functions that dereference a Pointer and deliver the designated array, several pointer arithmetic operations, and "copy" procedures that copy the contents of a source pointer into the array designated by a destination pointer. As in C, it treats an object Ptr of type Pointer as a pointer to the first element of an array, so that for example, adding 1 to Ptr yields a pointer to the second element of the array.

2

The generic allows two styles of usage: one in which the array is terminated by a special terminator element; and another in which the programmer needs to keep track of the length.

Static Semantics

3

The generic library package Interfaces.C.Pointers has the following declaration:

4

```
generic
  type Index is (<>);
  type Element is private;
  type Element_Array is array (Index range <>) of aliased Element;
  Default_Terminator : Element;
package Interfaces.C.Pointers is
  pragma Preelaborate(Pointers);
```

5

```
type Pointer is access all Element;
```

6

```
function Value(Ref          : in Pointer;
               Terminator : in Element := Default_Terminator)
  return Element_Array;
```

7

```
function Value(Ref      : in Pointer;
               Length : in ptrdiff_t)
  return Element_Array;
```

8

```
Pointer_Error : exception;
```

9

```
<-- C-style Pointer arithmetic>
```

10

```
function "+" (Left : in Pointer;   Right : in ptrdiff_t) return Pointer;█
function "+" (Left : in ptrdiff_t; Right : in Pointer)   return Pointer;█
function "-" (Left : in Pointer;   Right : in ptrdiff_t) return Pointer;█
function "-" (Left : in Pointer;   Right : in Pointer)   return ptrdiff_t;█
```

11

```
procedure Increment (Ref : in out Pointer);
procedure Decrement (Ref : in out Pointer);
```

12

```
pragma Convention (Intrinsic, "+");
```

```
pragma Convention (Intrinsic, "-");
pragma Convention (Intrinsic, Increment);
pragma Convention (Intrinsic, Decrement);
```

13

```
function Virtual_Length (Ref          : in Pointer;
                        Terminator : in Element := Default_Terminator)
return ptrdiff_t;
```

14

```
procedure Copy_Terminated_Array
(Source      : in Pointer;
Target      : in Pointer;
Limit       : in ptrdiff_t := ptrdiff_t'Last;
Terminator  : in Element := Default_Terminator);
```

15

```
procedure Copy_Array (Source : in Pointer;
                     Target  : in Pointer;
                     Length  : in ptrdiff_t);
```

16

```
end Interfaces.C.Pointers;
```

17

The type `Pointer` is C-compatible and corresponds to one use of C's "Element *". An object of type `Pointer` is interpreted as a pointer to the initial `Element` in an `Element_Array`. Two styles are supported:

18

- Explicit termination of an array value with `Default_Terminator` (a special terminator value);

19

- Programmer-managed length, with `Default_Terminator` treated simply as a data element.

20

```
function Value(Ref          : in Pointer;
              Terminator : in Element := Default_Terminator)
return Element_Array;
```

21

This function returns an `Element_Array` whose value is the array pointed to by `Ref`, up to and including the first `Terminator`; the lower bound of the array is `Index'First`. `Interfaces.C.Strings.Dereference_Error` is propagated if `Ref` is null.

22

```
function Value(Ref      : in Pointer;
              Length   : in ptrdiff_t)
  return Element_Array;
```

23

This function returns an `Element_Array` comprising the first `Length` elements pointed to by `Ref`. The exception `Interfaces.C.Strings.Dereference_Error` is propagated if `Ref` is null.

24

The "+" and "-" functions perform arithmetic on `Pointer` values, based on the `Size` of the array elements. In each of these functions, `Pointer_Error` is propagated if a `Pointer` parameter is null.

25

```
procedure Increment (Ref : in out Pointer);
```

26

Equivalent to `Ref := Ref+1`.

27

```
procedure Decrement (Ref : in out Pointer);
```

28

Equivalent to `Ref := Ref-1`.

29

```
function Virtual_Length (Ref          : in Pointer;
                       Terminator    : in Element := Default_Terminator)
  return ptrdiff_t;
```

30

Returns the number of `Elements`, up to the one just before the first `Terminator`, in `Value(Ref, Terminator)`.

31

```
procedure Copy_Terminated_Array
  (Source      : in Pointer;
   Target      : in Pointer;
   Limit       : in ptrdiff_t := ptrdiff_t'Last;
   Terminator  : in Element := Default_Terminator);
```

32

This procedure copies Value(Source, Terminator) into the array pointed to by Target; it stops either after Terminator has been copied, or the number of elements copied is Limit, whichever occurs first. Dereference_Error is propagated if either Source or Target is null.

33

```
procedure Copy_Array (Source : in Pointer;
                     Target  : in Pointer;
                     Length  : in ptrdiff_t);
```

34

This procedure copies the first Length elements from the array pointed to by Source, into the array pointed to by Target. Dereference_Error is propagated if either Source or Target is null.

Erroneous Execution

35

It is erroneous to dereference a Pointer that does not designate an aliased Element.

36

Execution of Value(Ref, Terminator) is erroneous if Ref does not designate an aliased Element in an Element_Array terminated by Terminator.

37

Execution of Value(Ref, Length) is erroneous if Ref does not designate an aliased Element in an Element_Array containing at least Length Elements between the designated Element and the end of the array, inclusive.

38

Execution of Virtual_Length(Ref, Terminator) is erroneous if Ref does not designate an aliased Element in an Element_Array terminated by Terminator.

39

Execution of Copy_Terminated_Array(Source, Target, Limit, Terminator) is erroneous in either of the following situations:

40

- Execution of both Value(Source, Terminator) and Value(Source, Limit) are erroneous, or

41

- Copying writes past the end of the array containing the Element designated by Target.

42

Execution of Copy_Array(Source, Target, Length) is erroneous if either Value(Source, Length) is erroneous, or copying writes past the end of the array containing the Element designated by Target.

NOTES

43

12 To compose a Pointer from an Element_Array, use 'Access on the first element. For example (assuming appropriate instantiations):

44

```
Some_Array    : Element_Array(0..5) ;
Some_Pointer  : Pointer := Some_Array(0)'Access;
               Examples
```

45

<Example of Interfaces.C.Pointers:>

46

```
with Interfaces.C.Pointers;
with Interfaces.C.Strings;
procedure Test_Pointers is
  package C renames Interfaces.C;
  package Char_Ptrs is
    new C.Pointers (Index           => C.size_t,
                   Element         => C.char,
                   Element_Array   => C.char_array,
                   Default_Terminator => C.nul);
```

47

```
use type Char_Ptrs.Pointer;
subtype Char_Star is Char_Ptrs.Pointer;
```

48

```
procedure Strcpy (Target_Ptr, Source_Ptr : Char_Star) is
  Target_Temp_Ptr : Char_Star := Target_Ptr;
  Source_Temp_Ptr : Char_Star := Source_Ptr;
```

```

        Element : C.char;
begin
    if Target_Temp_Ptr = null or Source_Temp_Ptr = null then
        raise C.Strings.Dereference_Error;
    end if;

    loop
        Element           := Source_Temp_Ptr.all;
        Target_Temp_Ptr.all := Element;
        exit when C."="(Element, C.nul);
        Char_Ptrs.Increment(Target_Temp_Ptr);
        Char_Ptrs.Increment(Source_Temp_Ptr);
    end loop;
end Strcpy;
begin
    ...
end Test_Pointers;

```

49/1

16.3.3 B.3.3 Pragma Unchecked_Union

1/2

A pragma `Unchecked_Union` specifies an interface correspondence between a given discriminated type and some C union. The pragma specifies that the associated type shall be given a representation that leaves no space for its discriminant(s).

Syntax

2/2

The form of a pragma `Unchecked_Union` is as follows:

3/2

```
pragma Unchecked_Union (<first_subtype>local_name);
```

Legality Rules

4/2

`Unchecked_Union` is a representation pragma, specifying the unchecked union aspect of representation.

5/2

The `<first_subtype>local_name` of a pragma `Unchecked_Union` shall denote an unconstrained discriminated record subtype having a variant part.

6/2

A type to which a pragma `Unchecked_Union` applies is called an `<unchecked union type>`. A subtype of an unchecked union type is defined to be an `<unchecked union subtype>`. An object of an unchecked union type is defined to be an `<unchecked union object>`.

7/2

All component subtypes of an unchecked union type shall be C-compatible.

8/2

If a component subtype of an unchecked union type is subject to a per-object constraint, then the component subtype shall be an unchecked union subtype.

9/2

Any name that denotes a discriminant of an object of an unchecked union type shall occur within the declarative region of the type.

10/2

A component declared in a `variant_part` of an unchecked union type shall not have a controlled, protected, or task part.

11/2

The completion of an incomplete or private type declaration having a `known_discriminant_part` shall not be an unchecked union type.

12/2

An unchecked union subtype shall only be passed as a generic actual parameter if the corresponding formal type has no known discriminants or is an unchecked union type.

Static Semantics

13/2

An unchecked union type is eligible for convention C.

14/2

All objects of an unchecked union type have the same size.

15/2

Discriminants of objects of an unchecked union type are of size zero.

16/2

Any check which would require reading a discriminant of an unchecked union object is suppressed (see Section 12.5 [11.5], page 431). These checks include:

17/2

- The check performed when addressing a variant component (i.e., a component that was declared in a variant part) of an unchecked union object that the object has this component (see Section 5.1.3 [4.1.3], page 183).

18/2

- Any checks associated with a type or subtype conversion of a value of an unchecked union type (see Section 5.6 [4.6], page 219). This includes, for example, the check associated with the implicit subtype conversion of an assignment statement.

19/2

- The subtype membership check associated with the evaluation of a qualified expression (see Section 5.7 [4.7], page 229) or an uninitialized allocator (see Section 5.8 [4.8], page 230).

Dynamic Semantics

20/2

A view of an unchecked union object (including a type conversion or function call) has

<inferable discriminants> if it has a constrained nominal subtype, unless the object is a component of an enclosing unchecked union object that is subject to a per-object constraint and the enclosing object lacks inferable discriminants.

21/2

An expression of an unchecked union type has inferable discriminants if it is either a name of an object with inferable discriminants or a qualified expression whose subtype_mark denotes a constrained subtype.

22/2

Program_Error is raised in the following cases:

23/2

- Evaluation of the predefined equality operator for an unchecked union type if either of the operands lacks inferable discriminants.

24/2

- Evaluation of the predefined equality operator for a type which has a subcomponent of an unchecked union type whose nominal subtype is unconstrained.

25/2

- Evaluation of a membership test if the subtype_mark denotes a constrained unchecked union subtype and the expression lacks inferable discriminants.

26/2

- Conversion from a derived unchecked union type to an unconstrained non-union type if the operand of the conversion lacks inferable discriminants.

27/2

- Execution of the default implementation of the Write or Read attribute of an unchecked union type.

28/2

- Execution of the default implementation of the Output or Input attribute of an unchecked union type if the type lacks default discriminant values.

Implementation Permissions

29/2

An implementation may require that pragma Controlled be specified for the type of an access subcomponent of an unchecked union type.

NOTES

30/2

13 The use of an unchecked union to obtain the effect of an unchecked conversion results in erroneous execution (see Section 12.5 [11.5], page 431). Execution of the following example is erroneous even if `Float'Size = Integer'Size`:

31/2

```
type T (Flag : Boolean := False) is
  record
    case Flag is
      when False =>
        F1 : Float := 0.0;
      when True =>
        F2 : Integer := 0;
    end case;
  end record;
pragma Unchecked_Union (T);
```

32/2

```
X : T;
Y : Integer := X.F2; -- <erroneous>
```

16.4 B.4 Interfacing with COBOL

1

The facilities relevant to interfacing with the COBOL language are the package `Interfaces.COBOL` and support for the `Import`, `Export` and `Convention` pragmas with `<convention>_identifier COBOL`.

2

The COBOL interface package supplies several sets of facilities:

3

- A set of types corresponding to the native COBOL types of the supported COBOL implementation (so-called "internal COBOL representations"), allowing Ada data to be passed as parameters to COBOL programs

4

- A set of types and constants reflecting external data representations such as might be found in files or databases, allowing COBOL-generated data to be read by an Ada program, and Ada-generated data to be read by COBOL programs

5

- A generic package for converting between an Ada decimal type value and either an internal or external COBOL representation

Static Semantics

6

The library package Interfaces.COBOL has the following declaration:

7

```
package Interfaces.COBOL is
  pragma Preelaborate(COBOL);
```

8

```
<-- Types and operations for internal data representations>
```

9

```
type Floating      is digits <implementation-defined>;
type Long_Floating is digits <implementation-defined>;
```

10

```
type Binary        is range <implementation-defined>;
type Long_Binary   is range <implementation-defined>;
```

11

```
Max_Digits_Binary      : constant := <implementation-defined>;
Max_Digits_Long_Binary : constant := <implementation-defined>;
```

12

```
type Decimal_Element is mod <implementation-defined>;
type Packed_Decimal is array (Positive range <>) of Decimal_Element;
pragma Pack(Packed_Decimal);
```

13

```
type COBOL_Character is <implementation-defined character type>;
```

14

```
Ada_To_COBOL : array (Character) of COBOL_Character := <implementation-defined>;
```

15

```
COBOL_To_Ada : array (COBOL_Character) of Character := <implementation-defined>;
```

16

```
type Alphanumeric is array (Positive range <>) of COBOL_Character;
pragma Pack(Alphanumeric);
```

17

```
function To_COBOL (Item : in String) return Alphanumeric;  
function To_Ada   (Item : in Alphanumeric) return String;
```

18

```
procedure To_COBOL (Item      : in String;  
                   Target    : out Alphanumeric;  
                   Last      : out Natural);
```

19

```
procedure To_Ada (Item      : in Alphanumeric;  
                 Target    : out String;  
                 Last      : out Natural);
```

20

```
type Numeric is array (Positive range <>) of COBOL_Character;  
pragma Pack(Numeric);
```

21

```
<-- Formats for COBOL data representations>
```

22

```
type Display_Format is private;
```

23

```
Unsigned           : constant Display_Format;  
Leading_Separate   : constant Display_Format;  
Trailing_Separate  : constant Display_Format;  
Leading_Nonseparate : constant Display_Format;  
Trailing_Nonseparate : constant Display_Format;
```

24

```
type Binary_Format is private;
```

25

```
High_Order_First  : constant Binary_Format;  
Low_Order_First   : constant Binary_Format;  
Native_Binary     : constant Binary_Format;
```

26

```
type Packed_Format is private;
```

27

```
Packed_Unsigned    : constant Packed_Format;  
Packed_Signed      : constant Packed_Format;
```

28

```
<-- Types for external representation of COBOL binary data>
```

29

```
type Byte is mod 2**COBOL_Character'Size;  
type Byte_Array is array (Positive range <>) of Byte;  
pragma Pack (Byte_Array);
```

30

```
Conversion_Error : exception;
```

31

```
generic  
  type Num is delta <> digits <>;  
package Decimal_Conversions is
```

32

```
<-- Display Formats: data values are represented as Numeric>
```

33

```
function Valid (Item    : in Numeric;  
               Format   : in Display_Format) return Boolean;
```

34

```
function Length (Format : in Display_Format) return Natural;
```

35

```
function To_Decimal (Item    : in Numeric;  
                   Format   : in Display_Format) return Num;
```

36

```
function To_Display (Item    : in Num;  
                   Format   : in Display_Format) return Numeric;■
```

37

```
<-- Packed Formats: data values are represented as Packed_Decimal>■
```

38

```
function Valid (Item    : in Packed_Decimal;  
               Format  : in Packed_Format) return Boolean;
```

39

```
function Length (Format : in Packed_Format) return Natural;
```

40

```
function To_Decimal (Item    : in Packed_Decimal;  
                   Format  : in Packed_Format) return Num;
```

41

```
function To_Packed (Item    : in Num;  
                  Format  : in Packed_Format) return Packed_Decimal;■
```

42

```
<-- Binary Formats: external data values are represented as Byte_Array>■
```

43

```
function Valid (Item    : in Byte_Array;  
               Format  : in Binary_Format) return Boolean;
```

44

```
function Length (Format : in Binary_Format) return Natural;  
function To_Decimal (Item    : in Byte_Array;  
                   Format  : in Binary_Format) return Num;
```

45

```
function To_Binary (Item    : in Num;  
                  Format  : in Binary_Format) return Byte_Array;■
```

46

```
<-- Internal Binary formats: data values are of type Binary or Long_Binary>
```

47

```
function To_Decimal (Item : in Binary)      return Num;  
function To_Decimal (Item : in Long_Binary) return Num;
```

48

```
function To_Binary      (Item : in Num) return Binary;  
function To_Long_Binary (Item : in Num) return Long_Binary;
```

49

```
end Decimal_Conversions;
```

50

```
private
  ... -- <not specified by the language>
end Interfaces.COBOL;
```

51

Each of the types in Interfaces.COBOL is COBOL-compatible.

52

The types Floating and Long_Floating correspond to the native types in COBOL for data items with computational usage implemented by floating point. The types Binary and Long_Binary correspond to the native types in COBOL for data items with binary usage, or with computational usage implemented by binary.

53

Max_Digits_Binary is the largest number of decimal digits in a numeric value that is represented as Binary. Max_Digits_Long_Binary is the largest number of decimal digits in a numeric value that is represented as Long_Binary.

54

The type Packed_Decimal corresponds to COBOL's packed-decimal usage.

55

The type COBOL_Character defines the run-time character set used in the COBOL implementation. Ada_To_COBOL and COBOL_To_Ada are the mappings between the Ada and COBOL run-time character sets.

56

Type Alphanumeric corresponds to COBOL's alphanumeric data category.

57

Each of the functions To_COBOL and To_Ada converts its parameter based on the mappings Ada_To_COBOL and COBOL_To_Ada, respectively. The length of the result for each is the length of the parameter, and the lower bound of the result is 1. Each component of the result is obtained by applying the relevant mapping to the corresponding component of the parameter.

58

Each of the procedures To_COBOL and To_Ada copies converted elements from Item to Target, using the appropriate mapping (Ada_To_COBOL or COBOL_To_Ada, respectively). The index in Target of the last element assigned is returned in Last (0 if Item is a null array). If Item'Length exceeds Target'Length, Constraint_Error is propagated.

59

Type Numeric corresponds to COBOL's numeric data category with display usage.

60

The types Display_Format, Binary_Format, and Packed_Format are used in conversions between Ada decimal type values and COBOL internal or external data representations.

The value of the constant `Native_Binary` is either `High_Order_First` or `Low_Order_First`, depending on the implementation.

61

```
function Valid (Item    : in Numeric;  
               Format   : in Display_Format) return Boolean;
```

62

The function `Valid` checks that the `Item` parameter has a value consistent with the value of `Format`. If the value of `Format` is other than `Unsigned`, `Leading_Separate`, and `Trailing_Separate`, the effect is implementation defined. If `Format` does have one of these values, the following rules apply:

63/1

- `Format=Unsigned`: if `Item` comprises one or more decimal digit characters then `Valid` returns `True`, else it returns `False`.

64/1

- `Format=Leading_Separate`: if `Item` comprises a single occurrence of the plus or minus sign character, and then one or more decimal digit characters, then `Valid` returns `True`, else it returns `False`.

65/1

- `Format=Trailing_Separate`: if `Item` comprises one or more decimal digit characters and finally a plus or minus sign character, then `Valid` returns `True`, else it returns `False`.

66

```
function Length (Format : in Display_Format) return Natural;
```

67

The `Length` function returns the minimal length of a `Numeric` value sufficient to hold

any value of type Num when represented as Format.

68

```
function To_Decimal (Item    : in Numeric;  
                    Format   : in Display_Format) return Num;
```

69

Produces a value of type Num corresponding to Item as represented by Format. The number of digits after the assumed radix point in Item is Num'Scale. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

70

```
function To_Display (Item    : in Num;  
                   Format   : in Display_Format) return Numeric;
```

71/1

This function returns the Numeric value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Num is negative and Format is Unsigned.

72

```
function Valid (Item    : in Packed_Decimal;  
              Format   : in Packed_Format) return Boolean;
```

73

This function returns True if Item has a value consistent with Format, and False otherwise. The rules for the formation of Packed_Decimal values are implementation defined.

74

```
function Length (Format : in Packed_Format) return Natural;
```

75

This function returns the minimal length of a Packed_Decimal value sufficient to hold any

value of type Num when represented as Format.

76

```
function To_Decimal (Item    : in Packed_Decimal;  
                    Format   : in Packed_Format) return Num;
```

77

Produces a value of type Num corresponding to Item as represented by Format. Num'Scale is the number of digits after the assumed radix point in Item. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

78

```
function To_Packed (Item    : in Num;  
                  Format   : in Packed_Format) return Packed_Decimal;■
```

79/1

This function returns the Packed_Decimal value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Num is negative and Format is Packed_Unsigned.

80

```
function Valid (Item    : in Byte_Array;  
              Format   : in Binary_Format) return Boolean;
```

81

This function returns True if Item has a value consistent with Format, and False otherwise.

82

```
function Length (Format : in Binary_Format) return Natural;
```

83

This function returns the minimal length of a Byte_Array value sufficient to hold any value of type Num when represented as Format.

84

```
function To_Decimal (Item    : in Byte_Array;  
                    Format  : in Binary_Format) return Num;
```

85

Produces a value of type Num corresponding to Item as represented by Format. Num'Scale is the number of digits after the assumed radix point in Item. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

86

```
function To_Binary (Item    : in Num;  
                  Format  : in Binary_Format) return Byte_Array;
```

87/1

This function returns the Byte_Array value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1.

88

```
function To_Decimal (Item : in Binary)      return Num;
```

```
function To_Decimal (Item : in Long_Binary) return Num;
```

89

These functions convert from COBOL binary format to a corresponding value of the decimal type Num. Conversion_Error is propagated if Item is too large for Num.

90

```
function To_Binary      (Item : in Num) return Binary;
```

```
function To_Long_Binary (Item : in Num) return Long_Binary;
```

91

These functions convert from Ada decimal to COBOL binary format. Conversion_Error is propagated if the value of Item is too large to be represented in the result type.

Implementation Requirements

92

An implementation shall support pragma Convention with a COBOL <convention>_identifier for a COBOL-eligible type (see Section 16.1 [B.1], page 894).

Implementation Permissions

93

An implementation may provide additional constants of the private types Display_Format, Binary_Format, or Packed_Format.

94

An implementation may provide further floating point and integer types in Interfaces.COBOLE to match additional native COBOL types, and may also supply corresponding conversion functions in the generic package Decimal_Conversions.

Implementation Advice

95

An Ada implementation should support the following interface correspondences between Ada and COBOL.

96

- An Ada access T parameter is passed as a "BY REFERENCE" data item of the COBOL type corresponding to T.

97

- An Ada in scalar parameter is passed as a "BY CONTENT" data item of the corresponding COBOL type.

98

- Any other Ada parameter is passed as a "BY REFERENCE" data item of the COBOL type corresponding to the Ada parameter type; for scalars, a local copy is used if necessary to ensure by-copy semantics.

NOTES

99

14 An implementation is not required to support pragma Convention for access types, nor is it required to support pragma Import, Export or Convention for functions.

100

15 If an Ada subprogram is exported to COBOL, then a call from COBOL call may specify either "BY CONTENT" or "BY REFERENCE".

Examples

101

<Examples of Interfaces.COBOLE:>

102

```
with Interfaces.COBOL;  
procedure Test_Call is
```

103

```
<-- Calling a foreign COBOL program>  
<-- Assume that a COBOL program PROG has the following declaration>█  
<-- in its LINKAGE section:>  
<-- 01 Parameter-Area>  
<--     05 NAME    PIC X(20).>  
<--     05 SSN    PIC X(9).>  
<--     05 SALARY PIC 99999V99 USAGE COMP.>  
<-- The effect of PROG is to update SALARY based on some algorithm>█
```

104

```
package COBOL renames Interfaces.COBOL;
```

105

```
type Salary_Type is delta 0.01 digits 7;
```

106

```
type COBOL_Record is  
  record  
    Name    : COBOL.Numeric(1..20);  
    SSN     : COBOL.Numeric(1..9);  
    Salary  : COBOL.Binary; <-- Assume Binary = 32 bits>  
  end record;  
pragma Convention (COBOL, COBOL_Record);
```

107

```
procedure Prog (Item : in out COBOL_Record);  
pragma Import (COBOL, Prog, "PROG");
```

108

```
package Salary_Conversions is  
  new COBOL.Decimal_Conversions(Salary_Type);
```

109

```
Some_Salary : Salary_Type := 12_345.67;  
Some_Record : COBOL_Record :=  
  (Name => "Johnson, John",  
   SSN  => "111223333",
```

```

110         Salary => Salary_Conversions.To_Binary(Some_Salary));

begin
    Prog (Some_Record);
    ...
end Test_Call;

111

with Interfaces.COBOL;
with COBOL_Sequential_IO; <-- Assumed to be supplied by implementation>
procedure Test_External_Formats is

112

    <-- Using data created by a COBOL program>
    <-- Assume that a COBOL program has created a sequential file with>
    <-- the following record structure, and that we need to>
    <-- process the records in an Ada program>
    <-- 01 EMPLOYEE-RECORD>
    <--     05 NAME      PIC X(20).>
    <--     05 SSN      PIC X(9).>
    <--     05 SALARY   PIC 99999V99 USAGE COMP.>
    <--     05 ADJUST   PIC S999V999 SIGN LEADING SEPARATE.>
    <-- The COMP data is binary (32 bits), high-order byte first>

113

package COBOL renames Interfaces.COBOL;

114

type Salary_Type      is delta 0.01  digits 7;
type Adjustments_Type is delta 0.001 digits 6;

115

type COBOL_Employee_Record_Type is <-- External representation>
record
    Name      : COBOL.Alphanumeric(1..20);
    SSN       : COBOL.Alphanumeric(1..9);
    Salary    : COBOL.Byte_Array(1..4);
    Adjust    : COBOL.Numeric(1..7); <-- Sign and 6 digits>
end record;
pragma Convention (COBOL, COBOL_Employee_Record_Type);

116

package COBOL_Employee_IO is

```

```

        new COBOL_Sequential_IO(COBOL_Employee_Record_Type);
use COBOL_Employee_IO;
117

COBOL_File : File_Type;
118

type Ada_Employee_Record_Type is <-- Internal representation>
    record
        Name      : String(1..20);
        SSN       : String(1..9);
        Salary    : Salary_Type;
        Adjust    : Adjustments_Type;
    end record;
119

COBOL_Record : COBOL_Employee_Record_Type;
Ada_Record   : Ada_Employee_Record_Type;
120

package Salary_Conversions is
    new COBOL.Decimal_Conversions(Salary_Type);
use Salary_Conversions;
121

package Adjustments_Conversions is
    new COBOL.Decimal_Conversions(Adjustments_Type);
use Adjustments_Conversions;
122

begin
    Open (COBOL_File, Name => "Some_File");
123

    loop
        Read (COBOL_File, COBOL_Record);
124

        Ada_Record.Name := To_Ada(COBOL_Record.Name);
        Ada_Record.SSN  := To_Ada(COBOL_Record.SSN);
        Ada_Record.Salary :=
            To_Decimal(COBOL_Record.Salary, COBOL.High_Order_First);
        Ada_Record.Adjust :=

```



```

        To_Decimal(COBOL_Record.Adjust, COBOL.Leadings_Separate);
    ... <-- Process Ada_Record>
end loop;
exception
    when End_Error => ...
end Test_External_Formats;

```

16.5 B.5 Interfacing with Fortran

1

The facilities relevant to interfacing with the Fortran language are the package `Interfaces.Fortran` and support for the `Import`, `Export` and `Convention` pragmas with `<convention>_identifier Fortran`.

2

The package `Interfaces.Fortran` defines Ada types whose representations are identical to the default representations of the Fortran intrinsic types `Integer`, `Real`, `Double Precision`, `Complex`, `Logical`, and `Character` in a supported Fortran implementation. These Ada types can therefore be used to pass objects between Ada and Fortran programs.

Static Semantics

3

The library package `Interfaces.Fortran` has the following declaration:

4

```

with Ada.Numerics.Generic_Complex_Types; <-- see Section 21.1.1 [G.1.1],
page 1084>
pragma Elaborate_All(Ada.Numerics.Generic_Complex_Types);
package Interfaces.Fortran is
    pragma Pure(Fortran);

```

5

```

    type Fortran_Integer is range <implementation-defined>;

```

6

```

    type Real                is digits <implementation-defined>;
    type Double_Precision is digits <implementation-defined>;

```

7

```

    type Logical is new Boolean;

```

8

```

    package Single_Precision_Complex_Types is
        new Ada.Numerics.Generic_Complex_Types (Real);

```

9

```

    type Complex is new Single_Precision_Complex_Types.Complex;

```

10

```
subtype Imaginary is Single_Precision_Complex_Types.Imaginary;  
i : Imaginary renames Single_Precision_Complex_Types.i;  
j : Imaginary renames Single_Precision_Complex_Types.j;
```

11

```
type Character_Set is <implementation-defined character type>;
```

12

```
type Fortran_Character is array (Positive range <>) of Character_Set;█  
pragma Pack (Fortran_Character);
```

13

```
function To_Fortran (Item : in Character) return Character_Set;  
function To_Ada (Item : in Character_Set) return Character;
```

14

```
function To_Fortran (Item : in String) return Fortran_Character;  
function To_Ada (Item : in Fortran_Character) return String;
```

15

```
procedure To_Fortran (Item      : in String;  
                    Target    : out Fortran_Character;  
                    Last      : out Natural);
```

16

```
procedure To_Ada (Item      : in Fortran_Character;  
                Target    : out String;  
                Last      : out Natural);
```

17

```
end Interfaces.Fortran;
```

18

The types Fortran_Integer, Real, Double_Precision, Logical, Complex, and Fortran_Character are Fortran-compatible.

19

The To_Fortran and To_Ada functions map between the Ada type Character and the Fortran type Character_Set, and also between the Ada type String and the Fortran type Fortran_Character. The To_Fortran and To_Ada procedures have analogous effects to the string conversion subprograms found in Interfaces.COBOL.

Implementation Requirements

20

An implementation shall support pragma Convention with a Fortran <convention>_identifier for a Fortran-eligible type (see Section 16.1 [B.1], page 894).

Implementation Permissions

21

An implementation may add additional declarations to the Fortran interface packages. For example, the Fortran interface package for an implementation of Fortran 77 (ANSI X3.9-1978) that defines types like Integer*<n>, Real*<n>, Logical*<n>, and Complex*<n> may contain the declarations of types named Integer_Star_<n>, Real_Star_<n>, Logical_Star_<n>, and Complex_Star_<n>. (This convention should not apply to Character*<n>, for which the Ada analog is the constrained array subtype Fortran_Character (1..<n>).) Similarly, the Fortran interface package for an implementation of Fortran 90 that provides multiple <kinds> of intrinsic types, e.g. Integer (Kind=<n>), Real (Kind=<n>), Logical (Kind=<n>), Complex (Kind=<n>), and Character (Kind=<n>), may contain the declarations of types with the recommended names Integer_Kind_<n>, Real_Kind_<n>, Logical_Kind_<n>, Complex_Kind_<n>, and Character_Kind_<n>.

Implementation Advice

22

An Ada implementation should support the following interface correspondences between Ada and Fortran:

23

- An Ada procedure corresponds to a Fortran subroutine.

24

- An Ada function corresponds to a Fortran function.

25

- An Ada parameter of an elementary, array, or record type T is passed as a TF argument to a Fortran procedure, where TF is the Fortran type corresponding to the Ada type T, and where the INTENT attribute of the corresponding dummy argument matches the Ada formal parameter mode; the Fortran implementation's parameter passing conventions are used. For elementary types, a local copy is used if necessary to ensure by-copy semantics.

26

- An Ada parameter of an access-to-subprogram type is passed as a reference to a Fortran procedure whose interface corresponds to the designated subprogram's specification.

NOTES

27

16 An object of a Fortran-compatible record type, declared in a library package or subprogram, can correspond to a Fortran common block; the type also corresponds to a Fortran "derived type".

Examples

28

<Example of Interfaces.Fortran:>

29

```
with Interfaces.Fortran;
use Interfaces.Fortran;
procedure Ada_Application is
```

30

```
    type Fortran_Matrix is array (Integer range <>,
                                   Integer range <>) of Double_Precision;
    pragma Convention (Fortran, Fortran_Matrix);    <-- stored in Fortran's>
                                                    <-- column-major order>
    procedure Invert (Rank : in Fortran_Integer; X : in out Fortran_Matrix);
    pragma Import (Fortran, Invert);                <-- a Fortran subroutine>
```

31

```
    Rank      : constant Fortran_Integer := 100;
    My_Matrix : Fortran_Matrix (1 .. Rank, 1 .. Rank);
```

32

```
begin
```

33

```
    ...
    My_Matrix := ...;
    ...
    Invert (Rank, My_Matrix);
    ...
```

34

```
end Ada_Application;
```

17 Annex C Systems Programming

1

The Systems Programming Annex specifies additional capabilities provided for low-level programming. These capabilities are also required in many real-time, embedded, distributed, and information systems.

17.1 C.1 Access to Machine Operations

1

This clause specifies rules regarding access to machine instructions from within an Ada program.

Implementation Requirements

2

The implementation shall support machine code insertions (see Section 14.8 [13.8], page 518) or intrinsic subprograms (see Section 7.3.1 [6.3.1], page 263) (or both). Implementation-defined attributes shall be provided to allow the use of Ada entities as operands.

Implementation Advice

3

The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment, including privileged instructions, if any.

4

The interfacing pragmas (see Chapter 16 [Annex B], page 894) should support interface to assembler; the default assembler should be associated with the convention identifier Assembler.

5

If an entity is exported to assembly language, then the implementation should allocate it at an addressable location, and should ensure that it is retained by the linking process, even if not otherwise referenced from the Ada code. The implementation should assume that any call to a machine code or assembler subprogram is allowed to read or update every object that is specified as exported.

Documentation Requirements

6

The implementation shall document the overhead associated with calling machine-code or intrinsic subprograms, as compared to a fully-inlined call, and to a regular out-of-line call.

7

The implementation shall document the types of the package `System.Machine_Code` usable for machine code insertions, and the attributes to be used in machine code insertions for references to Ada entities.

8

The implementation shall document the subprogram calling conventions associated with the convention identifiers available for use with the interfacing pragmas (Ada and Assembler,

at a minimum), including register saving, exception propagation, parameter passing, and function value returning.

9

For exported and imported subprograms, the implementation shall document the mapping between the `Link_Name` string, if specified, or the Ada designator, if not, and the external link name used for such a subprogram.

Implementation Advice

10

The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms.

11

It is recommended that intrinsic subprograms be provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs. Examples of such instructions include:

12

- Atomic read–modify–write operations — e.g., test and set, compare and swap, decrement and test, enqueue/dequeue.

13

- Standard numeric functions — e.g., `<sin>`, `<log>`.

14

- String manipulation operations — e.g., translate and test.

15

- Vector operations — e.g., compare vector against thresholds.

16

- Direct operations on I/O ports.

17.2 C.2 Required Representation Support

1/2

This clause specifies minimal requirements on the support for representation items and related features.

Implementation Requirements

2

The implementation shall support at least the functionality defined by the recommended levels of support in Section 13.

17.3 C.3 Interrupt Support

1

This clause specifies the language–defined model for hardware interrupts in addition to mechanisms for handling interrupts.

Dynamic Semantics

2

An <interrupt> represents a class of events that are detected by the hardware or the system software. Interrupts are said to occur. An <occurrence> of an interrupt is separable into generation and delivery. <Generation> of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program. <Delivery> is the action that invokes part of the program as response to the interrupt occurrence. Between generation and delivery, the interrupt occurrence (or interrupt) is <pending>. Some or all interrupts may be <blocked>. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Certain interrupts are <reserved>. The set of reserved interrupts is implementation defined. A reserved interrupt is either an interrupt for which user–defined handlers are not supported, or one which already has an attached handler by some other implementation–defined means. Program units can be connected to non–reserved interrupts. While connected, the program unit is said to be <attached> to that interrupt. The execution of that program unit, the <interrupt handler>, is invoked upon delivery of the interrupt occurrence.

3

While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt. While the handler executes, the corresponding interrupt is blocked.

4

While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Whether such occurrences remain pending or are lost is implementation defined.

5

Each interrupt has a <default treatment> which determines the system’s response to an occurrence of that interrupt when no user–defined handler is attached. The set of possible default treatments is implementation defined, as is the method (if one exists) for configuring the default treatments for interrupts.

6

An interrupt is delivered to the handler (or default treatment) that is in effect for that interrupt at the time of delivery.

7

An exception propagated from a handler that is invoked by an interrupt has no effect.

8

If the Ceiling_Locking policy (see Section 18.3 [D.3], page 991) is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object.

Implementation Requirements

9

The implementation shall provide a mechanism to determine the minimum stack space that is needed for each interrupt handler and to reserve that space for the execution of

the handler. This space should accommodate nested invocations of the handler where the system permits this.

10

If the hardware or the underlying system holds pending interrupt occurrences, the implementation shall provide for later delivery of these occurrences to the program.

11

If the Ceiling.Locking policy is not in effect, the implementation shall provide means for the application to specify whether interrupts are to be blocked during protected actions.

Documentation Requirements

12

The implementation shall document the following items:

13

1. For each interrupt, which interrupts are blocked from delivery when a handler attached to that interrupt executes (either as a result of an interrupt delivery or of an ordinary call on a procedure of the corresponding protected object).

14

2. Any interrupts that cannot be blocked, and the effect of attaching handlers to such interrupts, if this is permitted.

15

3. Which run-time stack an interrupt handler uses when it executes as a result of an interrupt delivery; if this is configurable, what is the mechanism to do so; how to specify how much space to reserve on that stack.

16

4. Any implementation- or hardware-specific activity that happens before a user-defined interrupt handler gets control (e.g., reading device registers, acknowledging devices).

17

5. Any timing or other limitations imposed on the execution of interrupt handlers.

18

6. The state (blocked/unblocked) of the non-reserved interrupts when the program starts; if some interrupts are unblocked, what is the mechanism a program can use to protect itself before it can attach the corresponding handlers.

19

7. Whether the interrupted task is allowed to resume execution before the interrupt handler returns.

20

8. The treatment of interrupt occurrences that are generated while the interrupt is blocked; i.e., whether one or more occurrences are held for later delivery, or all are lost.

21

9. Whether predefined or implementation–defined exceptions are raised as a result of the occurrence of any interrupt, and the mapping between the machine interrupts (or traps) and the predefined exceptions.

22

10. On a multi–processor, the rules governing the delivery of an interrupt to a particular processor.

Implementation Permissions

23/2

If the underlying system or hardware does not allow interrupts to be blocked, then no blocking is required as part of the execution of subprograms of a protected object for which one of its subprograms is an interrupt handler.

24

In a multi–processor with more than one interrupt subsystem, it is implementation defined whether (and how) interrupt sources from separate subsystems share the same `Interrupt_ID` type (see Section 17.3.2 [C.3.2], page 957). In particular, the meaning of a blocked or pending interrupt may then be applicable to one processor only.

25

Implementations are allowed to impose timing or other limitations on the execution of interrupt handlers.

26/2

Other forms of handlers are allowed to be supported, in which case the rules of this clause should be adhered to.

27

The active priority of the execution of an interrupt handler is allowed to vary from one occurrence of the same interrupt to another.

Implementation Advice

28/2

If the `Ceiling.Locking` policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for finer–grained control of interrupt blocking.

NOTES

29

1 The default treatment for an interrupt can be to keep the interrupt pending or to deliver it to an implementation–defined handler. Examples of actions that an implementation–defined handler is allowed

to perform include aborting the partition, ignoring (i.e., discarding occurrences of) the interrupt, or queuing one or more occurrences of the interrupt for possible later delivery when a user-defined handler is attached to that interrupt.

30

2 It is a bounded error to call `Task_Identification.Current_Task` (see Section 17.7.1 [C.7.1], page 965) from an interrupt handler.

31

3 The rule that an exception propagated from an interrupt handler has no effect is modeled after the rule about exceptions propagated out of task bodies.

17.3.1 C.3.1 Protected Procedure Handlers

Syntax

1

The form of a pragma `Interrupt_Handler` is as follows:

2

```
pragma Interrupt_Handler(<handler_>name);
```

3

The form of a pragma `Attach_Handler` is as follows:

4

```
pragma Attach_Handler(<handler_>name, expression);
```

Name Resolution Rules

5

For the `Interrupt_Handler` and `Attach_Handler` pragmas, the `<handler_>name` shall resolve to denote a protected procedure with a parameterless profile.

6

For the `Attach_Handler` pragma, the expected type for the expression is `Interrupts.Interrupt_ID` (see Section 17.3.2 [C.3.2], page 957).

Legality Rules

7/2

The `Attach_Handler` pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration (see [S0193], page 338) or single_protected_declaration (see [S0194], page 338) shall be a library-level declaration.

8/2

The `Interrupt_Handler` pragma is only allowed immediately within the protected_definition

where the corresponding subprogram is declared. The corresponding `protected_-type_declaration` (see [S0193], page 338) or `single_protected_declaration` (see [S0194], page 338) shall be a library-level declaration.

Dynamic Semantics

9

If the pragma `Interrupt_Handler` appears in a `protected_definition`, then the corresponding procedure can be attached dynamically, as a handler, to interrupts (see Section 17.3.2 [C.3.2], page 957). Such procedures are allowed to be attached to multiple interrupts.

10

The expression in the `Attach_Handler` pragma as evaluated at object creation time specifies an interrupt. As part of the initialization of that object, if the `Attach_Handler` pragma is specified, the `<handler>` procedure is attached to the specified interrupt. A check is made that the corresponding interrupt is not reserved. `Program_Error` is raised if the check fails, and the existing treatment for the interrupt is not affected.

11/2

If the `Ceiling_Locking` policy (see Section 18.3 [D.3], page 991) is in effect, then upon the initialization of a protected object for which either an `Attach_Handler` or `Interrupt_Handler` pragma applies to one of its procedures, a check is made that the ceiling priority defined in the `protected_definition` is in the range of `System.Interrupt_Priority`. If the check fails, `Program_Error` is raised.

12/1

When a protected object is finalized, for any of its procedures that are attached to interrupts, the handler is detached. If the handler was attached by a procedure in the `Interrupts` package or if no user handler was previously attached to the interrupt, the default treatment is restored. If an `Attach_Handler` pragma was used and the most recently attached handler for the same interrupt is the same as the one that was attached at the time the protected object was initialized, the previous handler is restored.

13

When a handler is attached to an interrupt, the interrupt is blocked (subject to the Implementation Permission in Section 17.3 [C.3], page 951) during the execution of every protected action on the protected object containing the handler.

Erroneous Execution

14

If the `Ceiling_Locking` policy (see Section 18.3 [D.3], page 991) is in effect and an interrupt is delivered to a handler, and the interrupt hardware priority is higher than the ceiling priority of the corresponding protected object, the execution of the program is erroneous.

14.1/1

If the handlers for a given interrupt attached via pragma `Attach_Handler` are not attached and detached in a stack-like (LIFO) order, program execution is erroneous. In particular, when a protected object is finalized, the execution is erroneous if any of the procedures of the protected object are attached to interrupts via pragma `Attach_Handler` and the most recently attached handler for the same interrupt is not the same as the one that was attached at the time the protected object was initialized.

Metrics

15

The following metric shall be documented by the implementation:

16/2

- The worst–case overhead for an interrupt handler that is a parameterless protected procedure, in clock cycles. This is the execution time not directly attributable to the handler procedure or the interrupted execution. It is estimated as $C - (A+B)$, where A is how long it takes to complete a given sequence of instructions without any interrupt, B is how long it takes to complete a normal call to a given protected procedure, and C is how long it takes to complete the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt.

Implementation Permissions

17

When the pragmas `Attach_Handler` or `Interrupt_Handler` apply to a protected procedure, the implementation is allowed to impose implementation–defined restrictions on the corresponding `protected_type_declaration` (see [S0193], page 338) and `protected_body` (see [S0198], page 338).

18

An implementation may use a different mechanism for invoking a protected procedure in response to a hardware interrupt than is used for a call to that protected procedure from a task.

19

Notwithstanding what this subclause says elsewhere, the `Attach_Handler` and `Interrupt_Handler` pragmas are allowed to be used for other, implementation defined, forms of interrupt handlers.

Implementation Advice

20

Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware.

21

Whenever practical, the implementation should detect violations of any implementation–defined restrictions before run time.

NOTES

22

4 The `Attach_Handler` pragma can provide static attachment of handlers to interrupts if the implementation supports preelaboration of protected objects. (See Section 17.4 [C.4], page 960.)

23/2

5 A protected object that has a (protected) procedure attached to an interrupt should have a ceiling priority at least as high as the highest processor priority at which that interrupt will ever be delivered.

24

6 Protected procedures can also be attached dynamically to interrupts via operations declared in the predefined package Interrupts.

25

7 An example of a possible implementation-defined restriction is disallowing the use of the standard storage pools within the body of a protected procedure that is an interrupt handler.

17.3.2 C.3.2 The Package Interrupts

Static Semantics

1

The following language-defined packages exist:

2

```
with System;
package Ada.Interrupts is
  type Interrupt_ID is <implementation-defined>;
  type Parameterless_Handler is
    access protected procedure;
```

3/1

<This paragraph was deleted.>

4

```
function Is_Reserved (Interrupt : Interrupt_ID)
  return Boolean;
```

5

```
function Is_Attached (Interrupt : Interrupt_ID)
  return Boolean;
```

6

```
function Current_Handler (Interrupt : Interrupt_ID)
  return Parameterless_Handler;
```

7

```
procedure Attach_Handler
  (New_Handler : in Parameterless_Handler;
   Interrupt    : in Interrupt_ID);
```

8

```
procedure Exchange_Handler
```

```

(Old_Handler : out Parameterless_Handler;
 New_Handler : in Parameterless_Handler;
 Interrupt    : in Interrupt_ID);

9

procedure Detach_Handler
  (Interrupt : in Interrupt_ID);

10

function Reference(Interrupt : Interrupt_ID)
  return System.Address;

11

private
  ... -- <not specified by the language>
end Ada.Interrupts;

12

package Ada.Interrupts.Names is
  <implementation-defined> : constant Interrupt_ID :=
    <implementation-defined>;
    . . .
  <implementation-defined> : constant Interrupt_ID :=
    <implementation-defined>;
end Ada.Interrupts.Names;

```

Dynamic Semantics

13

The `Interrupt_ID` type is an `implementation-defined` discrete type used to identify interrupts.

14

The `Is_Reserved` function returns `True` if and only if the specified interrupt is reserved.

15

The `Is_Attached` function returns `True` if and only if a user-specified interrupt handler is attached to the interrupt.

16/1

The `Current_Handler` function returns a value that represents the attached handler of the interrupt. If no user-defined handler is attached to the interrupt, `Current_Handler` returns null.

17

The `Attach_Handler` procedure attaches the specified handler to the interrupt, overriding any existing treatment (including a user handler) in effect for that interrupt. If `New_Handler` is null, the default treatment is restored. If `New_Handler` designates a protected procedure to which the pragma `Interrupt_Handler` does not apply, `Program_Error` is raised. In this case, the operation does not modify the existing interrupt treatment.

18/1

The `Exchange_Handler` procedure operates in the same manner as `Attach_Handler` with the addition that the value returned in `Old_Handler` designates the previous treatment for the specified interrupt. If the previous treatment is not a user-defined handler, null is returned.

19

The `Detach_Handler` procedure restores the default treatment for the specified interrupt.

20

For all operations defined in this package that take a parameter of type `Interrupt_ID`, with the exception of `Is_Reserved` and `Reference`, a check is made that the specified interrupt is not reserved. `Program_Error` is raised if this check fails.

21

If, by using the `Attach_Handler`, `Detach_Handler`, or `Exchange_Handler` procedures, an attempt is made to detach a handler that was attached statically (using the pragma `Attach_Handler`), the handler is not detached and `Program_Error` is raised.

22/2

The `Reference` function returns a value of type `System.Address` that can be used to attach a task entry via an address clause (see Section 23.7.1 [J.7.1], page 1171) to the interrupt specified by `Interrupt`. This function raises `Program_Error` if attaching task entries to interrupts (or to this particular interrupt) is not supported.

Implementation Requirements

23

At no time during attachment or exchange of handlers shall the current handler of the corresponding interrupt be undefined.

Documentation Requirements

24/2

If the `Ceiling-Locking` policy (see Section 18.3 [D.3], page 991) is in effect, the implementation shall document the default ceiling priority assigned to a protected object that contains either the `Attach_Handler` or `Interrupt_Handler` pragmas, but not the `Interrupt_Priority` pragma. This default need not be the same for all interrupts.

Implementation Advice

25

If implementation-defined forms of interrupt handler procedures are supported, such as protected procedures with parameters, then for each such form of a handler, a type analogous to `Parameterless_Handler` should be specified in a child package of `Interrupts`, with the same operations as in the predefined package `Interrupts`.

NOTES

26

8 The package `Interrupts.Names` contains implementation-defined names (and constant values) for the interrupts that are supported by the implementation.

Examples

27

<Example of interrupt handlers:>

```

Device_Priority : constant
  array (1..5) of System.Interrupt_Priority := ( ... );
protected type Device_Interface
  (Int_ID : Ada.Interrupts.Interrupt_ID) is
  procedure Handler;
  pragma Attach_Handler(Handler, Int_ID);
  ...
  pragma Interrupt_Priority(Device_Priority(Int_ID));
end Device_Interface;
...
Device_1_Driver : Device_Interface(1);
...
Device_5_Driver : Device_Interface(5);
...

```

17.4 C.4 Preelaboration Requirements

1

This clause specifies additional implementation and documentation requirements for the Preelaborate pragma (see Section 11.2.1 [10.2.1], page 413).

Implementation Requirements

2

The implementation shall not incur any run–time overhead for the elaboration checks of subprograms and protected_bodies declared in preelaborated library units.

3

The implementation shall not execute any memory write operations after load time for the elaboration of constant objects declared immediately within the declarative region of a preelaborated library package, so long as the subtype and initial expression (or default initial expressions if initialized by default) of the object_declaration satisfy the following restrictions. The meaning of <load time> is implementation defined.

4

- Any subtype_mark denotes a statically constrained subtype, with statically constrained subcomponents, if any;

4.1/2

- no subtype_mark denotes a controlled type, a private type, a private extension, a generic formal private type, a generic formal derived type, or a descendant of such a type;

5

- any constraint is a static constraint;

6

- any allocator is for an access-to-constant type;

7

- any uses of predefined operators appear only within static expressions;

8

- any primaries that are names, other than attribute_references for the Access or Address attributes, appear only within static expressions;

9

- any name that is not part of a static expression is an expanded name or direct_name that statically denotes some entity;

10

- any discrete_choice of an array_aggregate is static;

11

- no language-defined check associated with the elaboration of the object_declaration can fail.

Documentation Requirements

12

The implementation shall document any circumstances under which the elaboration of a preelaborated package causes code to be executed at run time.

13

The implementation shall document whether the method used for initialization of preelaborated variables allows a partition to be restarted without reloading.

Implementation Advice

14

It is recommended that preelaborated packages be implemented in such a way that there should be little or no code executed at run time for the elaboration of entities not already covered by the Implementation Requirements.

17.5 C.5 Pragma Discard_Names

1

A pragma Discard_Names may be used to request a reduction in storage used for the names of certain entities.

Syntax

2

The form of a pragma `Discard_Names` is as follows:

3

```
pragma Discard_Names[([On => ] local_name)];
```

4

A pragma `Discard_Names` is allowed only immediately within a `declarative_part`, immediately within a `package_specification`, or as a configuration pragma.

Legality Rules

5

The `local_name` (if present) shall denote a non-derived enumeration first subtype, a tagged first subtype, or an exception. The pragma applies to the type or exception. Without a `local_name`, the pragma applies to all such entities declared after the pragma, within the same declarative region. Alternatively, the pragma can be used as a configuration pragma. If the pragma applies to a type, then it applies also to all descendants of the type.

Static Semantics

6

If a `local_name` is given, then a pragma `Discard_Names` is a representation pragma.

7/2

If the pragma applies to an enumeration type, then the semantics of the `Wide_Wide_Image` and `Wide_Wide_Value` attributes are implementation defined for that type; the semantics of `Image`, `Wide_Image`, `Value`, and `Wide_Value` are still defined in terms of `Wide_Wide_Image` and `Wide_Wide_Value`. In addition, the semantics of `Text_IO.Enumeration_IO` are implementation defined. If the pragma applies to a tagged type, then the semantics of the `Tags.Wide_Wide_Expanded_Name` function are implementation defined for that type; the semantics of `Tags.Expanded_Name` and `Tags.Wide_Expanded_Name` are still defined in terms of `Tags.Wide_Wide_Expanded_Name`. If the pragma applies to an exception, then the semantics of the `Exceptions.Wide_Wide_Exception_Name` function are implementation defined for that exception; the semantics of `Exceptions.Exception_Name` and `Exceptions.Wide_Exception_Name` are still defined in terms of `Exceptions.Wide_Wide_Exception_Name`.

Implementation Advice

8

If the pragma applies to an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity.

17.6 C.6 Shared Variable Control

1

This clause specifies representation pragmas that control the use of shared variables.

Syntax

2

The form for pragmas `Atomic`, `Volatile`, `Atomic_Components`, and `Volatile_Components` is as follows:

3

```
pragma Atomic(local_name);
```

4

```
pragma Volatile(local_name);
```

5

```
pragma Atomic_Components(<array_>local_name);
```

6

```
pragma Volatile_Components(<array_>local_name);
```

7/2

An `<atomic>` type is one to which a `pragma Atomic` applies. An `<atomic>` object (including a component) is one to which a `pragma Atomic` applies, or a component of an array to which a `pragma Atomic_Components` applies, or any object of an atomic type, other than objects obtained by evaluating a slice.

8

A `<volatile>` type is one to which a `pragma Volatile` applies. A `<volatile>` object (including a component) is one to which a `pragma Volatile` applies, or a component of an array to which a `pragma Volatile_Components` applies, or any object of a volatile type. In addition, every atomic type or object is also defined to be volatile. Finally, if an object is volatile, then so are all of its subcomponents (the same does not apply to atomic).

Name Resolution Rules

9

The `local_name` in an `Atomic` or `Volatile` pragma shall resolve to denote either an `object_declaration`, a `non-inherited component_declaration` (see [S0070], page 130), or a `full_type_declaration` (see [S0024], page 53). The `<array_>local_name` in an `Atomic_Components` or `Volatile_Components` pragma shall resolve to denote the declaration of an array type or an array object of an anonymous type.

Legality Rules

10

It is illegal to apply either an `Atomic` or `Atomic_Components` pragma to an object or type if the implementation cannot support the indivisible reads and updates required by the pragma (see below).

11

It is illegal to specify the `Size` attribute of an atomic object, the `Component_Size` attribute for an array type with atomic components, or the layout attributes of an atomic component, in a way that prevents the implementation from performing the required indivisible reads and updates.

12

If an atomic object is passed as a parameter, then the type of the formal parameter shall either be atomic or allow pass by copy (that is, not be a nonatomic by-reference type). If an atomic object is used as an actual for a generic formal object of mode in out, then the

type of the generic formal object shall be atomic. If the prefix of an attribute_reference for an Access attribute denotes an atomic object (including a component), then the designated type of the resulting access type shall be atomic. If an atomic type is used as an actual for a generic formal derived type, then the ancestor of the formal type shall be atomic or allow pass by copy. Corresponding rules apply to volatile objects and types.

13

If a pragma Volatile, Volatile_Components, Atomic, or Atomic_Components applies to a stand-alone constant object, then a pragma Import shall also apply to it.

Static Semantics

14

These pragmas are representation pragmas (see Section 14.1 [13.1], page 481).

Dynamic Semantics

15

For an atomic object (including an atomic component) all reads and updates of the object as a whole are indivisible.

16

For a volatile object all reads and updates of the object as a whole are performed directly to memory.

17

Two actions are sequential (see Section 10.10 [9.10], page 389) if each is the read or update of the same atomic object.

18

If a type is atomic or volatile and it is not a by-copy type, then the type is defined to be a by-reference type. If any subcomponent of a type is atomic or volatile, then the type is defined to be a by-reference type.

19

If an actual parameter is atomic or volatile, and the corresponding formal parameter is not, then the parameter is passed by copy.

Implementation Requirements

20

The external effect of a program (see Section 2.1.3 [1.1.3], page 23) is defined to include each read and update of a volatile or atomic object. The implementation shall not generate any memory reads or updates of atomic or volatile objects other than those specified by the program.

21

If a pragma Pack applies to a type any of whose subcomponents are atomic, the implementation shall not pack the atomic subcomponents more tightly than that for which it can support indivisible reads and updates.

Implementation Advice

22/2

A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others.

23/2

A load or store of an atomic object should, where possible, be implemented by a single load or store instruction.

NOTES

24

9 An imported volatile or atomic constant behaves as a constant (i.e. read-only) with respect to other parts of the Ada program, but can still be modified by an "external source."

17.7 C.7 Task Information

1/2

This clause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined. Finally, a package that associates termination procedures with a task or set of tasks is defined.

17.7.1 C.7.1 The Package `Task_Identification`

Static Semantics

1

The following language-defined library package exists:

2/2

```
package Ada.Task_Identification is
  pragma Preelaborate(Task_Identification);
  type Task_Id is private;
  pragma Preelaborable_Initialization (Task_Id);
  Null_Task_Id : constant Task_Id;
  function "=" (Left, Right : Task_Id) return Boolean;
```

3/1

```
  function Image      (T : Task_Id) return String;
  function Current_Task return Task_Id;
  procedure Abort_Task (T : in Task_Id);
```

4

```
  function Is_Terminated(T : Task_Id) return Boolean;
  function Is_Callable (T : Task_Id) return Boolean;
private
  ... -- <not specified by the language>
end Ada.Task_Identification;
```

Dynamic Semantics

5

A value of the type `Task_Id` identifies an existent task. The constant `Null_Task_Id` does

not identify any task. Each object of the type `Task_Id` is default initialized to the value of `Null_Task_Id`.

6

The function `"=` returns `True` if and only if `Left` and `Right` identify the same task or both have the value `Null_Task_Id`.

7

The function `Image` returns an implementation-defined string that identifies `T`. If `T` equals `Null_Task_Id`, `Image` returns an empty string.

8

The function `Current_Task` returns a value that identifies the calling task.

9

The effect of `Abort_Task` is the same as the `abort_statement` for the task identified by `T`. In addition, if `T` identifies the environment task, the entire partition is aborted, See Section 19.1 [E.1], page 1034.

10

The functions `Is_Terminated` and `Is_Callable` return the value of the corresponding attribute of the task identified by `T`.

11

For a prefix `T` that is of a task type (after any implicit dereference), the following attribute is defined:

12

`T'Identity`

Yields a value of the type `Task_Id` that identifies the task denoted by `T`.

13

For a prefix `E` that denotes an `entry_declaration`, the following attribute is defined:

14

`E'Caller`

Yields a value of the type `Task_Id` that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an `entry_body` or `accept_statement` corresponding to the `entry_declaration` denoted by `E`.

15

`Program_Error` is raised if a value of `Null_Task_Id` is passed as a parameter to `Abort_Task`, `Is_Terminated`, and `Is_Callable`.

16

Abort_Task is a potentially blocking operation (see Section 10.5.1 [9.5.1], page 344).

Bounded (Run-Time) Errors

17/2

It is a bounded error to call the Current_Task function from an entry body, interrupt handler, or finalization of a task attribute. Program_Error is raised, or an implementation-defined value of the type Task_Id is returned.

Erroneous Execution

18

If a value of Task_Id is passed as a parameter to any of the operations declared in this package (or any language-defined child of this package), and the corresponding task object no longer exists, the execution of the program is erroneous.

Documentation Requirements

19

The implementation shall document the effect of calling Current_Task from an entry body or interrupt handler.

NOTES

20

10 This package is intended for use in writing user-defined task scheduling packages and constructing server tasks. Current_Task can be used in conjunction with other operations requiring a task as an argument such as Set_Priority (see Section 18.5 [D.5], page 996).

21

11 The function Current_Task and the attribute Caller can return a Task_Id value that identifies the environment task.

17.7.2 C.7.2 The Package Task_Attributes

Static Semantics

1

The following language-defined generic library package exists:

2

```
with Ada.Task_Identification; use Ada.Task_Identification;
generic
  type Attribute is private;
  Initial_Value : in Attribute;
package Ada.Task_Attributes is
```

3

```
  type Attribute_Handle is access all Attribute;
```

4

```
function Value(T : Task_Id := Current_Task)
  return Attribute;
```

5

```
function Reference(T : Task_Id := Current_Task)
  return Attribute_Handle;
```

6

```
procedure Set_Value(Val : in Attribute;
                   T : in Task_Id := Current_Task);
procedure Reinitialize(T : in Task_Id := Current_Task);
```

7

```
end Ada.Task_Attributes;
```

Dynamic Semantics

8

When an instance of `Task_Attributes` is elaborated in a given active partition, an object of the actual type corresponding to the formal type `Attribute` is implicitly created for each task (of that partition) that exists and is not yet terminated. This object acts as a user-defined attribute of the task. A task created previously in the partition and not yet terminated has this attribute from that point on. Each task subsequently created in the partition will have this attribute when created. In all these cases, the initial value of the given attribute is `Initial_Value`.

9

The `Value` operation returns the value of the corresponding attribute of `T`.

10

The `Reference` operation returns an access value that designates the corresponding attribute of `T`.

11

The `Set_Value` operation performs any finalization on the old value of the attribute of `T` and assigns `Val` to that attribute (see Section 6.2 [5.2], page 242, and Section 8.6 [7.6], page 295).

12

The effect of the `Reinitialize` operation is the same as `Set_Value` where the `Val` parameter is replaced with `Initial_Value`.

13

For all the operations declared in this package, `Tasking_Error` is raised if the task identified by `T` is terminated. `Program_Error` is raised if the value of `T` is `Null_Task_Id`.

13.1/2

After a task has terminated, all of its attributes are finalized, unless they have been finalized earlier. When the master of an instantiation of `Ada.Task_Attributes` is finalized, the corresponding attribute of each task is finalized, unless it has been finalized earlier.

Bounded (Run-Time) Errors

13.2/1

If the package `Ada.Task_Attributes` is instantiated with a controlled type and the controlled type has user-defined `Adjust` or `Finalize` operations that in turn access task attributes by any of the above operations, then a call of `Set_Value` of the instantiated package constitutes a bounded error. The call may perform as expected or may result in forever blocking the calling task and subsequently some or all tasks of the partition.

Erroneous Execution

14

It is erroneous to dereference the access value returned by a given call on `Reference` after a subsequent call on `Reinitialize` for the same task attribute, or after the associated task terminates.

15

If a value of `Task_Id` is passed as a parameter to any of the operations declared in this package and the corresponding task object no longer exists, the execution of the program is erroneous.

15.1/2

An access to a task attribute via a value of type `Attribute_Handle` is erroneous if executed concurrently with another such access or a call of any of the operations declared in package `Task_Attributes`. An access to a task attribute is erroneous if executed concurrently with or after the finalization of the task attribute.

Implementation Requirements

16/1

For a given attribute of a given task, the implementation shall perform the operations declared in this package atomically with respect to any of these operations of the same attribute of the same task. The granularity of any locking mechanism necessary to achieve such atomicity is implementation defined.

17/2

After task attributes are finalized, the implementation shall reclaim any storage associated with the attributes.

Documentation Requirements

18

The implementation shall document the limit on the number of attributes per task, if any, and the limit on the total storage for attribute values per task, if such a limit exists.

19

In addition, if these limits can be configured, the implementation shall document how to configure them.

Metrics

20/2

The implementation shall document the following metrics: A task calling the following subprograms shall execute at a sufficiently high priority as to not be preempted during the measurement period. This period shall start just before issuing the call and end just after the call completes. If the attributes of task `T` are accessed by the measurement tests, no other task shall access attributes of that task during the measurement period. For all measurements described here, the `Attribute` type shall be a scalar type whose size is

equal to the size of the predefined type Integer. For each measurement, two cases shall be documented: one where the accessed attributes are of the calling task (that is, the default value for the T parameter is used), and the other, where T identifies another, non-terminated, task.

21

The following calls (to subprograms in the Task_Attributes package) shall be measured:

22

- a call to Value, where the return value is Initial_Value;

23

- a call to Value, where the return value is not equal to Initial_Value;

24

- a call to Reference, where the return value designates a value equal to Initial_Value;

25

- a call to Reference, where the return value designates a value not equal to Initial_Value;

26/2

- a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is equal to Initial_Value;

27

- a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is not equal to Initial_Value.

Implementation Permissions

28

An implementation need not actually create the object corresponding to a task attribute until its value is set to something other than that of Initial_Value, or until Reference is called for the task attribute. Similarly, when the value of the attribute is to be reinitialized to that of Initial_Value, the object may instead be finalized and its storage reclaimed, to be recreated when needed later. While the object does not exist, the function Value may simply return Initial_Value, rather than implicitly creating the object.

29

An implementation is allowed to place restrictions on the maximum number of attributes a task may have, the maximum size of each attribute, and the total storage size allocated for all the attributes of a task.

Implementation Advice

30/2

Some implementations are targeted to domains in which memory use at run time must be completely deterministic. For such implementations, it is recommended that the storage

for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the attributes of a task, or by using the pre-allocated storage for the first N attribute objects, and the heap for the others. In the latter case, N should be documented.

30.1/2

Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination.

NOTES

31

12 An attribute always exists (after instantiation), and has the initial value. It need not occupy memory until the first operation that potentially changes the attribute value. The same holds true after Reinitialize.

32

13 The result of the Reference function should be used with care; it is always safe to use that result in the task body whose attribute is being accessed. However, when the result is being used by another task, the programmer must make sure that the task whose attribute is being accessed is not yet terminated. Failing to do so could make the program execution erroneous.

33/2

<This paragraph was deleted.>

17.7.3 C.7.3 The Package Task_Termination

Static Semantics

1/2

The following language-defined library package exists:

2/2

```
with Ada.Task_Identification;
with Ada.Exceptions;
package Ada.Task_Termination is
  pragma Preelaborate(Task_Termination);
```

3/2

```
  type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);■
```

4/2

```
  type Termination_Handler is access protected procedure
    (Cause : in Cause_Of_Termination;
```

```
T      : in Ada.Task_Identification.Task_Id;
X      : in Ada.Exceptions.Exception_Occurrence);
```

5/2

```
procedure Set_Dependents_Fallback_Handler
  (Handler: in Termination_Handler);
function Current_Task_Fallback_Handler return Termination_Handler;■
```

6/2

```
procedure Set_Specific_Handler
  (T      : in Ada.Task_Identification.Task_Id;
   Handler : in Termination_Handler);
function Specific_Handler (T : Ada.Task_Identification.Task_Id)
  return Termination_Handler;
```

7/2

```
end Ada.Task_Termination;
```

Dynamic Semantics

8/2

The type `Termination_Handler` identifies a protected procedure to be executed by the implementation when a task terminates. Such a protected procedure is called a `<handler>`. In all cases `T` identifies the task that is terminating. If the task terminates due to completing the last statement of its body, or as a result of waiting on a terminate alternative, then `Cause` is set to `Normal` and `X` is set to `Null_Occurrence`. If the task terminates because it is being aborted, then `Cause` is set to `Abnormal` and `X` is set to `Null_Occurrence`. If the task terminates because of an exception raised by the execution of its `task_body`, then `Cause` is set to `Unhandled_Exception` and `X` is set to the associated exception occurrence.

9/2

Each task has two termination handlers, a `<fall-back handler>` and a `<specific handler>`. The specific handler applies only to the task itself, while the fall-back handler applies only to the dependent tasks of the task. A handler is said to be `<set>` if it is associated with a non-null value of type `Termination_Handler`, and `<cleared>` otherwise. When a task is created, its specific handler and fall-back handler are cleared.

10/2

The procedure `Set_Dependents_Fallback_Handler` changes the fall-back handler for the calling task; if `Handler` is null, that fall-back handler is cleared, otherwise it is set to be `Handler.all`. If a fall-back handler had previously been set it is replaced.

11/2

The function `Current_Task_Fallback_Handler` returns the fall-back handler that is currently set for the calling task, if one is set; otherwise it returns null.

12/2

The procedure `Set_Specific_Handler` changes the specific handler for the task identified by `T`; if `Handler` is null, that specific handler is cleared, otherwise it is set to be `Handler.all`. If a specific handler had previously been set it is replaced.

13/2

The function `Specific_Handler` returns the specific handler that is currently set for the task identified by `T`, if one is set; otherwise it returns null.

14/2

As part of the finalization of a `task_body`, after performing the actions specified in Section 8.6 [7.6], page 295, for finalization of a master, the specific handler for the task, if one is set, is executed. If the specific handler is cleared, a search for a fall-back handler proceeds by recursively following the master relationship for the task. If a task is found whose fall-back handler is set, that handler is executed; otherwise, no handler is executed.

15/2

For `Set_Specific_Handler` or `Specific_Handler`, `Tasking_Error` is raised if the task identified by `T` has already terminated. `Program_Error` is raised if the value of `T` is `Ada.Task_Identification.Null_Task_Id`.

16/2

An exception propagated from a handler that is invoked as part of the termination of a task has no effect.

Erroneous Execution

17/2

For a call of `Set_Specific_Handler` or `Specific_Handler`, if the task identified by `T` no longer exists, the execution of the program is erroneous.

18 Annex D Real-Time Systems

1

This Annex specifies additional characteristics of Ada implementations intended for real-time systems software. To conform to this Annex, an implementation shall also conform to the Systems Programming Annex.

Metrics

2

The metrics are documentation requirements; an implementation shall document the values of the language-defined metrics for at least one configuration of hardware or an underlying system supported by the implementation, and shall document the details of that configuration.

3

The metrics do not necessarily yield a simple number. For some, a range is more suitable, for others a formula dependent on some parameter is appropriate, and for others, it may be more suitable to break the metric into several cases. Unless specified otherwise, the metrics in this annex are expressed in processor clock cycles. For metrics that require documentation of an upper bound, if there is no upper bound, the implementation shall report that the metric is unbounded.

NOTES

4

1 The specification of the metrics makes a distinction between upper bounds and simple execution times. Where something is just specified as "the execution time of" a piece of code, this leaves one the freedom to choose a nonpathological case. This kind of metric is of the form "there exists a program such that the value of the metric is V". Conversely, the meaning of upper bounds is "there is no program such that the value of the metric is greater than V". This kind of metric can only be partially tested, by finding the value of V for one or more test programs.

5

2 The metrics do not cover the whole language; they are limited to features that are specified in Chapter 17 [Annex C], page 949, "Chapter 17 [Annex C], page 949, Systems Programming" and in this Annex. The metrics are intended to provide guidance to potential users as to whether a particular implementation of such a feature is going to be adequate for a particular real-time application. As such, the metrics are aimed at known implementation choices that can result in significant performance differences.

6

3 The purpose of the metrics is not necessarily to provide fine-grained quantitative results or to serve as a comparison

between different implementations on the same or different platforms. Instead, their goal is rather qualitative; to define a standard set of approximate values that can be measured and used to estimate the general suitability of an implementation, or to evaluate the comparative utility of certain features of an implementation for a particular real–time application.

18.1 D.1 Task Priorities

1

This clause specifies the priority model for real–time systems. In addition, the methods for specifying priorities are defined.

Syntax

2

The form of a pragma `Priority` is as follows:

3

```
pragma Priority(expression);
```

4

The form of a pragma `Interrupt_Priority` is as follows:

5

```
pragma Interrupt_Priority[(expression)];  
Name Resolution Rules
```

6

The expected type for the expression in a `Priority` or `Interrupt_Priority` pragma is `Integer`.

Legality Rules

7

A `Priority` pragma is allowed only immediately within a `task_definition`, a `protected_definition`, or the `declarative_part` of a `subprogram_body`. An `Interrupt_Priority` pragma is allowed only immediately within a `task_definition` or a `protected_definition`. At most one such pragma shall appear within a given construct.

8

For a `Priority` pragma that appears in the `declarative_part` of a `subprogram_body`, the expression shall be static, and its value shall be in the range of `System.Priority`.

Static Semantics

9

The following declarations exist in package `System`:

10

```
subtype Any_Priority is Integer range <implementation–defined>;  
subtype Priority is Any_Priority
```

```
    range Any_Priority'First .. <implementation-defined>;
subtype Interrupt_Priority is Any_Priority
    range Priority'Last+1 .. Any_Priority'Last;
```

11

```
    Default_Priority : constant Priority := (Priority'First + Priority'Last)/2;■
```

12

The full range of priority values supported by an implementation is specified by the subtype `Any_Priority`. The subrange of priority values that are high enough to require the blocking of one or more interrupts is specified by the subtype `Interrupt_Priority`. The subrange of priority values below `System.Interrupt_Priority'First` is specified by the subtype `System.-Priority`.

13

The priority specified by a `Priority` or `Interrupt_Priority` pragma is the value of the expression in the pragma, if any. If there is no expression in an `Interrupt_Priority` pragma, the priority value is `Interrupt_Priority'Last`.

Dynamic Semantics

14

A `Priority` pragma has no effect if it occurs in the `declarative_part` of the `subprogram_body` of a subprogram other than the main subprogram.

15

A `<task priority>` is an integer value that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. Unless otherwise specified, whenever tasks compete for processors or other implementation-defined resources, the resources are allocated to the task with the highest priority value. The `<base priority>` of a task is the priority with which it was created, or to which it was later set by `Dynamic_Priorities.Set_Priority` (see Section 18.5 [D.5], page 996). At all times, a task also has an `<active priority>`, which generally reflects its base priority as well as any priority it inherits from other sources. `<Priority inheritance>` is the process by which the priority of a task or other entity (e.g. a protected object; see Section 18.3 [D.3], page 991) is used in the evaluation of another task's active priority.

16

The effect of specifying such a pragma in a `protected_definition` is discussed in Section 18.3 [D.3], page 991.

17

The expression in a `Priority` or `Interrupt_Priority` pragma that appears in a `task_definition` is evaluated for each task object (see Section 10.1 [9.1], page 329). For a `Priority` pragma, the value of the expression is converted to the subtype `Priority`; for an `Interrupt_Priority` pragma, this value is converted to the subtype `Any_Priority`. The priority value is then associated with the task object whose `task_definition` contains the pragma.

18

Likewise, the priority value is associated with the environment task if the pragma appears in the `declarative_part` of the main subprogram.

19

The initial value of a task's base priority is specified by default or by means of a `Priority`

or `Interrupt_Priority` pragma. After a task is created, its base priority can be changed only by a call to `Dynamic_Priorities.Set_Priority` (see Section 18.5 [D.5], page 996). The initial base priority of a task in the absence of a pragma is the base priority of the task that creates it at the time of creation (see Section 10.1 [9.1], page 329). If a pragma `Priority` does not apply to the main subprogram, the initial base priority of the environment task is `System.Default_Priority`. The task's active priority is used when the task competes for processors. Similarly, the task's active priority is used to determine the task's position in any queue when `Priority_Queueing` is specified (see Section 18.4 [D.4], page 994).

20/2

At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant. For a task that is not held (see Section 18.11 [D.11], page 1016), its base priority is a source of priority inheritance unless otherwise specified for a particular task dispatching policy. Other sources of priority inheritance are specified under the following conditions:

21/1

- During activation, a task being activated inherits the active priority that its activator (see Section 10.2 [9.2], page 333) had at the time the activation was initiated.

22/1

- During rendezvous, the task accepting the entry call inherits the priority of the entry call (see Section 10.5.3 [9.5.3], page 352, and Section 18.4 [D.4], page 994).

23

- During a protected action on a protected object, a task inherits the ceiling priority of the protected object (see Section 10.5 [9.5], page 343, and Section 18.3 [D.3], page 991).

24

In all of these cases, the priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists.

Implementation Requirements

25

The range of `System.Interrupt_Priority` shall include at least one value.

26

The range of `System.Priority` shall include at least 30 values.

NOTES

27

- 4 The priority expression can include references to discriminants of the enclosing type.

28

5 It is a consequence of the active priority rules that at the point when a task stops inheriting a priority from another source, its active priority is re-evaluated. This is in addition to other instances described in this Annex for such re-evaluation.

29

6 An implementation may provide a non-standard mode in which tasks inherit priorities under conditions other than those specified above.

18.2 D.2 Priority Scheduling

1/2

This clause describes the rules that determine which task is selected for execution when more than one task is ready (see Chapter 10 [9], page 328).

18.2.1 D.2.1 The Task Dispatching Model

1/2

The task dispatching model specifies task scheduling, based on conceptual priority-ordered ready queues.

Static Semantics

1.1/2

The following language-defined library package exists:

1.2/2

```
package Ada.Dispatching is
  pragma Pure(Dispatching);
  Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

1.3/2

Dispatching serves as the parent of other language-defined library units concerned with task dispatching.

Dynamic Semantics

2/2

A task can become a <running task> only if it is ready (see Chapter 10 [9], page 328) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

3

Its implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.

4/2

<Task dispatching> is the process by which one ready task is selected for execution on a processor. This selection is done at certain points during the execution of a task called <task dispatching points>. A task reaches a task dispatching point whenever it becomes

blocked, and when it terminates. Other task dispatching points are defined throughout this Annex for specific policies.

5/2

<Task dispatching policies> are specified in terms of conceptual <ready queues> and task states. A ready queue is an ordered list of ready tasks. The first position in a queue is called the <head of the queue>, and the last position is called the <tail of the queue>. A task is <ready> if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor.

6/2

Each processor also has one <running task>, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs.

7/2

<This paragraph was deleted.>

8/2

<This paragraph was deleted.>

Implementation Permissions

9/2

An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation–defined effect on task dispatching.

10

An implementation may place implementation–defined restrictions on tasks whose active priority is in the Interrupt_Priority range.

10.1/2

For optimization purposes, an implementation may alter the points at which task dispatching occurs, in an implementation–defined manner. However, a delay_statement always corresponds to at least one task dispatching point.

NOTES

11

7 Section 9 specifies under which circumstances a task becomes ready. The ready state is affected by the rules for task activation and termination, delay statements, and entry calls. When a task is not ready, it is said to be blocked.

12

8 An example of a possible implementation–defined execution resource is a page of physical memory, which needs to be loaded with

a particular page of virtual memory before a task can continue execution.

13

9 The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation.

14

10 While a task is running, it is not on any ready queue. Any time the task that is running on a processor is added to a ready queue, a new running task is selected for that processor.

15

11 In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains.

16

12 The priority of a task is determined by rules specified in this subclause, and under Section 18.1 [D.1], page 975, "Section 18.1 [D.1], page 975, Task Priorities", Section 18.3 [D.3], page 991, "Section 18.3 [D.3], page 991, Priority Ceiling Locking", and Section 18.5 [D.5], page 996, "Section 18.5 [D.5], page 996, Dynamic Priorities".

17/2

13 The setting of a task's base priority as a result of a call to `Set_Priority` does not always take effect immediately when `Set_Priority` is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action.

18.2.2 D.2.2 Task Dispatching Pragmas

0.1/2

This clause allows a single task dispatching policy to be defined for all priorities, or the range of priorities to be split into subranges that are assigned individual dispatching policies.

Syntax

1

The form of a pragma `Task_Dispatching_Policy` is as follows:

2

```
pragma Task_Dispatching_Policy(<policy_>identifier);
```

2.1/2

The form of a pragma Priority_Specific_Dispatching is as follows:

2.2/2

```
pragma Priority_Specific_Dispatching (  
    <policy_>identifier,          <first_priority_>expression,  
    <last_priority_>expression);
```

Name Resolution Rules

2.3/2

The expected type for <first_priority_>expression and <last_priority_>expression is Integer.

Legality Rules

3/2

The <policy_>identifier used in a pragma Task_Dispatching_Policy shall be the name of a task dispatching policy.

3.1/2

The <policy_>identifier used in a pragma Priority_Specific_Dispatching shall be the name of a task dispatching policy.

3.2/2

Both <first_priority_>expression and <last_priority_>expression shall be static expressions in the range of System.Any_Priority; <last_priority_>expression shall have a value greater than or equal to <first_priority_>expression.

Static Semantics

3.3/2

Pragma Task_Dispatching_Policy specifies the single task dispatching policy.

3.4/2

Pragma Priority_Specific_Dispatching specifies the task dispatching policy for the specified range of priorities. Tasks with base priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma have their active priorities determined according to the specified dispatching policy. Tasks with active priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma are dispatched according to the specified dispatching policy.

3.5/2

If a partition contains one or more Priority_Specific_Dispatching pragmas the dispatching policy for priorities not covered by any Priority_Specific_Dispatching pragmas is FIFO_Within_Priorities.

Post-Compilation Rules

4/2

A Task_Dispatching_Policy pragma is a configuration pragma. A Priority_Specific_Dispatching pragma is a configuration pragma.

4.1/2

The priority ranges specified in more than one Priority_Specific_Dispatching pragma within the same partition shall not be overlapping.

4.2/2

If a partition contains one or more `Priority_Specific_Dispatching` pragmas it shall not contain a `Task_Dispatching_Policy` pragma.

5/2

<This paragraph was deleted.>

Dynamic Semantics

6/2

A <task dispatching policy> specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues. A single task dispatching policy is specified by a `Task_Dispatching_Policy` pragma. Pragma `Priority_Specific_Dispatching` assigns distinct dispatching policies to subranges of `System.Any_Priority`.

6.1/2

If neither pragma applies to any of the program units comprising a partition, the task dispatching policy for that partition is unspecified.

6.2/2

If a partition contains one or more `Priority_Specific_Dispatching` pragmas a task dispatching point occurs for the currently running task of a processor whenever there is a non-empty ready queue for that processor with a higher priority than the priority of the running task.

6.3/2

A task that has its base priority changed may move from one dispatching policy to another. It is immediately subject to the new dispatching policy.

<Paragraphs 7 through 13 were moved to D.2.3.>

Implementation Requirements

13.1/2

An implementation shall allow, for a single partition, both the locking policy (see Section 18.3 [D.3], page 991) to be specified as `Ceiling_Locking` and also one or more `Priority_Specific_Dispatching` pragmas to be given.

Documentation Requirements

<Paragraphs 14 through 16 were moved to D.2.3.>

Implementation Permissions

17/2

Implementations are allowed to define other task dispatching policies, but need not support more than one task dispatching policy per partition.

18/2

An implementation need not support pragma `Priority_Specific_Dispatching` if it is infeasible to support it in the target environment.

NOTES

<Paragraphs 19 through 21 were deleted.>

18.2.3 D.2.3 Preemptive Dispatching

1/2

This clause defines a preemptive task dispatching policy.

Static Semantics

2/2

The <policy->identifier FIFO_Within_Priorities is a task dispatching policy.

Dynamic Semantics

3/2

When FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows:

4/2

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.

5/2

- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority.

6/2

- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.

7/2

- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

8/2

Each of the events specified above is a task dispatching point (see Section 18.2.1 [D.2.1], page 978).

9/2

A task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. The currently running task is said to be <preempted> and it is added at the head of the ready queue for its active priority.

Implementation Requirements

10/2

An implementation shall allow, for a single partition, both the task dispatching policy to be specified as FIFO_Within_Priorities and also the locking policy (see Section 18.3 [D.3], page 991) to be specified as Ceiling_Locking.

Documentation Requirements

11/2

<Priority inversion> is the duration for which a task remains at the head of the highest priority nonempty ready queue while the processor executes a lower priority task. The implementation shall document:

12/2

- The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and

13/2

- whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

NOTES

14/2

14 If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).

15/2

15 Setting the base priority of a ready task causes the task to move to the tail of the queue for its active priority, regardless of whether the active priority of the task actually changes.

18.2.4 D.2.4 Non-Preemptive Dispatching

1/2

This clause defines a non-preemptive task dispatching policy.

Static Semantics

2/2

The <policy_>identifier `Non-Preemptive-FIFO-Within-Priorities` is a task dispatching policy.

Legality Rules

3/2

`Non-Preemptive-FIFO-Within-Priorities` shall not be specified as the <policy_>identifier of pragma `Priority-Specific-Dispatching` (see Section 18.2.2 [D.2.2], page 980).

Dynamic Semantics

4/2

When `Non-Preemptive-FIFO-Within-Priorities` is in effect, modifications to the ready queues occur only as follows:

5/2

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority.

6/2

- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority.

7/2

- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.

8/2

- When a task executes a `delay_statement` that does not result in blocking, it is added to the tail of the ready queue for its active priority.

9/2

For this policy, a non-blocking `delay_statement` is the only non-blocking event that is a task dispatching point (see Section 18.2.1 [D.2.1], page 978).

Implementation Requirements

10/2

An implementation shall allow, for a single partition, both the task dispatching policy to be specified as `Non_Preemptive_FIFO_Within_Priorities` and also the locking policy (see Section 18.3 [D.3], page 991) to be specified as `Ceiling_Locking`.

Implementation Permissions

11/2

Since implementations are allowed to round all ceiling priorities in subrange `System.Priority` to `System.Priority'Last` (see Section 18.3 [D.3], page 991), an implementation may allow a task to execute within a protected object without raising its active priority provided the associated protected unit does not contain pragma `Interrupt_Priority`, `Interrupt_Handler`, or `Attach_Handler`.

18.2.5 D.2.5 Round Robin Dispatching

1/2

This clause defines the task dispatching policy `Round_Robin_Within_Priorities` and the package `Round_Robin`.

Static Semantics

2/2

The `<policy>_identifier` `Round_Robin_Within_Priorities` is a task dispatching policy.

3/2

The following language-defined library package exists:

4/2

```
with System;
with Ada.Real_Time;
package Ada.Dispatching.Round_Robin is
```

```

Default_Quantum : constant Ada.Real_Time.Time_Span :=
    <implementation-defined>;
procedure Set_Quantum (Pri      : in System.Priority;
                      Quantum  : in Ada.Real_Time.Time_Span);
procedure Set_Quantum (Low, High : in System.Priority;
                      Quantum   : in Ada.Real_Time.Time_Span);
function Actual_Quantum (Pri : System.Priority) return Ada.Real_Time.Time_Span;
function Is_Round_Robin (Pri : System.Priority) return Boolean;
end Ada.Dispatching.Round_Robin;

```

5/2

When task dispatching policy `Round_Robin_Within_Priorities` is the single policy in effect for a partition, each task with priority in the range of `System.Interrupt_Priority` is dispatched according to policy `FIFO_Within_Priorities`.

Dynamic Semantics

6/2

The procedures `Set_Quantum` set the required `Quantum` value for a single priority level `Pri` or a range of priority levels `Low .. High`. If no quantum is set for a Round Robin priority level, `Default_Quantum` is used.

7/2

The function `Actual_Quantum` returns the actual quantum used by the implementation for the priority level `Pri`.

8/2

The function `Is_Round_Robin` returns `True` if priority `Pri` is covered by task dispatching policy `Round_Robin_Within_Priorities`; otherwise it returns `False`.

9/2

A call of `Actual_Quantum` or `Set_Quantum` raises exception `Dispatching.Dispatching_Policy_Error` if a predefined policy other than `Round_Robin_Within_Priorities` applies to the specified priority or any of the priorities in the specified range.

10/2

For `Round_Robin_Within_Priorities`, the dispatching rules for `FIFO_Within_Priorities` apply with the following additional rules:

11/2

- When a task is added or moved to the tail of the ready queue for its base priority, it has an execution time budget equal to the quantum for that priority level. This will also occur when a blocked task becomes executable again.

12/2

- When a task is preempted (by a higher priority task) and is added to the head of the ready queue for its priority level, it retains its remaining budget.

13/2

- While a task is executing, its budget is decreased by the amount of execution time it uses. The accuracy of this accounting is the same as that for execution time clocks (see Section 18.14 [D.14], page 1021).

14/2

- When a task has exhausted its budget and is without an inherited priority (and is not executing within a protected operation), it is moved to the tail of the ready queue for its priority level. This is a task dispatching point.

Implementation Requirements

15/2

An implementation shall allow, for a single partition, both the task dispatching policy to be specified as `Round_Robin_Within_Priorities` and also the locking policy (see Section 18.3 [D.3], page 991) to be specified as `Ceiling_Locking`.

Documentation Requirements

16/2

An implementation shall document the quantum values supported.

17/2

An implementation shall document the accuracy with which it detects the exhaustion of the budget of a task.

NOTES

18/2

16 Due to implementation constraints, the quantum value returned by `Actual_Quantum` might not be identical to that set with `Set_Quantum`.

19/2

17 A task that executes continuously with an inherited priority will not be subject to round robin dispatching.

18.2.6 D.2.6 Earliest Deadline First Dispatching

1/2

The deadline of a task is an indication of the urgency of the task; it represents a point on an ideal physical time line. The deadline might affect how resources are allocated to the task.

2/2

This clause defines a package for representing the deadline of a task and a dispatching policy that defines Earliest Deadline First (EDF) dispatching. A pragma is defined to assign an initial deadline to a task.

Syntax

3/2

The form of a pragma `Relative_Deadline` is as follows:

4/2

```
pragma Relative_Deadline (<relative_deadline_>expression);
```

Name Resolution Rules

5/2

The expected type for <relative_deadline_>expression is Real_Time.Time_Span.

Legality Rules

6/2

A Relative_Deadline pragma is allowed only immediately within a task_definition or the declarative_part of a subprogram_body. At most one such pragma shall appear within a given construct.

Static Semantics

7/2

The <policy_>identifier EDF_Across_Priorities is a task dispatching policy.

8/2

The following language-defined library package exists:

9/2

```
with Ada.Real_Time;
with Ada.Task_Identification;
package Ada.Dispatching.EDF is
  subtype Deadline is Ada.Real_Time.Time;
  Default_Deadline : constant Deadline :=
    Ada.Real_Time.Time_Last;
  procedure Set_Deadline (D : in Deadline;
    T : in Ada.Task_Identification.Task_Id :=
    Ada.Task_Identification.Current_Task);
  procedure Delay_Until_And_Set_Deadline (
    Delay_Until_Time : in Ada.Real_Time.Time;
    Deadline_Offset : in Ada.Real_Time.Time_Span);
  function Get_Deadline (T : Ada.Task_Identification.Task_Id :=
    Ada.Task_Identification.Current_Task) return Deadline;
end Ada.Dispatching.EDF;
```

Post-Compilation Rules

10/2

If the EDF_Across_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see Section 18.3 [D.3], page 991) shall also be specified for the partition.

11/2

If the EDF_Across_Priorities policy appears in a Priority_Specific_Dispatching pragma (see Section 18.2.2 [D.2.2], page 980) in a partition, then the Ceiling_Locking policy (see Section 18.3 [D.3], page 991) shall also be specified for the partition.

Dynamic Semantics

12/2

A Relative_Deadline pragma has no effect if it occurs in the declarative_part of the subprogram_body of a subprogram other than the main subprogram.

13/2

The initial absolute deadline of a task containing pragma `Relative_Deadline` is the value of `Real_Time.Clock` + `<relative_deadline_>`expression, where the call of `Real_Time.Clock` is made between task creation and the start of its activation. If there is no `Relative_Deadline` pragma then the initial absolute deadline of a task is the value of `Default_Deadline`. The environment task is also given an initial deadline by this rule.

14/2

The procedure `Set_Deadline` changes the absolute deadline of the task to `D`. The function `Get_Deadline` returns the absolute deadline of the task.

15/2

The procedure `Delay_Until_And_Set_Deadline` delays the calling task until time `Delay_Until_Time`. When the task becomes runnable again it will have deadline `Delay_Until_Time` + `Deadline_Offset`.

16/2

On a system with a single processor, the setting of the deadline of a task to the new value occurs immediately at the first point that is outside the execution of a protected action. If the task is currently on a ready queue it is removed and re-entered on to the ready queue determined by the rules defined below.

17/2

When `EDF_Across_Priorities` is specified for priority range `<Low>..<High>` all ready queues in this range are ordered by deadline. The task at the head of a queue is the one with the earliest deadline.

18/2

A task dispatching point occurs for the currently running task `<T>` to which policy `EDF_Across_Priorities` applies:

19/2

- when a change to the deadline of `<T>` occurs;

20/2

- there is a task on the ready queue for the active priority of `<T>` with a deadline earlier than the deadline of `<T>`; or

21/2

- there is a non-empty ready queue for that processor with a higher priority than the active priority of the running task.

22/2

In these cases, the currently running task is said to be preempted and is returned to the ready queue for its active priority.

23/2

For a task `<T>` to which policy `EDF_Across_Priorities` applies, the base priority is not a source of priority inheritance; the active priority when first activated or while it is blocked is defined as the maximum of the following:

24/2

- the lowest priority in the range specified as EDF_Across_Priorities that includes the base priority of <T>;

25/2

- the priorities, if any, currently inherited by <T>;

26/2

- the highest priority <P>, if any, less than the base priority of <T> such that one or more tasks are executing within a protected object with ceiling priority <P> and task <T> has an earlier deadline than all such tasks.

27/2

When a task <T> is first activated or becomes unblocked, it is added to the ready queue corresponding to this active priority. Until it becomes blocked again, the active priority of <T> remains no less than this value; it will exceed this value only while it is inheriting a higher priority.

28/2

When the setting of the base priority of a ready task takes effect and the new priority is in a range specified as EDF_Across_Priorities, the task is added to the ready queue corresponding to its new active priority, as determined above.

29/2

For all the operations defined in Dispatching.EDF, Tasking_Error is raised if the task identified by T has terminated. Program_Error is raised if the value of T is Null_Task_Id.

Bounded (Run-Time) Errors

30/2

If EDF_Across_Priorities is specified for priority range <Low>..<High>, it is a bounded error to declare a protected object with ceiling priority <Low> or to assign the value <Low> to attribute 'Priority. In either case either Program_Error is raised or the ceiling of the protected object is assigned the value <Low>+1.

Erroneous Execution

31/2

If a value of Task_Id is passed as a parameter to any of the subprograms of this package and the corresponding task object no longer exists, the execution of the program is erroneous.

Documentation Requirements

32/2

On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the deadline of a task to be delayed later than what is specified for a single processor.

NOTES

33/2

18 If two adjacent priority ranges, $\langle A \rangle.. \langle B \rangle$ and $\langle B \rangle + 1.. \langle C \rangle$ are specified to have policy `EDF_Across_Priorities` then this is not equivalent to this policy being specified for the single range, $\langle A \rangle.. \langle C \rangle$.

34/2

19 The above rules implement the preemption–level protocol (also called Stack Resource Policy protocol) for resource sharing under EDF dispatching. The preemption–level for a task is denoted by its base priority. The definition of a ceiling preemption–level for a protected object follows the existing rules for ceiling locking.

18.3 D.3 Priority Ceiling Locking

1

This clause specifies the interactions between priority task scheduling and protected object ceilings. This interaction is based on the concept of the $\langle \text{ceiling priority} \rangle$ of a protected object.

Syntax

2

The form of a pragma `Locking_Policy` is as follows:

3

```
pragma Locking_Policy(<policy_>identifier);  
Legality Rules
```

4

The $\langle \text{policy_} \rangle \text{identifier}$ shall either be `Ceiling_Locking` or an implementation–defined identifier.

Post-Compilation Rules

5

A `Locking_Policy` pragma is a configuration pragma.

Dynamic Semantics

6/2

A locking policy specifies the details of protected object locking. All protected objects have a priority. The locking policy specifies the meaning of the priority of a protected object, and the relationships between these priorities and task priorities. In addition, the policy specifies the state of a task when it executes a protected action, and how its active priority is affected by the locking. The $\langle \text{locking policy} \rangle$ is specified by a `Locking_Policy` pragma. For implementation–defined locking policies, the meaning of the priority of a protected object is implementation defined. If no `Locking_Policy` pragma applies to any of the program units comprising a partition, the locking policy for that partition, as well as the meaning of the priority of a protected object, are implementation defined.

6.1/2

The expression of a `Priority` or `Interrupt_Priority` pragma (see Section 18.1 [D.1], page 975) is evaluated as part of the creation of the corresponding protected object and converted to

the subtype `System.Any_Priority` or `System.Interrupt_Priority`, respectively. The value of the expression is the initial priority of the corresponding protected object. If no `Priority` or `Interrupt_Priority` pragma applies to a protected object, the initial priority is specified by the locking policy.

7

There is one predefined locking policy, `Ceiling_Locking`; this policy is defined as follows:

8/2

- Every protected object has a <ceiling priority>, which is determined by either a `Priority` or `Interrupt_Priority` pragma as defined in Section 18.1 [D.1], page 975, or by assignment to the `Priority` attribute as described in Section 18.5.2 [D.5.2], page 998. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.

9/2

- The initial ceiling priority of a protected object is equal to the initial priority for that object.

10/2

- If an `Interrupt_Handler` or `Attach_Handler` pragma (see Section 17.3.1 [C.3.1], page 954) appears in a `protected_definition` without an `Interrupt_Priority` pragma, the initial priority of protected objects of that type is implementation defined, but in the range of the subtype `System.Interrupt_Priority`.

11/2

- If no pragma `Priority`, `Interrupt_Priority`, `Interrupt_Handler`, or `Attach_Handler` is specified in the `protected_definition`, then the initial priority of the corresponding protected object is `System.Priority'Last`.

12

- While a task executes a protected action, it inherits the ceiling priority of the corresponding protected object.

13

- When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; `Program_Error` is raised if this check fails.

Bounded (Run-Time) Errors

13.1/2

Following any change of priority, it is a bounded error for the active priority of any task with a call queued on an entry of a protected object to be higher than the ceiling priority of the protected object. In this case one of the following applies:

13.2/2

- at any time prior to executing the entry body `Program_Error` is raised in the calling task;

13.3/2

- when the entry is open the entry body is executed at the ceiling priority of the protected object;

13.4/2

- when the entry is open the entry body is executed at the ceiling priority of the protected object and then `Program_Error` is raised in the calling task; or

13.5/2

- when the entry is open the entry body is executed at the ceiling priority of the protected object that was in effect when the entry call was queued.

Implementation Permissions

14

The implementation is allowed to round all ceilings in a certain subrange of `System.Priority` or `System.Interrupt_Priority` up to the top of that subrange, uniformly.

15/2

Implementations are allowed to define other locking policies, but need not support more than one locking policy per partition.

16

Since implementations are allowed to place restrictions on code that runs at an interrupt-level active priority (see Section 17.3.1 [C.3.1], page 954, and Section 18.2.1 [D.2.1], page 978), the implementation may implement a language feature in terms of a protected object with an implementation-defined ceiling, but the ceiling shall be no less than `Priority'Last`.

Implementation Advice

17

The implementation should use names that end with `"_Locking"` for implementation-defined locking policies.

NOTES

18

20 While a task executes in a protected action, it can be preempted only by tasks whose active priorities are higher than the ceiling priority of the protected object.

19

21 If a protected object has a ceiling priority in the range of Interrupt.Priority, certain interrupts are blocked while protected actions of that object execute. In the extreme, if the ceiling is Interrupt.Priority'Last, all blockable interrupts are blocked during that time.

20

22 The ceiling priority of a protected object has to be in the Interrupt.Priority range if one of its procedures is to be used as an interrupt handler (see Section 17.3 [C.3], page 951).

21

23 When specifying the ceiling of a protected object, one should choose a value that is at least as high as the highest active priority at which tasks can be executing when they call protected operations of that object. In determining this value the following factors, which can affect active priority, should be considered: the effect of Set.Priority, nested protected operations, entry calls, task activation, and other implementation-defined factors.

22

24 Attaching a protected procedure whose ceiling is below the interrupt hardware priority to an interrupt causes the execution of the program to be erroneous (see Section 17.3.1 [C.3.1], page 954).

23

25 On a single processor implementation, the ceiling priority rules guarantee that there is no possibility of deadlock involving only protected subprograms (excluding the case where a protected operation calls another protected operation on the same protected object).

18.4 D.4 Entry Queuing Policies

1/1

This clause specifies a mechanism for a user to choose an entry <queuing policy>. It also defines two such policies. Other policies are implementation defined.

Syntax

2

The form of a pragma Queuing_Policy is as follows:

3

```
pragma Queuing_Policy(<policy_>identifier);  
Legality Rules
```

4

The <policy>identifier shall be either FIFO_Queueing, Priority_Queueing or an implementation-defined identifier.

Post-Compilation Rules

5

A Queuing_Policy pragma is a configuration pragma.

Dynamic Semantics

6

A <queuing policy> governs the order in which tasks are queued for entry service, and the order in which different entry queues are considered for service. The queuing policy is specified by a Queuing_Policy pragma.

7/2

Two queuing policies, FIFO_Queueing and Priority_Queueing, are language defined. If no Queuing_Policy pragma applies to any of the program units comprising the partition, the queuing policy for that partition is FIFO_Queueing. The rules for this policy are specified in Section 10.5.3 [9.5.3], page 352, and Section 10.7.1 [9.7.1], page 378.

8

The Priority_Queueing policy is defined as follows:

9

- The calls to an entry (including a member of an entry family) are queued in an order consistent with the priorities of the calls. The <priority of an entry call> is initialized from the active priority of the calling task at the time the call is made, but can change later. Within the same priority, the order is consistent with the calling (or requeuing, or priority setting) time (that is, a FIFO order).

10/1

- After a call is first queued, changes to the active priority of a task do not affect the priority of the call, unless the base priority of the task is set while the task is blocked on an entry call.

11

- When the base priority of a task is set (see Section 18.5 [D.5], page 996), if the task is blocked on an entry call, and the call is queued, the priority of the call is updated to the new active priority of the calling task. This causes the call to be removed from and then reinserted in the queue at the new active priority.

12

- When more than one condition of an entry_barrier of a protected object becomes True, and more than one of the respective queues is nonempty, the call with the highest priority is selected. If more than one such call has the same priority, the call that is queued on the entry whose declaration is first in textual order in the protected_definition is selected. For members of the same entry family, the one with the lower family index is selected.

13

- If the expiration time of two or more open `delay_alternatives` is the same and no other `accept_alternatives` are open, the `sequence_of_statements` of the `delay_alternative` that is first in textual order in the `selective_accept` is executed.

14

- When more than one alternative of a `selective_accept` is open and has queued calls, an alternative whose queue has the highest-priority call at its head is selected. If two or more open alternatives have equal-priority queued calls, then a call on the entry in the `accept_alternative` that is first in textual order in the `selective_accept` is selected.

Implementation Permissions

15/2

Implementations are allowed to define other queuing policies, but need not support more than one queuing policy per partition.

15.1/2

Implementations are allowed to defer the reordering of entry queues following a change of base priority of a task blocked on the entry call if it is not practical to reorder the queue immediately.

Implementation Advice

16

The implementation should use names that end with `"_Queuing"` for implementation-defined queuing policies.

18.5 D.5 Dynamic Priorities

1/2

This clause describes how the priority of an entity can be modified or queried at run time.

18.5.1 D.5.1 Dynamic Priorities for Tasks

1

This clause describes how the base priority of a task can be modified or queried at run time.

Static Semantics

2

The following language-defined library package exists:

3/2

```
with System;
with Ada.Task_Identification; <-- See Section 17.7.1 [C.7.1], page 965>
package Ada.Dynamic_Priorities is
  pragma Preelaborate(Dynamic_Priorities);
```

4

```
  procedure Set_Priority(Priority : in System.Any_Priority;
```

```
T : in Ada.Task_Identification.Task_Id :=  
Ada.Task_Identification.Current_Task);
```

5

```
function Get_Priority (T : Ada.Task_Identification.Task_Id :=  
Ada.Task_Identification.Current_Task)  
return System.Any_Priority;
```

6

```
end Ada.Dynamic_Priorities;  
Dynamic Semantics
```

7

The procedure `Set_Priority` sets the base priority of the specified task to the specified Priority value. `Set_Priority` has no effect if the task is terminated.

8

The function `Get_Priority` returns T's current base priority. `Tasking_Error` is raised if the task is terminated.

9

`Program_Error` is raised by `Set_Priority` and `Get_Priority` if T is equal to `Null_Task_Id`.

10/2

On a system with a single processor, the setting of the base priority of a task <T> to the new value occurs immediately at the first point when <T> is outside the execution of a protected action.

Bounded (Run-Time) Errors

11/2

<This paragraph was deleted.>

Erroneous Execution

12

If any subprogram in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous.

Documentation Requirements

12.1/2

On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the priority of a task to be delayed later than what is specified for a single processor.

Metrics

13

The implementation shall document the following metric:

14

- The execution time of a call to `Set_Priority`, for the nonpreempting case, in processor clock cycles. This is measured for a call that modifies the priority of a ready task that is not running (which cannot be the calling one), where the new base priority of the

affected task is lower than the active priority of the calling task, and the affected task is not on any entry queue and is not executing a protected operation.

NOTES

15/2

26 Setting a task's base priority affects task dispatching. First, it can change the task's active priority. Second, under the `FIFO_Within_Priorities` policy it always causes the task to move to the tail of the ready queue corresponding to its active priority, even if the new base priority is unchanged.

16

27 Under the priority queuing policy, setting a task's base priority has an effect on a queued entry call if the task is blocked waiting for the call. That is, setting the base priority of a task causes the priority of a queued entry call from that task to be updated and the call to be removed and then reinserted in the entry queue at the new priority (see Section 18.4 [D.4], page 994), unless the call originated from the `triggering_statement` of an `asynchronous_select`.

17

28 The effect of two or more `Set_Priority` calls executed in parallel on the same task is defined as executing these calls in some serial order.

18

29 The rule for when `Tasking_Error` is raised for `Set_Priority` or `Get_Priority` is different from the rule for when `Tasking_Error` is raised on an entry call (see Section 10.5.3 [9.5.3], page 352). In particular, setting or querying the priority of a completed or an abnormal task is allowed, so long as the task is not yet terminated.

19

30 Changing the priorities of a set of tasks can be performed by a series of calls to `Set_Priority` for each task separately. For this to work reliably, it should be done within a protected operation that has high enough ceiling priority to guarantee that the operation completes without being preempted by any of the affected tasks.

18.5.2 D.5.2 Dynamic Priorities for Protected Objects

1/2

This clause specifies how the priority of a protected object can be modified or queried at run time.

Static Semantics

2/2

The following attribute is defined for a prefix P that denotes a protected object:

3/2

P'Priority

Denotes a non-aliased component of the protected object P. This component is of type System.Any_Priority and its value is the priority of P. P'Priority denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P.

4/2

The initial value of this attribute is the initial value of the priority of the protected object, and can be changed by an assignment.

Dynamic Semantics

5/2

If the locking policy Ceiling_Locking (see Section 18.3 [D.3], page 991) is in effect then the ceiling priority of a protected object <P> is set to the value of <P>'Priority at the end of each protected action of <P>.

6/2

If the locking policy Ceiling_Locking is in effect, then for a protected object <P> with either an Attach_Handler or Interrupt_Handler pragma applying to one of its procedures, a check is made that the value to be assigned to <P>'Priority is in the range System.Interrupt_Priority. If the check fails, Program_Error is raised.

Metrics

7/2

The implementation shall document the following metric:

8/2

- The difference in execution time of calls to the following procedures in protected object P:

9/2

```
protected P is
  procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority);
  procedure Set_Ceiling (Pr : System.Any_Priority);
```

end P;

10/2

```
protected body P is
  procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority) is
  begin
    null;
  end;
  procedure Set_Ceiling (Pr : System.Any_Priority) is
  begin
    P'Priority := Pr;
  end;
end P;
```

NOTES

11/2

31 Since P'Priority is a normal variable, the value following an assignment to the attribute immediately reflects the new value even though its impact on the ceiling priority of P is postponed until completion of the protected action in which it is executed.

18.6 D.6 Preemptive Abort

1

This clause specifies requirements on the immediacy with which an aborted construct is completed.

Dynamic Semantics

2

On a system with a single processor, an aborted construct is completed immediately at the first point that is outside the execution of an abort-deferred operation.

Documentation Requirements

3

On a multiprocessor, the implementation shall document any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor.

Metrics

4

The implementation shall document the following metrics:

5

- The execution time, in processor clock cycles, that it takes for an abort_statement to cause the completion of the aborted task. This is measured in a situation where a task T2 preempts task T1 and aborts T1. T1 does not have any finalization code. T2 shall verify that T1 has terminated, by means of the Terminated attribute.

6

- On a multiprocessor, an upper bound in seconds, on the time that the completion of an aborted task can be delayed beyond the point that it is required for a single processor.

7/2

- An upper bound on the execution time of an asynchronous_select, in processor clock cycles. This is measured between a point immediately before a task T1 executes a protected operation Pr.Set that makes the condition of an entry_barrier Pr.Wait True, and the point where task T2 resumes execution immediately after an entry call to Pr.Wait in an asynchronous_select. T1 preempts T2 while T2 is executing the abortable part, and then blocks itself so that T2 can execute. The execution time of T1 is measured separately, and subtracted.

8

- An upper bound on the execution time of an asynchronous_select, in the case that no asynchronous transfer of control takes place. This is measured between a point immediately before a task executes the asynchronous_select with a nonnull abortable part, and the point where the task continues execution immediately after it. The execution time of the abortable part is subtracted.

Implementation Advice

9

Even though the abort_statement is included in the list of potentially blocking operations (see Section 10.5.1 [9.5.1], page 344), it is recommended that this statement be implemented in a way that never requires the task executing the abort_statement to block.

10

On a multi-processor, the delay associated with aborting a task on another processor should be bounded; the implementation should use periodic polling, if necessary, to achieve this.

NOTES

11

32 Abortion does not change the active or base priority of the aborted task.

12

33 Abortion cannot be more immediate than is allowed by the rules for deferral of abortion during finalization and in protected actions.

18.7 D.7 Tasking Restrictions

1

This clause defines restrictions that can be used with a pragma Restrictions (see Section 14.12 [13.12], page 535) to facilitate the construction of highly efficient tasking run-time systems.

Static Semantics

2

The following <restriction_>identifiers are language defined:

3

No_Task_Hierarchy

All (nonenvironment) tasks depend directly on the environment task of the partition.

4/2

No_Nested_Finalization

Objects of a type that needs finalization (see Section 8.6 [7.6], page 295) and access types that designate a type that needs finalization shall be declared only at library level.

5

No_Abort_Statements

There are no abort_statements, and there are no calls on Task_Identification.Abort_Task. ■

6

No_Terminate_Alternatives

There are no selective_accepts with terminate_alternatives.

7

No_Task_Allocators

There are no allocators for task types or types containing task subcomponents.

8

No_Implicit_Heap_Allocations

There are no operations that implicitly

require heap storage allocation to be performed by the implementation.

The operations that implicitly require heap storage allocation are implementation defined.

9/2

No_Dynamic_Priorities

There are no semantic dependences on the package Dynamic_Priorities, and no occurrences of the attribute Priority.

10/2

No_Dynamic_Attachment

There is no call to any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

10.1/2

No_Local_Protected_Objects

Protected objects shall be declared only at library level.

10.2/2

No_Local_Timing_Events

Timing_Events shall be declared only at library level.

10.3/2

No_Protected_Type_Allocators

There are no allocators for

protected types or types containing protected type subcomponents.

10.4/2
No_Relative_Delay

There are no delay_relative_statements.

10.5/2
No_Requeue_Statements

There are no requeue_statements.

10.6/2
No_Select_Statements

There are no select_statements.

10.7/2
No_Specific_Termination_Handlers

There are no calls to the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination.

10.8/2
Simple_Barriers

The Boolean expression in an entry barrier shall be either a static Boolean expression or a Boolean component of the enclosing protected object.

11
The following <restriction_parameter_>identifiers are language defined:

12
Max_Select_Alternatives

Specifies the maximum number of alternatives in a selective_accept.

13
Max_Task_Entries

Specifies the maximum number of entries per task. The bounds of every entry family of a task unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static. A value of zero indicates that no rendezvous are possible.

14

Max_Protected_Entries

Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static.

Dynamic Semantics

15/2

The following <restriction_>identifier is language defined:

15.1/2

No_Task_Termination

All tasks are non-terminating. It is implementation-defined what happens if a task attempts to terminate. If there is a fall-back handler (see C.7.3) set for the partition it should be called when the first task attempts to terminate.



16

The following <restriction_parameter>identifiers are language defined:

17/1

Max_Storage_At_Blocking

Specifies the maximum portion (in storage elements) of a task's Storage_Size that can be retained by a blocked task. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; otherwise, the behavior is implementation defined.

18/1

Max_Asynchronous_Select_Nesting

Specifies the maximum dynamic nesting level of asynchronous_selects. A value of zero prevents the use of any asynchronous_select (see [S0223], page 384) and, if a program contains an asynchronous_select (see [S0223], page 384), it is illegal. If an implementation chooses to detect a violation of this restriction for values other than zero, Storage_Error should be raised; otherwise, the behavior is implementation defined.

19/1

Max_Tasks

Specifies the maximum number of task creations that may be executed over the lifetime of a partition, not counting the creation of the environment task. A value of zero prevents any task creation and, if a program contains a task creation, it is illegal. If an implementation chooses to detect a violation of this restriction, `Storage_Error` should be raised; otherwise, the behavior is implementation defined.

19.1/2

`Max_Entry_Queue_Length`

`Max_Entry_Queue_Length` defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of `Program_Error` at the point of the call or requeue.

20

It is implementation defined whether the use of pragma Restrictions results in a reduction in executable program size, storage requirements, or execution time. If possible, the implementation should provide quantitative descriptions of such effects for each restriction.

Implementation Advice

21

When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation.

NOTES

22

34 The above `Storage_Checks` can be suppressed with pragma `Suppress`.

18.8 D.8 Monotonic Time

1

This clause specifies a high-resolution, monotonic clock package.

Static Semantics

2

The following language-defined library package exists:

3

```
package Ada.Real_Time is
```

4

```
    type Time is private;
    Time_First : constant Time;
    Time_Last : constant Time;
    Time_Unit : constant := <implementation-defined-real-number>;
```

5

```
    type Time_Span is private;
    Time_Span_First : constant Time_Span;
    Time_Span_Last : constant Time_Span;
    Time_Span_Zero : constant Time_Span;
    Time_Span_Unit : constant Time_Span;
```

6

```
    Tick : constant Time_Span;
    function Clock return Time;
```

7

```
    function "+" (Left : Time; Right : Time_Span) return Time;
    function "+" (Left : Time_Span; Right : Time) return Time;
    function "-" (Left : Time; Right : Time_Span) return Time;
    function "-" (Left : Time; Right : Time) return Time_Span;
```

8

```
    function "<" (Left, Right : Time) return Boolean;
    function "<=" (Left, Right : Time) return Boolean;
    function ">" (Left, Right : Time) return Boolean;
    function ">=" (Left, Right : Time) return Boolean;
```

9

```
    function "+" (Left, Right : Time_Span) return Time_Span;
```



```
function "-" (Left, Right : Time_Span) return Time_Span;
function "-" (Right : Time_Span) return Time_Span;
function "*" (Left : Time_Span; Right : Integer) return Time_Span;
function "*" (Left : Integer; Right : Time_Span) return Time_Span;
function "/" (Left, Right : Time_Span) return Integer;
function "/" (Left : Time_Span; Right : Integer) return Time_Span;
```

10

```
function "abs"(Right : Time_Span) return Time_Span;
```

11/1

<This paragraph was deleted.>

12

```
function "<" (Left, Right : Time_Span) return Boolean;
function "<=" (Left, Right : Time_Span) return Boolean;
function ">" (Left, Right : Time_Span) return Boolean;
function ">=" (Left, Right : Time_Span) return Boolean;
```

13

```
function To_Duration (TS : Time_Span) return Duration;
function To_Time_Span (D : Duration) return Time_Span;
```

14/2

```
function Nanoseconds (NS : Integer) return Time_Span;
function Microseconds (US : Integer) return Time_Span;
function Milliseconds (MS : Integer) return Time_Span;
function Seconds (S : Integer) return Time_Span;
function Minutes (M : Integer) return Time_Span;
```

15

```
type Seconds_Count is range <implementation-defined>;
```

16

```
procedure Split(T : in Time; SC : out Seconds_Count; TS : out Time_Span);
function Time_Of(SC : Seconds_Count; TS : Time_Span) return Time;
```

17

```
private
  ... -- <not specified by the language>
end Ada.Real_Time;
```

18

In this Annex, <real time> is defined to be the physical time as observed in the external

environment. The type `Time` is a `<time type>` as defined by Section 10.6 [9.6], page 358; values of this type may be used in a `delay_until` statement. Values of this type represent segments of an ideal time line. The set of values of the type `Time` corresponds one-to-one with an implementation-defined range of mathematical integers.

19

The `Time` value `I` represents the half-open real time interval that starts with $E+I*\text{Time_Unit}$ and is limited by $E+(I+1)*\text{Time_Unit}$, where `Time_Unit` is an implementation-defined real number and `E` is an unspecified origin point, the `<epoch>`, that is the same for all values of the type `Time`. It is not specified by the language whether the time values are synchronized with any standard time reference. For example, `E` can correspond to the time of system initialization or it can correspond to the epoch of some time standard.

20

Values of the type `Time_Span` represent length of real time duration. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers. The `Time_Span` value corresponding to the integer `I` represents the real-time duration $I*\text{Time_Unit}$.

21

`Time_First` and `Time_Last` are the smallest and largest values of the `Time` type, respectively. Similarly, `Time_Span_First` and `Time_Span_Last` are the smallest and largest values of the `Time_Span` type, respectively.

22

A value of type `Seconds_Count` represents an elapsed time, measured in seconds, since the epoch.

Dynamic Semantics

23

`Time_Unit` is the smallest amount of real time representable by the `Time` type; it is expressed in seconds. `Time_Span_Unit` is the difference between two successive values of the `Time` type. It is also the smallest positive value of type `Time_Span`. `Time_Unit` and `Time_Span_Unit` represent the same real time duration. A `<clock tick>` is a real time interval during which the clock value (as observed by calling the `Clock` function) remains constant. `Tick` is the average length of such intervals.

24/2

The function `To_Duration` converts the value `TS` to a value of type `Duration`. Similarly, the function `To_Time_Span` converts the value `D` to a value of type `Time_Span`. For `To_Duration`, the result is rounded to the nearest value of type `Duration` (away from zero if exactly halfway between two values). If the result is outside the range of `Duration`, `Constraint_Error` is raised. For `To_Time_Span`, the value of `D` is first rounded to the nearest integral multiple of `Time_Unit`, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of `Time_Span`, `Constraint_Error` is raised. Otherwise, the value is converted to the type `Time_Span`.

25

`To_Duration(Time_Span_Zero)` returns `0.0`, and `To_Time_Span(0.0)` returns `Time_Span_Zero`.

26/2

The functions `Nanoseconds`, `Microseconds`, `Milliseconds`, `Seconds`, and `Minutes` convert the

input parameter to a value of the type `Time_Span`. `NS`, `US`, `MS`, `S`, and `M` are interpreted as a number of nanoseconds, microseconds, milliseconds, seconds, and minutes respectively. The input parameter is first converted to seconds and rounded to the nearest integral multiple of `Time_Unit`, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of `Time_Span`, `Constraint_Error` is raised. Otherwise, the rounded value is converted to the type `Time_Span`.

27

The effects of the operators on `Time` and `Time_Span` are as for the operators defined for integer types.

28

The function `Clock` returns the amount of time since the epoch.

29

The effects of the `Split` and `Time_Of` operations are defined as follows, treating values of type `Time`, `Time_Span`, and `Seconds_Count` as mathematical integers. The effect of `Split(T,SC,TS)` is to set `SC` and `TS` to values such that $T * \text{Time_Unit} = SC * 1.0 + TS * \text{Time_Unit}$, and $0.0 \leq TS * \text{Time_Unit} < 1.0$. The value returned by `Time_Of(SC,TS)` is the value `T` such that $T * \text{Time_Unit} = SC * 1.0 + TS * \text{Time_Unit}$.

Implementation Requirements

30

The range of `Time` values shall be sufficient to uniquely represent the range of real times from program start—up to 50 years later. `Tick` shall be no greater than 1 millisecond. `Time_Unit` shall be less than or equal to 20 microseconds.

31

`Time_Span_First` shall be no greater than -3600 seconds, and `Time_Span_Last` shall be no less than 3600 seconds.

32

A <clock jump> is the difference between two successive distinct values of the clock (as observed by calling the `Clock` function). There shall be no backward clock jumps.

Documentation Requirements

33

The implementation shall document the values of `Time_First`, `Time_Last`, `Time_Span_First`, `Time_Span_Last`, `Time_Span_Unit`, and `Tick`.

34

The implementation shall document the properties of the underlying time base used for the clock and for type `Time`, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

35

The implementation shall document whether or not there is any synchronization with external time references, and if such synchronization exists, the sources of synchronization information, the frequency of synchronization, and the synchronization method applied.

36/1

The implementation shall document any aspects of the external environment that could interfere with the clock behavior as defined in this clause.

Metrics

37

For the purpose of the metrics defined in this clause, real time is defined to be the International Atomic Time (TAI).

38

The implementation shall document the following metrics:

39

- An upper bound on the real-time duration of a clock tick. This is a value D such that if t1 and t2 are any real times such that t1 < t2 and Clockt1 = Clockt2 then t2 - t1 <= D.

40

- An upper bound on the size of a clock jump.

41

- An upper bound on the <drift rate> of Clock with respect to real time. This is a real number D such that

42

$$E*(1-D) \leq (\text{Clockt}+E - \text{Clockt}) \leq E*(1+D)$$

provided that: $\text{Clockt} + E*(1+D) \leq \text{Time_Last}$.

43

- where Clockt is the value of Clock at time t, and E is a real time duration not less than 24 hours. The value of E used for this metric shall be reported.

44

- An upper bound on the execution time of a call to the Clock function, in processor clock cycles.

45

- Upper bounds on the execution times of the operators of the types Time and Time.Span, in processor clock cycles.

Implementation Permissions

46

Implementations targeted to machines with word size smaller than 32 bits need not support the full range and granularity of the Time and Time.Span types.

Implementation Advice

47

When appropriate, implementations should provide configuration mechanisms to change the value of Tick.

48

It is recommended that `Calendar.Clock` and `Real_Time.Clock` be implemented as transformations of the same time base.

49

It is recommended that the "best" time base which exists in the underlying system be available to the application through `Clock`. "Best" may mean highest accuracy or largest range.

NOTES

50

35 The rules in this clause do not imply that the implementation can protect the user from operator or installation errors which could result in the clock being set incorrectly.

51

36 `Time_Unit` is the granularity of the `Time` type. In contrast, `Tick` represents the granularity of `Real_Time.Clock`. There is no requirement that these be the same.

18.9 D.9 Delay Accuracy

1

This clause specifies performance requirements for the `delay_statement`. The rules apply both to `delay_relative_statement` (see [S0211], page 359) and to `delay_until_statement` (see [S0210], page 359). Similarly, they apply equally to a simple `delay_statement` (see [S0209], page 359) and to one which appears in a `delay_alternative` (see [S0217], page 379).

Dynamic Semantics

2

The effect of the `delay_statement` for `Real_Time.Time` is defined in terms of `Real_Time.Clock`:

3

- If `C1` is a value of `Clock` read before a task executes a `delay_relative_statement` with duration `D`, and `C2` is a value of `Clock` read after the task resumes execution following that `delay_statement`, then $C2 - C1 \geq D$.

4

- If `C` is a value of `Clock` read after a task resumes execution following a `delay_until_statement` with `Real_Time.Time` value `T`, then $C \geq T$.

5

A simple `delay_statement` with a negative or zero value for the expiration time does not cause the calling task to be blocked; it is nevertheless a potentially blocking operation (see Section 10.5.1 [9.5.1], page 344).

6/2

When a `delay_statement` appears in a `delay_alternative` of a `timed_entry_call` the selection of the entry call is attempted, regardless of the specified expiration time. When a `delay_statement` appears in a `select_alternative`, and a call is queued on one of the open entries, the selection of that entry call proceeds, regardless of the value of the delay expression.

Documentation Requirements

7

The implementation shall document the minimum value of the delay expression of a `delay_relative_statement` that causes the task to actually be blocked.

8

The implementation shall document the minimum difference between the value of the delay expression of a `delay_until_statement` and the value of `Real_Time.Clock`, that causes the task to actually be blocked.

Metrics

9

The implementation shall document the following metrics:

10

- An upper bound on the execution time, in processor clock cycles, of a `delay_relative_statement` whose requested value of the delay expression is less than or equal to zero.

11

- An upper bound on the execution time, in processor clock cycles, of a `delay_until_statement` whose requested value of the delay expression is less than or equal to the value of `Real_Time.Clock` at the time of executing the statement. Similarly, for `Calendar.Clock`.

12

- An upper bound on the <lateness> of a `delay_relative_statement`, for a positive value of the delay expression, in a situation where the task has sufficient priority to preempt the processor as soon as it becomes ready, and does not need to wait for any other execution resources. The upper bound is expressed as a function of the value of the delay expression. The lateness is obtained by subtracting the value of the delay expression from the <actual duration>. The actual duration is measured from a point immediately before a task executes the `delay_statement` to a point immediately after the task resumes execution following this statement.

13

- An upper bound on the lateness of a `delay_until_statement`, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it does not need to wait for any other execution resources. The upper

bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a `delay_until` statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement.

NOTES

14/2

<This paragraph was deleted.>

18.10 D.10 Synchronous Task Control

1

This clause describes a language-defined private semaphore (suspension object), which can be used for <two-stage suspend> operations and as a simple building block for implementing higher-level queues.

Static Semantics

2

The following language-defined package exists:

3/2

```
package Ada.Synchronous_Task_Control is
  pragma Preelaborate(Synchronous_Task_Control);
```

4

```
  type Suspension_Object is limited private;
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object);
private
  ... -- <not specified by the language>
end Ada.Synchronous_Task_Control;
```

5

The type `Suspension_Object` is a by-reference type.

Dynamic Semantics

6/2

An object of the type `Suspension_Object` has two visible states: `True` and `False`. Upon initialization, its value is set to `False`.

7/2

The operations `Set_True` and `Set_False` are atomic with respect to each other and with respect to `Suspend_Until_True`; they set the state to `True` and `False` respectively.

8

`Current_State` returns the current state of the object.

9/2

The procedure `Suspend_Until_True` blocks the calling task until the state of the object `S` is `True`; at that point the task becomes ready and the state of the object becomes `False`.

10

`Program_Error` is raised upon calling `Suspend_Until_True` if another task is already waiting on that suspension object. `Suspend_Until_True` is a potentially blocking operation (see Section 10.5.1 [9.5.1], page 344).

Implementation Requirements

11

The implementation is required to allow the calling of `Set_False` and `Set_True` during any protected action, even one that has its ceiling priority in the `Interrupt_Priority` range.

18.11 D.11 Asynchronous Task Control

1

This clause introduces a language-defined package to do asynchronous suspend/resume on tasks. It uses a conceptual `<held priority>` value to represent the task's `<held>` state.

Static Semantics

2

The following language-defined library package exists:

3/2

```
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control is
  pragma Preelaborate(Asynchronous_Task_Control);
  procedure Hold(T : in Ada.Task_Identification.Task_Id);
  procedure Continue(T : in Ada.Task_Identification.Task_Id);
  function Is_Held(T : Ada.Task_Identification.Task_Id)
    return Boolean;
end Ada.Asynchronous_Task_Control;
```

Dynamic Semantics

4/2

After the `Hold` operation has been applied to a task, the task becomes `<held>`. For each processor there is a conceptual `<idle task>`, which is always ready. The base priority of the idle task is below `System.Any_Priority'First`. The `<held priority>` is a constant of the type `Integer` whose value is below the base priority of the idle task.

4.1/2

For any priority below `System.Any_Priority'First`, the task dispatching policy is `FIFO_Within_Priorities`.

5/2

The `Hold` operation sets the state of `T` to `held`. For a `held` task, the active priority is reevaluated as if the base priority of the task were the `held` priority.

6/2

The `Continue` operation resets the state of `T` to `not-held`; its active priority is then reevaluated as determined by the task dispatching policy associated with its base priority.

7

The `Is_Held` function returns `True` if and only if `T` is in the held state.

8

As part of these operations, a check is made that the task identified by `T` is not terminated. `Tasking_Error` is raised if the check fails. `Program_Error` is raised if the value of `T` is `Null_Task_Id`.

Erroneous Execution

9

If any operation in this package is called with a parameter `T` that specifies a task object that no longer exists, the execution of the program is erroneous.

Implementation Permissions

10

An implementation need not support `Asynchronous_Task_Control` if it is infeasible to support it in the target environment.

NOTES

11

37 It is a consequence of the priority rules that held tasks cannot be dispatched on any processor in a partition (unless they are inheriting priorities) since their priorities are defined to be below the priority of any idle task.

12

38 The effect of calling `Get_Priority` and `Set_Priority` on a Held task is the same as on any other task.

13

39 Calling `Hold` on a held task or `Continue` on a non-held task has no effect.

14

40 The rules affecting queuing are derived from the above rules, in addition to the normal priority rules:

15

- When a held task is on the ready queue, its priority is so low as to never reach the top of the queue as long as there are other tasks on that queue.

16

- If a task is executing in a protected action, inside a rendezvous, or is inheriting priorities from other sources (e.g. when activated), it continues to execute until it is no longer executing the corresponding construct.

17

- If a task becomes held while waiting (as a caller) for a rendezvous to complete, the active priority of the accepting task is not affected.

18/1

- If a task becomes held while waiting in a `selective_accept`, and an entry call is issued to one of the open entries, the corresponding `accept_alternative` (see [S0216], page 379) executes. When the rendezvous completes, the active priority of the accepting task is lowered to the held priority (unless it is still inheriting from other sources), and the task does not execute until another `Continue`.

19

- The same holds if the held task is the only task on a protected entry queue whose barrier becomes open. The corresponding entry body executes.

18.12 D.12 Other Optimizations and Determinism Rules

1

This clause describes various requirements for improving the response and determinism in a real-time system.

Implementation Requirements

2

If the implementation blocks interrupts (see Section 17.3 [C.3], page 951) not as a result of direct user action (e.g. an execution of a protected action) there shall be an upper bound on the duration of this blocking.

3

The implementation shall recognize entry-less protected types. The overhead of acquiring the execution resource of an object of such a type (see Section 10.5.1 [9.5.1], page 344) shall be minimized. In particular, there should not be any overhead due to evaluating `entry_barrier` conditions.

4

`Unchecked_Deallocation` shall be supported for terminated tasks that are designated by access types, and shall have the effect of releasing all the storage associated with the task. This includes any run-time system or heap storage that has been implicitly allocated for the task by the implementation.

Documentation Requirements

5

The implementation shall document the upper bound on the duration of interrupt blocking caused by the implementation. If this is different for different interrupts or interrupt priority levels, it should be documented for each case.

Metrics

6

The implementation shall document the following metric:

7

- The overhead associated with obtaining a mutual–exclusive access to an entry–less protected object. This shall be measured in the following way:

8

For a protected object of the form:

9

```
protected Lock is
  procedure Set;
  function Read return Boolean;
private
  Flag : Boolean := False;
end Lock;
```

10

```
protected body Lock is
  procedure Set is
  begin
    Flag := True;
  end Set;
  function Read return Boolean
  Begin
    return Flag;
  end Read;
end Lock;
```

11

The execution time, in processor clock cycles, of a call to Set. This shall be measured between the point just before issuing the call, and the point just after the call completes. The function Read shall be called later to verify that Set was indeed called (and not optimized away). The calling task shall have sufficiently high priority as to not be preempted during the measurement period. The protected object shall have sufficiently high ceiling priority to allow the task to call Set.

12

For a multiprocessor, if supported, the metric shall be reported for the case where no contention (on the execution resource) exists from tasks executing on other processors.

18.13 D.13 Run-time Profiles

1/2

This clause specifies a mechanism for defining run-time profiles.

Syntax

2/2

The form of a pragma Profile is as follows:

3/2

```
pragma Profile (<profile_>identifier {, <profile_>pragma_argument_association});
```

Legality Rules

4/2

The <profile_>identifier shall be the name of a run-time profile. The semantics of any <profile_>pragma_argument_association (see [S0020], page 44)s are defined by the run-time profile specified by the <profile_>identifier.

Static Semantics

5/2

A profile is equivalent to the set of configuration pragmas that is defined for each run-time profile.

Post-Compilation Rules

6/2

A pragma Profile is a configuration pragma. There may be more than one pragma Profile for a partition.

18.13.1 D.13.1 The Ravenscar Profile

1/2

This clause defines the Ravenscar profile.

Legality Rules

2/2

The <profile_>identifier Ravenscar is a run-time profile. For run-time profile Ravenscar, there shall be no <profile_>pragma_argument_associations.

Static Semantics

3/2

The run-time profile Ravenscar is equivalent to the following set of pragmas:

4/2

```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
```

```

No_Abort_Statements,
No_Dynamic_Attachment,
No_Dynamic_Priorities,
No_Implicit_Heap_Allocations,
No_Local_Protected_Objects,
No_Local_Timing_Events,
No_Protected_Type_Allocators,
No_Relative_Delay,
No_Requeue_Statements,
No_Select_Statements,
No_Specific_Termination_Handlers,
No_Task_Allocators,
No_Task_Hierarchy,
No_Task_Termination,
Simple_Barriers,
Max_Entry_Queue_Length => 1,
Max_Protected_Entries => 1,
Max_Task_Entries => 0,
No_Dependence => Ada.Asynchronous_Task_Control,
No_Dependence => Ada.Calendar,
No_Dependence => Ada.Execution_Time.Group_Budget,
No_Dependence => Ada.Execution_Time.Timers,
No_Dependence => Ada.Task_Attributes);

```

NOTES

5/2

41 The effect of the `Max_Entry_Queue_Length => 1` restriction applies only to protected entry queues due to the accompanying restriction of `Max_Task_Entries => 0`.

18.14 D.14 Execution Time

1/2

This clause describes a language-defined package to measure execution time.

Static Semantics

2/2

The following language-defined library package exists:

3/2

```

with Ada.Task_Identification;
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time is

```

4/2

```

    type CPU_Time is private;

```

```

CPU_Time_First : constant CPU_Time;
CPU_Time_Last  : constant CPU_Time;
CPU_Time_Unit  : constant := <implementation-defined-real-number>;
CPU_Tick      : constant Time_Span;

```

5/2

```

function Clock
  (T : Ada.Task_Identification.Task_Id
   := Ada.Task_Identification.Current_Task)
  return CPU_Time;

```

6/2

```

function "+" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
function "+" (Left : Time_Span; Right : CPU_Time) return CPU_Time;
function "-" (Left : CPU_Time; Right : Time_Span) return CPU_Time;
function "-" (Left : CPU_Time; Right : CPU_Time) return Time_Span;

```

7/2

```

function "<" (Left, Right : CPU_Time) return Boolean;
function "<=" (Left, Right : CPU_Time) return Boolean;
function ">" (Left, Right : CPU_Time) return Boolean;
function ">=" (Left, Right : CPU_Time) return Boolean;

```

8/2

```

procedure Split
  (T : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

```

9/2

```

function Time_Of (SC : Seconds_Count;
                 TS : Time_Span := Time_Span_Zero) return CPU_Time;

```

10/2

```

private
  ... -- <not specified by the language>
end Ada.Execution_Time;

```

11/2

The <execution time> or CPU time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. It is implementation defined which task, if any, is charged the execution time that is consumed by interrupt handlers and run-time services on behalf of the system.

12/2

The type CPU_Time represents the execution time of a task. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers.

13/2

The CPU_Time value I represents the half–open execution–time interval that starts with I*CPU_Time_Unit and is limited by (I+1)*CPU_Time_Unit, where CPU_Time_Unit is an implementation–defined real number. For each task, the execution time value is set to zero at the creation of the task.

14/2

CPU_Time_First and CPU_Time_Last are the smallest and largest values of the CPU_Time type, respectively.

Dynamic Semantics

15/2

CPU_Time_Unit is the smallest amount of execution time representable by the CPU_Time type; it is expressed in seconds. A <CPU clock tick> is an execution time interval during which the clock value (as observed by calling the Clock function) remains constant. CPU_Tick is the average length of such intervals.

16/2

The effects of the operators on CPU_Time and Time_Span are as for the operators defined for integer types.

17/2

The function Clock returns the current execution time of the task identified by T; Tasking_Error is raised if that task has terminated; Program_Error is raised if the value of T is Task_Identification.Null_Task_Id.

18/2

The effects of the Split and Time_Of operations are defined as follows, treating values of type CPU_Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split (T, SC, TS) is to set SC and TS to values such that $T * CPU_Time_Unit = SC * 1.0 + TS * CPU_Time_Unit$, and $0.0 \leq TS * CPU_Time_Unit < 1.0$. The value returned by Time_Of(SC,TS) is the execution–time value T such that $T * CPU_Time_Unit = SC * 1.0 + TS * CPU_Time_Unit$.

Erroneous Execution

19/2

For a call of Clock, if the task identified by T no longer exists, the execution of the program is erroneous.

Implementation Requirements

20/2

The range of CPU_Time values shall be sufficient to uniquely represent the range of execution times from the task start–up to 50 years of execution time later. CPU_Tick shall be no greater than 1 millisecond.

Documentation Requirements

21/2

The implementation shall document the values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick.

22/2

The implementation shall document the properties of the underlying mechanism used to measure execution times, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

Metrics

23/2

The implementation shall document the following metrics:

24/2

- An upper bound on the execution–time duration of a clock tick. This is a value D such that if t_1 and t_2 are any execution times of a given task such that $t_1 < t_2$ and $\text{Clock}t_1 = \text{Clock}t_2$ then $t_2 - t_1 \leq D$.

25/2

- An upper bound on the size of a clock jump. A clock jump is the difference between two successive distinct values of an execution–time clock (as observed by calling the `Clock` function with the same `Task_Id`).

26/2

- An upper bound on the execution time of a call to the `Clock` function, in processor clock cycles.

27/2

- Upper bounds on the execution times of the operators of the type `CPU_Time`, in processor clock cycles.

Implementation Permissions

28/2

Implementations targeted to machines with word size smaller than 32 bits need not support the full range and granularity of the `CPU_Time` type.

Implementation Advice

29/2

When appropriate, implementations should provide configuration mechanisms to change the value of `CPU_Tick`.

18.14.1 D.14.1 Execution Time Timers

1/2

This clause describes a language–defined package that provides a facility for calling a handler when a task has used a defined amount of CPU time.

Static Semantics

2/2

The following language–defined library package exists:

3/2

```
with System;  
package Ada.Execution_Time.Timers is
```


4/2

```
type Timer (T : not null access constant
            Ada.Task_Identification.Task_Id) is
    tagged limited private;
```

5/2

```
type Timer_Handler is
    access protected procedure (TM : in out Timer);
```

6/2

```
Min_Handler_Ceiling : constant System.Any_Priority :=
    <implementation-defined>;
```

7/2

```
procedure Set_Handler (TM      : in out Timer;
                       In_Time : in Time_Span;
                       Handler  : in Timer_Handler);
procedure Set_Handler (TM      : in out Timer;
                       At_Time : in CPU_Time;
                       Handler  : in Timer_Handler);
function Current_Handler (TM : Timer) return Timer_Handler;
procedure Cancel_Handler (TM      : in out Timer;
                          Cancelled : out Boolean);
```

8/2

```
function Time_Remaining (TM : Timer) return Time_Span;
```

9/2

```
Timer_Resource_Error : exception;
```

10/2

```
private
    ... -- not specified by the language
end Ada.Execution_Time.Timers;
```

11/2

The type `Timer` represents an execution-time event for a single task and is capable of detecting execution-time overruns. The access discriminant `T` identifies the task concerned. The type `Timer` needs finalization (see Section 8.6 [7.6], page 295).

12/2

An object of type `Timer` is said to be `<set>` if it is associated with a non-null value of type `Timer_Handler` and `<cleared>` otherwise. All `Timer` objects are initially cleared.

13/2

The type `Timer_Handler` identifies a protected procedure to be executed by the implementation when the timer expires. Such a protected procedure is called a `<handler>`.

Dynamic Semantics

14/2

When a `Timer` object is created, or upon the first call of a `Set_Handler` procedure with the timer as parameter, the resources required to operate an execution–time timer based on the associated execution–time clock are allocated and initialized. If this operation would exceed the available resources, `Timer_Resource_Error` is raised.

15/2

The procedures `Set_Handler` associate the handler `Handler` with the timer `TM`; if `Handler` is null, the timer is cleared, otherwise it is set. The first procedure `Set_Handler` loads the timer `TM` with an interval specified by the `Time_Span` parameter. In this mode, the timer `TM` `<expires>` when the execution time of the task identified by `TM.T.all` has increased by `In_Time`; if `In_Time` is less than or equal to zero, the timer expires immediately. The second procedure `Set_Handler` loads the timer `TM` with the absolute value specified by `At_Time`. In this mode, the timer `TM` expires when the execution time of the task identified by `TM.T.all` reaches `At_Time`; if the value of `At_Time` has already been reached when `Set_Handler` is called, the timer expires immediately.

16/2

A call of a procedure `Set_Handler` for a timer that is already set replaces the handler and the (absolute or relative) execution time; if `Handler` is not null, the timer remains set.

17/2

When a timer expires, the associated handler is executed, passing the timer as parameter. The initial action of the execution of the handler is to clear the event.

18/2

The function `Current_Handler` returns the handler associated with the timer `TM` if that timer is set; otherwise it returns null.

19/2

The procedure `Cancel_Handler` clears the timer if it is set. `Cancelled` is assigned `True` if the timer was set prior to it being cleared; otherwise it is assigned `False`.

20/2

The function `Time_Remaining` returns the execution time interval that remains until the timer `TM` would expire, if that timer is set; otherwise it returns `Time_Span_Zero`.

21/2

The constant `Min_Handler_Ceiling` is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked.

22/2

As part of the finalization of an object of type `Timer`, the timer is cleared.

23/2

For all the subprograms defined in this package, `Tasking_Error` is raised if the task identified by `TM.T.all` has terminated, and `Program_Error` is raised if the value of `TM.T.all` is `Task_Identification.Null_Task_Id`.

24/2

An exception propagated from a handler invoked as part of the expiration of a timer has no effect.

Erroneous Execution

25/2

For a call of any of the subprograms defined in this package, if the task identified by TM.T.all no longer exists, the execution of the program is erroneous.

Implementation Requirements

26/2

For a given Timer object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timer object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler.

27/2

When an object of type Timer is finalized, the system resources used by the timer shall be deallocated.

Implementation Permissions

28/2

Implementations may limit the number of timers that can be defined for each task. If this limit is exceeded then Timer_Resource_Error is raised.

NOTES

29/2

42 A Timer_Handler can be associated with several Timer objects.

18.14.2 D.14.2 Group Execution Time Budgets

1/2

This clause describes a language-defined package to assign execution time budgets to groups of tasks.

Static Semantics

2/2

The following language-defined library package exists:

3/2

```
with System;  
package Ada.Execution_Time.Group_Budgets is
```

4/2

```
type Group_Budget is tagged limited private;
```

5/2

```
type Group_Budget_Handler is access  
protected procedure (GB : in out Group_Budget);
```

6/2

```
type Task_Array is array (Positive range <>) of
    Ada.Task_Identification.Task_Id;
```

7/2

```
Min_Handler_Ceiling : constant System.Any_Priority :=
    <implementation-defined>;
```

8/2

```
procedure Add_Task (GB : in out Group_Budget;
    T : in Ada.Task_Identification.Task_Id);
procedure Remove_Task (GB: in out Group_Budget;
    T : in Ada.Task_Identification.Task_Id);
function Is_Member (GB : Group_Budget;
    T : Ada.Task_Identification.Task_Id) return Boolean;
function Is_A_Group_Member
    (T : Ada.Task_Identification.Task_Id) return Boolean;
function Members (GB : Group_Budget) return Task_Array;
```

9/2

```
procedure Replenish (GB : in out Group_Budget; To : in Time_Span);
procedure Add (GB : in out Group_Budget; Interval : in Time_Span);
function Budget_Has_Expired (GB : Group_Budget) return Boolean;
function Budget_Remaining (GB : Group_Budget) return Time_Span;
```

10/2

```
procedure Set_Handler (GB : in out Group_Budget;
    Handler : in Group_Budget_Handler);
function Current_Handler (GB : Group_Budget)
    return Group_Budget_Handler;
procedure Cancel_Handler (GB : in out Group_Budget;
    Cancelled : out Boolean);
```

11/2

```
Group_Budget_Error : exception;
```

12/2

```
private
    -- not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

13/2

The type Group_Budget represents an execution time budget to be used by a group of

tasks. The type `Group_Budget` needs finalization (see Section 8.6 [7.6], page 295). A task can belong to at most one group. Tasks of any priority can be added to a group.

14/2

An object of type `Group_Budget` has an associated nonnegative value of type `Time_Span` known as its `<budget>`, which is initially `Time_Span_Zero`. The type `Group_Budget_Handler` identifies a protected procedure to be executed by the implementation when the budget is `<exhausted>`, that is, reaches zero. Such a protected procedure is called a `<handler>`.

15/2

An object of type `Group_Budget` also includes a handler, which is a value of type `Group_Budget_Handler`. The handler of the object is said to be `<set>` if it is not null and `<cleared>` otherwise. The handler of all `Group_Budget` objects is initially cleared.

Dynamic Semantics

16/2

The procedure `Add_Task` adds the task identified by `T` to the group `GB`; if that task is already a member of some other group, `Group_Budget_Error` is raised.

17/2

The procedure `Remove_Task` removes the task identified by `T` from the group `GB`; if that task is not a member of the group `GB`, `Group_Budget_Error` is raised. After successful execution of this procedure, the task is no longer a member of any group.

18/2

The function `Is_Member` returns `True` if the task identified by `T` is a member of the group `GB`; otherwise it return `False`.

19/2

The function `Is_A_Group_Member` returns `True` if the task identified by `T` is a member of some group; otherwise it returns `False`.

20/2

The function `Members` returns an array of values of type `Task_Identification.Task_Id` identifying the members of the group `GB`. The order of the components of the array is unspecified.

21/2

The procedure `Replenish` loads the group budget `GB` with `To` as the `Time_Span` value. The exception `Group_Budget_Error` is raised if the `Time_Span` value `To` is non-positive. Any execution of any member of the group of tasks results in the budget counting down, unless exhausted. When the budget becomes exhausted (reaches `Time_Span_Zero`), the associated handler is executed if the handler of group budget `GB` is set. Nevertheless, the tasks continue to execute.

22/2

The procedure `Add` modifies the budget of the group `GB`. A positive value for `Interval` increases the budget. A negative value for `Interval` reduces the budget, but never below `Time_Span_Zero`. A zero value for `Interval` has no effect. A call of procedure `Add` that results in the value of the budget going to `Time_Span_Zero` causes the associated handler to be executed if the handler of the group budget `GB` is set.

23/2

The function `Budget_Has_Expired` returns `True` if the budget of group `GB` is exhausted (equal to `Time_Span_Zero`); otherwise it returns `False`.

24/2

The function `Budget_Remaining` returns the remaining budget for the group GB. If the budget is exhausted it returns `Time_Span_Zero`. This is the minimum value for a budget.

25/2

The procedure `Set_Handler` associates the handler `Handler` with the `Group_Budget` GB; if `Handler` is null, the handler of `Group_Budget` is cleared, otherwise it is set.

26/2

A call of `Set_Handler` for a `Group_Budget` that already has a handler set replaces the handler; if `Handler` is not null, the handler for `Group_Budget` remains set.

27/2

The function `Current_Handler` returns the handler associated with the group budget GB if the handler for that group budget is set; otherwise it returns null.

28/2

The procedure `Cancel_Handler` clears the handler for the group budget if it is set. `Cancelled` is assigned `True` if the handler for the group budget was set prior to it being cleared; otherwise it is assigned `False`.

29/2

The constant `Min_Handler_Ceiling` is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked.

30/2

The precision of the accounting of task execution time to a `Group_Budget` is the same as that defined for execution-time clocks from the parent package.

31/2

As part of the finalization of an object of type `Group_Budget` all member tasks are removed from the group identified by that object.

32/2

If a task is a member of a `Group_Budget` when it terminates then as part of the finalization of the task it is removed from the group.

33/2

For all the operations defined in this package, `Tasking_Error` is raised if the task identified by `T` has terminated, and `Program_Error` is raised if the value of `T` is `Task_Identification.Null_Task_Id`.

34/2

An exception propagated from a handler invoked when the budget of a group of tasks becomes exhausted has no effect.

Erroneous Execution

35/2

For a call of any of the subprograms defined in this package, if the task identified by `T` no longer exists, the execution of the program is erroneous.

Implementation Requirements

36/2

For a given `Group_Budget` object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same

Group_Budget object. The replacement of a handler, by a call of Set_Handler, shall be performed atomically with respect to the execution of the handler.

NOTES

37/2

43 Clearing or setting of the handler of a group budget does not change the current value of the budget. Exhaustion or loading of a budget does not change whether the handler of the group budget is set or cleared.

38/2

44 A Group_Budget_Handler can be associated with several Group_Budget objects.

18.15 D.15 Timing Events

1/2

This clause describes a language-defined package to allow user-defined protected procedures to be executed at a specified time without the need for a task or a delay statement.

Static Semantics

2/2

The following language-defined library package exists:

3/2

```
package Ada.Real_Time.Timing_Events is
```

4/2

```
    type Timing_Event is tagged limited private;
    type Timing_Event_Handler
        is access protected procedure (Event : in out Timing_Event);
```

5/2

```
    procedure Set_Handler (Event   : in out Timing_Event;
                           At_Time  : in Time;
                           Handler  : in Timing_Event_Handler);
    procedure Set_Handler (Event   : in out Timing_Event;
                           In_Time  : in Time_Span;
                           Handler  : in Timing_Event_Handler);
    function Current_Handler (Event : Timing_Event)
        return Timing_Event_Handler;
    procedure Cancel_Handler (Event   : in out Timing_Event;
                              Cancelled : out Boolean);
```

6/2

```
    function Time_Of_Event (Event : Timing_Event) return Time;
```

7/2

```
private
  ... -- <not specified by the language>
end Ada.Real_Time.Timing_Events;
```

8/2

The type `Timing_Event` represents a time in the future when an event is to occur. The type `Timing_Event` needs finalization (see Section 8.6 [7.6], page 295).

9/2

An object of type `Timing_Event` is said to be `<set>` if it is associated with a non-null value of type `Timing_Event_Handler` and `<cleared>` otherwise. All `Timing_Event` objects are initially cleared.

10/2

The type `Timing_Event_Handler` identifies a protected procedure to be executed by the implementation when the timing event occurs. Such a protected procedure is called a `<handler>`.

Dynamic Semantics

11/2

The procedures `Set_Handler` associate the handler `Handler` with the event `Event`; if `Handler` is null, the event is cleared, otherwise it is set. The first procedure `Set_Handler` sets the execution time for the event to be `At_Time`. The second procedure `Set_Handler` sets the execution time for the event to be `Real_Time.Clock + In_Time`.

12/2

A call of a procedure `Set_Handler` for an event that is already set replaces the handler and the time of execution; if `Handler` is not null, the event remains set.

13/2

As soon as possible after the time set for the event, the handler is executed, passing the event as parameter. The handler is only executed if the timing event is in the set state at the time of execution. The initial action of the execution of the handler is to clear the event.

14/2

If the `Ceiling_Locking` policy (see Section 18.3 [D.3], page 991) is in effect when a procedure `Set_Handler` is called, a check is made that the ceiling priority of `Handler.all` is `Interrupt_Priority'Last`. If the check fails, `Program_Error` is raised.

15/2

If a procedure `Set_Handler` is called with zero or negative `In_Time` or with `At_Time` indicating a time in the past then the handler is executed immediately by the task executing the call of `Set_Handler`. The timing event `Event` is cleared.

16/2

The function `Current_Handler` returns the handler associated with the event `Event` if that event is set; otherwise it returns null.

17/2

The procedure `Cancel_Handler` clears the event if it is set. `Cancelled` is assigned `True` if the event was set prior to it being cleared; otherwise it is assigned `False`.

18/2

The function `Time_Of_Event` returns the time of the event if the event is set; otherwise it returns `Real_Time.Time_First`.

19/2

As part of the finalization of an object of type `Timing_Event`, the `Timing_Event` is cleared.

20/2

If several timing events are set for the same time, they are executed in FIFO order of being set.

21/2

An exception propagated from a handler invoked by a timing event has no effect.

Implementation Requirements

22/2

For a given `Timing_Event` object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same `Timing_Event` object. The replacement of a handler by a call of `Set_Handler` shall be performed atomically with respect to the execution of the handler.

Metrics

23/2

The implementation shall document the following metric:

24/2

- An upper bound on the lateness of the execution of a handler. That is, the maximum time between when a handler is actually executed and the time specified when the event was set.

Implementation Advice

25/2

The protected handler procedure should be executed directly by the real-time clock interrupt mechanism.

NOTES

26/2

45 Since a call of `Set_Handler` is not a potentially blocking operation, it can be called from within a handler.

27/2

46 A `Timing_Event_Handler` can be associated with several `Timing_Event` objects.

19 Annex E Distributed Systems

1

This Annex defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program.

Post-Compilation Rules

2

A <distributed system> is an interconnection of one or more <processing nodes> (a system resource that has both computational and storage capabilities), and zero or more <storage nodes> (a system resource that has only storage capabilities, with the storage addressable by one or more processing nodes).

3

A <distributed program> comprises one or more partitions that execute independently (except when they communicate) in a distributed system.

4

The process of mapping the partitions of a program to the nodes in a distributed system is called <configuring the partitions of the program>.

Implementation Requirements

5

The implementation shall provide means for explicitly assigning library units to a partition and for the configuring and execution of a program consisting of multiple partitions on a distributed system; the means are implementation defined.

Implementation Permissions

6

An implementation may require that the set of processing nodes of a distributed system be homogeneous.

NOTES

7

1 The partitions comprising a program may be executed on differently configured distributed systems or on a non-distributed system without requiring recompilation. A distributed program may be partitioned differently from the same set of library units without recompilation. The resulting execution is semantically equivalent.

8

2 A distributed program retains the same type safety as the equivalent single partition program.

19.1 E.1 Partitions

1

The partitions of a distributed program are classified as either active or passive.

Post-Compilation Rules

2

An <active partition> is a partition as defined in Section 11.2 [10.2], page 409. A <passive partition> is a partition that has no thread of control of its own, whose library units are all preelaborated, and whose data and subprograms are accessible to one or more active partitions.

3

A passive partition shall include only library_items that either are declared pure or are shared passive (see Section 11.2.1 [10.2.1], page 413, and Section 19.2.1 [E.2.1], page 1038).

4

An active partition shall be configured on a processing node. A passive partition shall be configured either on a storage node or on a processing node.

5

The configuration of the partitions of a program onto a distributed system shall be consistent with the possibility for data references or calls between the partitions implied by their semantic dependences. Any reference to data or call of a subprogram across partitions is called a <remote access>.

Dynamic Semantics

6

A library_item is elaborated as part of the elaboration of each partition that includes it. If a normal library unit (see Section 19.2 [E.2], page 1036) has state, then a separate copy of the state exists in each active partition that elaborates it. The state evolves independently in each such partition.

7

An active partition <terminates> when its environment task terminates. A partition becomes <inaccessible> if it terminates or if it is <aborted>. An active partition is aborted when its environment task is aborted. In addition, if a partition fails during its elaboration, it becomes inaccessible to other partitions. Other implementation–defined events can also result in a partition becoming inaccessible.

8/1

For a prefix D that denotes a library–level declaration, excepting a declaration of or within a declared–pure library unit, the following attribute is defined:

9

D'Partition_Id

Denotes a value of the type <universal_integer> that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see Section 19.2.3 [E.2.3], page 1041) the given

partition is the one
where the body of D
was elaborated.

Bounded (Run-Time) Errors

10

It is a bounded error for there to be cyclic elaboration dependences between the active partitions of a single distributed program. The possible effects, in each of the partitions involved, are deadlock during elaboration, or the raising of `Communication_Error` or `Program_Error`.

Implementation Permissions

11

An implementation may allow multiple active or passive partitions to be configured on a single processing node, and multiple passive partitions to be configured on a single storage node. In these cases, the scheduling policies, treatment of priorities, and management of shared resources between these partitions are implementation defined.

12

An implementation may allow separate copies of an active partition to be configured on different processing nodes, and to provide appropriate interactions between the copies to present a consistent state of the partition to other active partitions.

13

In an implementation, the partitions of a distributed program need not be loaded and elaborated all at the same time; they may be loaded and elaborated one at a time over an extended period of time. An implementation may provide facilities to abort and reload a partition during the execution of a distributed program.

14

An implementation may allow the state of some of the partitions of a distributed program to persist while other partitions of the program terminate and are later reinvoked.

NOTES

15

3 Library units are grouped into partitions after compile time, but before run time. At compile time, only the relevant library unit properties are identified using categorization pragmas.

16

4 The value returned by the `Partition_Id` attribute can be used as a parameter to implementation-provided subprograms in order to query information about the partition.

19.2 E.2 Categorization of Library Units

1

Library units can be categorized according to the role they play in a distributed program. Certain restrictions are associated with each category to ensure that the semantics of a distributed program remain close to the semantics for a nondistributed program.

2

A <categorization pragma> is a library unit pragma (see Section 11.1.5 [10.1.5], page 407) that restricts the declarations, child units, or semantic dependences of the library unit to which it applies. A <categorized library unit> is a library unit to which a categorization pragma applies.

3

The pragmas `Shared_Passive`, `Remote_Types`, and `Remote_Call_Interface` are categorization pragmas. In addition, for the purposes of this Annex, the pragma `Pure` (see Section 11.2.1 [10.2.1], page 413) is considered a categorization pragma.

4/1

A library package or generic library package is called a <shared passive> library unit if a `Shared_Passive` pragma applies to it. A library package or generic library package is called a <remote types> library unit if a `Remote_Types` pragma applies to it. A library unit is called a <remote call interface> if a `Remote_Call_Interface` pragma applies to it. A <normal library unit> is one to which no categorization pragma applies.

5

The various categories of library units and the associated restrictions are described in this clause and its subclauses. The categories are related hierarchically in that the library units of one category can depend semantically only on library units of that category or an earlier one, except that the body of a remote types or remote call interface library unit is unrestricted.

6

The overall hierarchy (including declared pure) is as follows:

7

Declared Pure

Can depend only on other declared pure library units;

8

Shared Passive

Can depend only on other shared passive or declared pure library units;

9

Remote Types

The declaration of the library unit can depend only on other remote types library units, or one of the above; the body of the library unit is unrestricted;

10

Remote Call Interface

The declaration of the library unit can depend only on other remote call interfaces, or one of the above; the body of the library unit is unrestricted;

11

Normal

Unrestricted.

12

Declared pure and shared passive library units are preelaborated. The declaration of a remote types or remote call interface library unit is required to be preelaborable.

Implementation Requirements

13/1

<This paragraph was deleted.>

Implementation Permissions

14

Implementations are allowed to define other categorization pragmas.

19.2.1 E.2.1 Shared Passive Library Units

1

A shared passive library unit is used for managing global data shared between active partitions. The restrictions on shared passive library units prevent the data or tasks of one active partition from being accessible to another active partition through references implicit in objects declared in the shared passive library unit.

Syntax

2

The form of a pragma Shared_Passive is as follows:

3

```
pragma Shared_Passive[(<library_unit_>name)];
```

Legality Rules

4

A *<shared passive library unit>* is a library unit to which a Shared_Passive pragma applies. The following restrictions apply to such a library unit:

5

- it shall be preelaborable (see Section 11.2.1 [10.2.1], page 413);

6

- it shall depend semantically only upon declared pure or shared passive library units;

7/1

- it shall not contain a library–level declaration of an access type that designates a class–wide type, task type, or protected type with entry_declarations.

8

Notwithstanding the definition of accessibility given in Section 4.10.2 [3.10.2], page 164, the declaration of a library unit P1 is not accessible from within the declarative region of a shared passive library unit P2, unless the shared passive library unit P2 depends semantically on P1.

Static Semantics

9

A shared passive library unit is preelaborated.

Post-Compilation Rules

10

A shared passive library unit shall be assigned to at most one partition within a given program.

11

Notwithstanding the rule given in Section 11.2 [10.2], page 409, a compilation unit in a given partition does not <need> (in the sense of Section 11.2 [10.2], page 409) the shared passive library units on which it depends semantically to be included in that same partition; they will typically reside in separate passive partitions.

19.2.2 E.2.2 Remote Types Library Units

1

A remote types library unit supports the definition of types intended for use in communication between active partitions.

Syntax

2

The form of a pragma Remote_Types is as follows:

3

```
pragma Remote_Types[(<library_unit_>name)];
```

Legality Rules

4

A <remote types library unit> is a library unit to which the pragma Remote_Types applies. The following restrictions apply to the declaration of such a library unit:

5

- it shall be preelaborable;

6

- it shall depend semantically only on declared pure, shared passive, or other remote types library units;

7

- it shall not contain the declaration of any variable within the visible part of the library unit;

8/2

- the full view of each type declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see Section 14.13.2 [13.13.2], page 540).

9/1

An access type declared in the visible part of a remote types or remote call interface library unit is called a <remote access type>. Such a type shall be:

9.1/1

- an access-to-subprogram type, or

9.2/1

- a general access type that designates a class-wide limited private type or a class-wide private type extension all of whose ancestors are either private type extensions or limited private types.

9.3/1

A type that is derived from a remote access type is also a remote access type.

10

The following restrictions apply to the use of a remote access-to-subprogram type:

11/2

- A value of a remote access-to-subprogram type shall be converted only to or from another (subtype-conformant) remote access-to-subprogram type;

12

- The prefix of an Access attribute_reference that yields a value of a remote access-to-subprogram type shall statically denote a (subtype-conformant) remote subprogram.

13

The following restrictions apply to the use of a remote access-to-class-wide type:

14/2

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall support external streaming (see Section 14.13.2 [13.13.2], page 540);

15

- A value of a remote access-to-class-wide type shall be explicitly converted only to another remote access-to-class-wide type;

16/1

- A value of a remote access-to-class-wide type shall be dereferenced (or implicitly converted to an anonymous access type) only as part of a dispatching call where the value designates a controlling operand of the call (see Section 19.4 [E.4], page 1044, "Section 19.4 [E.4], page 1044, Remote Subprogram Calls").

17/2

- The `Storage_Pool` attribute is not defined for a remote access-to-class-wide type; the expected type for an allocator shall not be a remote access-to-class-wide type. A remote access-to-class-wide type shall not be an actual parameter for a generic formal access type. The `Storage_Size` attribute of a remote access-to-class-wide type yields 0; it is not allowed in an `attribute_definition_clause`.

NOTES

18

5 A remote types library unit need not be pure, and the types it defines may include levels of indirection implemented by using access types. User-specified Read and Write attributes (see Section 14.13.2 [13.13.2], page 540) provide for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling any levels of indirection.

19.2.3 E.2.3 Remote Call Interface Library Units

1

A remote call interface library unit can be used as an interface for remote procedure calls (RPCs) (or remote function calls) between active partitions.

Syntax

2

The form of a `pragma Remote_Call_Interface` is as follows:

3

```
pragma Remote_Call_Interface[(<library_unit_>name)];
```

4

The form of a pragma `All_Calls_Remote` is as follows:

5

```
pragma All_Calls_Remote[(<library_unit_>name)];
```

6

A pragma `All_Calls_Remote` is a library unit pragma.

Legality Rules

7/1

A *<remote call interface (RCI)>* is a library unit to which the pragma `Remote_Call_Interface` applies. A subprogram declared in the visible part of such a library unit, or declared by such a library unit, is called a *<remote subprogram>*.

8

The declaration of an RCI library unit shall be preelaborable (see Section 11.2.1 [10.2.1], page 413), and shall depend semantically only upon declared pure, shared passive, remote types, or other remote call interface library units.

9/1

In addition, the following restrictions apply to an RCI library unit:

10/1

- its visible part shall not contain the declaration of a variable;

11/1

- its visible part shall not contain the declaration of a limited type;

12/1

- its visible part shall not contain a nested `generic_declaration`;

13/1

- it shall not be, nor shall its visible part contain, the declaration of a subprogram to which a pragma `Inline` applies;

14/2

- it shall not be, nor shall its visible part contain, a subprogram (or `access-to-subprogram`) declaration whose profile has an `access` parameter or a parameter of a type that does not support external streaming (see Section 14.13.2 [13.13.2], page 540);

15

- any public child of the library unit shall be a remote call interface library unit.

16

If a pragma `All_Calls_Remote` applies to a library unit, the library unit shall be a remote call interface.

Post-Compilation Rules

17

A remote call interface library unit shall be assigned to at most one partition of a given program. A remote call interface library unit whose parent is also an RCI library unit shall be assigned only to the same partition as its parent.

18

Notwithstanding the rule given in Section 11.2 [10.2], page 409, a compilation unit in a given partition that semantically depends on the declaration of an RCI library unit, `<needs>` (in the sense of Section 11.2 [10.2], page 409) only the declaration of the RCI library unit, not the body, to be included in that same partition. Therefore, the body of an RCI library unit is included only in the partition to which the RCI library unit is explicitly assigned.

Implementation Requirements

19/1

If a pragma `All_Calls_Remote` applies to a given RCI library unit, then the implementation shall route any call to a subprogram of the RCI unit from outside the declarative region of the unit through the Partition Communication Subsystem (PCS); see Section 19.5 [E.5], page 1050. Calls to such subprograms from within the declarative region of the unit are defined to be local and shall not go through the PCS.

Implementation Permissions

20

An implementation need not support the `Remote_Call_Interface` pragma nor the `All_Calls_Remote` pragma. Explicit message-based communication between active partitions can be supported as an alternative to RPC.

19.3 E.3 Consistency of a Distributed System

1

This clause defines attributes and rules associated with verifying the consistency of a distributed program.

Static Semantics

2/1

For a prefix `P` that statically denotes a program unit, the following attributes are defined:

3

`P'Version`

Yields a value of the predefined type `String` that identifies the version of the compilation unit

that contains the declaration of the program unit.

4

P'Body_Version

Yields a value of the predefined type String that identifies the version of the compilation unit that contains the body (but not any subunits) of the program unit.

5/1

The <version> of a compilation unit changes whenever the compilation unit changes in a semantically significant way. This International Standard does not define the exact meaning of "semantically significant". It is unspecified whether there are other events (such as recompilation) that result in the version of a compilation unit changing.

5.1/1

If P is not a library unit, and P has no completion, then P'Body_Version returns the Body_Version of the innermost program unit enclosing the declaration of P. If P is a library unit, and P has no completion, then P'Body_Version returns a value that is different from Body_Version of any version of P that has a completion.

Bounded (Run-Time) Errors

6

In a distributed program, a library unit is <consistent> if the same version of its declaration is used throughout. It is a bounded error to elaborate a partition of a distributed program that contains a compilation unit that depends on a different version of the declaration of a shared passive or RCI library unit than that included in the partition to which the shared passive or RCI library unit was assigned. As a result of this error, Program_Error can be raised in one or both partitions during elaboration; in any case, the partitions become inaccessible to one another.

19.4 E.4 Remote Subprogram Calls

1

A <remote subprogram call> is a subprogram call that invokes the execution of a subprogram in another partition. The partition that originates the remote subprogram call is the <calling partition>, and the partition that executes the corresponding subprogram body is the <called partition>. Some remote procedure calls are allowed to return prior to the completion of subprogram execution. These are called <asynchronous remote procedure calls>.

2

There are three different ways of performing a remote subprogram call:

3

- As a direct call on a (remote) subprogram explicitly declared in a remote call interface;

4

- As an indirect call through a value of a remote access-to-subprogram type;

5

- As a dispatching call with a controlling operand designated by a value of a remote access-to-class-wide type.

6

The first way of calling corresponds to a <static> binding between the calling and the called partition. The latter two ways correspond to a <dynamic> binding between the calling and the called partition.

7

A remote call interface library unit (see Section 19.2.3 [E.2.3], page 1041) defines the remote subprograms or remote access types used for remote subprogram calls.

Legality Rules

8

In a dispatching call with two or more controlling operands, if one controlling operand is designated by a value of a remote access-to-class-wide type, then all shall be.

Dynamic Semantics

9

For the execution of a remote subprogram call, subprogram parameters (and later the results, if any) are passed using a stream-oriented representation (see Section 14.13.1 [13.13.1], page 538) which is suitable for transmission between partitions. This action is called <marshalling>. <Unmarshalling> is the reverse action of reconstructing the parameters or results from the stream-oriented representation. Marshalling is performed initially as part of the remote subprogram call in the calling partition; unmarshalling is done in the called partition. After the remote subprogram completes, marshalling is performed in the called partition, and finally unmarshalling is done in the calling partition.

10

A <calling stub> is the sequence of code that replaces the subprogram body of a remotely called subprogram in the calling partition. A <receiving stub> is the sequence of code (the "wrapper") that receives a remote subprogram call on the called partition and invokes the appropriate subprogram body.

11

Remote subprogram calls are executed at most once, that is, if the subprogram call returns normally, then the called subprogram's body was executed exactly once.

12

The task executing a remote subprogram call blocks until the subprogram in the called partition returns, unless the call is asynchronous. For an asynchronous remote procedure call, the calling task can become ready before the procedure in the called partition returns.

13

If a construct containing a remote call is aborted, the remote subprogram call is <cancelled>.

Whether the execution of the remote subprogram is immediately aborted as a result of the cancellation is implementation defined.

14

If a remote subprogram call is received by a called partition before the partition has completed its elaboration, the call is kept pending until the called partition completes its elaboration (unless the call is cancelled by the calling partition prior to that).

15

If an exception is propagated by a remotely called subprogram, and the call is not an asynchronous call, the corresponding exception is reraised at the point of the remote subprogram call. For an asynchronous call, if the remote procedure call returns prior to the completion of the remotely called subprogram, any exception is lost.

16

The exception `Communication_Error` (see Section 19.5 [E.5], page 1050) is raised if a remote call cannot be completed due to difficulties in communicating with the called partition.

17

All forms of remote subprogram calls are potentially blocking operations (see Section 10.5.1 [9.5.1], page 344).

18/1

In a remote subprogram call with a formal parameter of a class-wide type, a check is made that the tag of the actual parameter identifies a tagged type declared in a declared-pure or shared passive library unit, or in the visible part of a remote types or remote call interface library unit. `Program_Error` is raised if this check fails. In a remote function call which returns a class-wide type, the same check is made on the function result.

19

In a dispatching call with two or more controlling operands that are designated by values of a remote access-to-class-wide type, a check is made (in addition to the normal `Tag_Check` -- see Section 12.5 [11.5], page 431) that all the remote access-to-class-wide values originated from `Access attribute_references` that were evaluated by tasks of the same active partition. `Constraint_Error` is raised if this check fails.

Implementation Requirements

20

The implementation of remote subprogram calls shall conform to the PCS interface as defined by the specification of the language-defined package `System.RPC` (see Section 19.5 [E.5], page 1050). The calling stub shall use the `Do_RPC` procedure unless the remote procedure call is asynchronous in which case `Do_APC` shall be used. On the receiving side, the corresponding receiving stub shall be invoked by the `RPC-receiver`.

20.1/1

With respect to shared variables in shared passive library units, the execution of the corresponding subprogram body of a synchronous remote procedure call is considered to be part of the execution of the calling task. The execution of the corresponding subprogram body of an asynchronous remote procedure call proceeds in parallel with the calling task and does not signal the next action of the calling task (see Section 10.10 [9.10], page 389).

NOTES

21

6 A given active partition can both make and receive remote subprogram calls. Thus, an active partition can act as both a client and a server.

22

7 If a given exception is propagated by a remote subprogram call, but the exception does not exist in the calling partition, the exception can be handled by an others choice or be propagated to and handled by a third partition.

19.4.1 E.4.1 Pragma Asynchronous

1

This subclause introduces the pragma Asynchronous which allows a remote subprogram call to return prior to completion of the execution of the corresponding remote subprogram body.

Syntax

2

The form of a pragma Asynchronous is as follows:

3

```
pragma Asynchronous(local_name);  
Legality Rules
```

4

The local_name of a pragma Asynchronous shall denote either:

5

- One or more remote procedures; the formal parameters of the procedure(s) shall all be of mode in;

6

- The first subtype of a remote access-to-procedure type; the formal parameters of the designated profile of the type shall all be of mode in;

7

- The first subtype of a remote access-to-class-wide type.

Static Semantics

8

A pragma Asynchronous is a representation pragma. When applied to a type, it specifies the type-related <asynchronous> aspect of the type.

Dynamic Semantics

9

A remote call is <asynchronous> if it is a call to a procedure, or a call through a value of

an access-to-procedure type, to which a pragma Asynchronous applies. In addition, if a pragma Asynchronous applies to a remote access-to-class-wide type, then a dispatching call on a procedure with a controlling operand designated by a value of the type is asynchronous if the formal parameters of the procedure are all of mode in.

Implementation Requirements

10

Asynchronous remote procedure calls shall be implemented such that the corresponding body executes at most once as a result of the call.

19.4.2 E.4.2 Example of Use of a Remote Access-to-Class-Wide Type

Examples

1

<Example of using a remote access-to-class-wide type to achieve dynamic binding across active partitions:>

2

```
package Tapes is
  pragma Pure(Tapes);
  type Tape is abstract tagged limited private;
  <-- Primitive dispatching operations where>
  <-- Tape is controlling operand>
  procedure Copy (From, To : access Tape; Num_Recs : in Natural) is abstract;
  procedure Rewind (T : access Tape) is abstract;
  <-- More operations>
private
  type Tape is ...
end Tapes;
```

3

```
with Tapes;
package Name_Server is
  pragma Remote_Call_Interface;
  <-- Dynamic binding to remote operations is achieved>
  <-- using the access-to-limited-class-wide type Tape_Ptr>
  type Tape_Ptr is access all Tapes.Tape'Class;
  <-- The following statically bound remote operations>
  <-- allow for a name-server capability in this example>
  function Find (Name : String) return Tape_Ptr;
  procedure Register (Name : in String; T : in Tape_Ptr);
  procedure Remove (T : in Tape_Ptr);
  <-- More operations>
end Name_Server;
```

4


```

package Tape_Driver is
  <-- Declarations are not shown, they are irrelevant here>
end Tape_Driver;

```

5

```

with Tapes, Name_Server;
package body Tape_Driver is
  type New_Tape is new Tapes.Tape with ...
  procedure Copy
    (From, To : access New_Tape; Num_Recs: in Natural) is
  begin
    . . .
  end Copy;
  procedure Rewind (T : access New_Tape) is
  begin
    . . .
  end Rewind;
  <-- Objects remotely accessible through use>
  <-- of Name_Server operations>
  Tape1, Tape2 : aliased New_Tape;
begin
  Name_Server.Register ("NINE-TRACK", Tape1'Access);
  Name_Server.Register ("SEVEN-TRACK", Tape2'Access);
end Tape_Driver;

```

6

```

with Tapes, Name_Server;
<-- Tape_Driver is not needed and thus not mentioned in the with_clause>
procedure Tape_Client is
  T1, T2 : Name_Server.Tape_Ptr;
begin
  T1 := Name_Server.Find ("NINE-TRACK");
  T2 := Name_Server.Find ("SEVEN-TRACK");
  Tapes.Rewind (T1);
  Tapes.Rewind (T2);
  Tapes.Copy (T1, T2, 3);
end Tape_Client;

```

7

<Notes on the example>

8/1

<This paragraph was deleted.>

9

- The package Tapes provides the necessary declarations of the type and its primitive operations.

10

- Name_Server is a remote call interface package and is elaborated in a separate active partition to provide the necessary naming services (such as Register and Find) to the entire distributed program through remote subprogram calls.

11

- Tape_Driver is a normal package that is elaborated in a partition configured on the processing node that is connected to the tape device(s). The abstract operations are overridden to support the locally declared tape devices (Tape1, Tape2). The package is not visible to its clients, but it exports the tape devices (as remote objects) through the services of the Name_Server. This allows for tape devices to be dynamically added, removed or replaced without requiring the modification of the clients' code.

12

- The Tape_Client procedure references only declarations in the Tapes and Name_Server packages. Before using a tape for the first time, it needs to query the Name_Server for a system-wide identity for that tape. From then on, it can use that identity to access the tape device.

13

- Values of remote access type Tape_Ptr include the necessary information to complete the remote dispatching operations that result from dereferencing the controlling operands T1 and T2.

19.5 E.5 Partition Communication Subsystem

1/2

The <Partition Communication Subsystem> (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS.

Static Semantics

2

The following language-defined library package exists:

3

```
with Ada.Streams; <-- see Section 14.13.1 [13.13.1], page 538>
package System.RPC is
```

4

```
    type Partition_Id is range 0 .. <implementation-defined>;
```

5

```
    Communication_Error : exception;
```

6

```
type Params_Stream_Type (  
    Initial_Size : Ada.Streams.Stream_Element_Count) is new  
    Ada.Streams.Root_Stream_Type with private;
```

7

```
procedure Read(  
    Stream : in out Params_Stream_Type;  
    Item : out Ada.Streams.Stream_Element_Array;  
    Last : out Ada.Streams.Stream_Element_Offset);
```

8

```
procedure Write(  
    Stream : in out Params_Stream_Type;  
    Item : in Ada.Streams.Stream_Element_Array);
```

9

```
<-- Synchronous call>  
procedure Do_RPC(  
    Partition : in Partition_Id;  
    Params : access Params_Stream_Type;  
    Result : access Params_Stream_Type);
```

10

```
<-- Asynchronous call>  
procedure Do_APC(  
    Partition : in Partition_Id;  
    Params : access Params_Stream_Type);
```

11

```
<-- The handler for incoming RPCs>  
type RPC_Receiver is access procedure(  
    Params : access Params_Stream_Type;  
    Result : access Params_Stream_Type);
```

12

```
procedure Establish_RPC_Receiver(  
    Partition : in Partition_Id;  
    Receiver : in RPC_Receiver);
```

13

```
private
```

```
... -- <not specified by the language>  
end System.RPC;
```

14

A value of the type `Partition_Id` is used to identify a partition.

15

An object of the type `Params_Stream_Type` is used for identifying the particular remote subprogram that is being called, as well as marshalling and unmarshalling the parameters or result of a remote subprogram call, as part of sending them between partitions.

16

The `Read` and `Write` procedures override the corresponding abstract operations for the type `Params_Stream_Type`.

Dynamic Semantics

17

The `Do_RPC` and `Do_APC` procedures send a message to the active partition identified by the `Partition` parameter.

18

After sending the message, `Do_RPC` blocks the calling task until a reply message comes back from the called partition or some error is detected by the underlying communication system in which case `Communication_Error` is raised at the point of the call to `Do_RPC`.

19

`Do_APC` operates in the same way as `Do_RPC` except that it is allowed to return immediately after sending the message.

20

Upon normal return, the stream designated by the `Result` parameter of `Do_RPC` contains the reply message.

21

The procedure `System.RPC.Establish_RPC_Receiver` is called once, immediately after elaborating the library units of an active partition (that is, right after the <elaboration of the partition>) if the partition includes an RCI library unit, but prior to invoking the main subprogram, if any. The `Partition` parameter is the `Partition_Id` of the active partition being elaborated. The `Receiver` parameter designates an implementation–provided procedure called the <RPC–receiver> which will handle all RPCs received by the partition from the PCS. `Establish_RPC_Receiver` saves a reference to the RPC–receiver; when a message is received at the called partition, the RPC–receiver is called with the `Params` stream containing the message. When the RPC–receiver returns, the contents of the stream designated by `Result` is placed in a message and sent back to the calling partition.

22

If a call on `Do_RPC` is aborted, a cancellation message is sent to the called partition, to request that the execution of the remotely called subprogram be aborted.

23

The subprograms declared in `System.RPC` are potentially blocking operations.

Implementation Requirements

24

The implementation of the RPC–receiver shall be reentrant, thereby allowing concurrent calls on it from the PCS to service concurrent remote subprogram calls into the partition.

24.1/1

An implementation shall not restrict the replacement of the body of System.RPC. An implementation shall not restrict children of System.RPC. The related implementation permissions in the introduction to Annex A do not apply.

24.2/1

If the implementation of System.RPC is provided by the user, an implementation shall support remote subprogram calls as specified.

Documentation Requirements

25

The implementation of the PCS shall document whether the RPC–receiver is invoked from concurrent tasks. If there is an upper limit on the number of such tasks, this limit shall be documented as well, together with the mechanisms to configure it (if this is supported).

Implementation Permissions

26

The PCS is allowed to contain implementation–defined interfaces for explicit message passing, broadcasting, etc. Similarly, it is allowed to provide additional interfaces to query the state of some remote partition (given its partition ID) or of the PCS itself, to set timeouts and retry parameters, to get more detailed error status, etc. These additional interfaces should be provided in child packages of System.RPC.

27

A body for the package System.RPC need not be supplied by the implementation.

27.1/2

An alternative declaration is allowed for package System.RPC as long as it provides a set of operations that is substantially equivalent to the specification defined in this clause.

Implementation Advice

28

Whenever possible, the PCS on the called partition should allow for multiple tasks to call the RPC–receiver with different messages and should allow them to block until the corresponding subprogram body returns.

29

The Write operation on a stream of type Params_Stream_Type should raise Storage_Error if it runs out of space trying to write the Item into the stream.

NOTES

30

8 The package System.RPC is not designed for direct calls by user programs. It is instead designed for use in the implementation of remote subprograms calls, being called by the calling stubs generated for a remote call interface library unit to initiate a remote call, and in turn calling back to an RPC–receiver that dispatches to the receiving stubs generated for the body of a remote call interface, to handle a remote call received from elsewhere.

20 Annex F Information Systems

1

This Annex provides a set of facilities relevant to Information Systems programming. These fall into several categories:

2

- an attribute definition clause specifying `Machine_Radix` for a decimal subtype;

3

- the package `Decimal`, which declares a set of constants defining the implementation's capacity for decimal types, and a generic procedure for decimal division; and

4/2

- the child packages `Text_IO.Editing`, `Wide_Text_IO.Editing`, and `Wide_Wide_Text_IO.Editing`, which support formatted and localized output of decimal data, based on "picture String" values.

5/2

See also: Section 4.5.9 [3.5.9], page 106, "Section 4.5.9 [3.5.9], page 106, Fixed Point Types"; Section 4.5.10 [3.5.10], page 109, "Section 4.5.10 [3.5.10], page 109, Operations of Fixed Point Types"; Section 5.6 [4.6], page 219, "Section 5.6 [4.6], page 219, Type Conversions"; Section 14.3 [13.3], page 486, "Section 14.3 [13.3], page 486, Operational and Representation Attributes"; Section 15.10.9 [A.10.9], page 731, "Section 15.10.9 [A.10.9], page 731, Input-Output for Real Types"; Section 16.3 [B.3], page 901, "Section 16.3 [B.3], page 901, Interfacing with C and C++"; Section 16.4 [B.4], page 931, "Section 16.4 [B.4], page 931, Interfacing with COBOL"; Chapter 21 [Annex G], page 1083, "Chapter 21 [Annex G], page 1083, Numerics".

6

The character and string handling packages in Chapter 15 [Annex A], page 553, "Chapter 15 [Annex A], page 553, Predefined Language Environment" are also relevant for Information Systems.

Implementation Advice

7

If COBOL (respectively, C) is widely supported in the target environment, implementations supporting the Information Systems Annex should provide the child package `Interfaces.COBOL` (respectively, `Interfaces.C`) specified in Chapter 16 [Annex B], page 894, and should support a <convention_>identifier of COBOL (respectively, C) in the interfacing pragmas (see Chapter 16 [Annex B], page 894), thus allowing Ada programs to interface with programs written in that language.

20.1 F.1 `Machine_Radix` Attribute Definition Clause

Static Semantics

1

`Machine_Radix` may be specified for a decimal first subtype (see Section 4.5.9 [3.5.9]),

page 106) via an `attribute_definition_clause`; the expression of such a clause shall be static, and its value shall be 2 or 10. A value of 2 implies a binary base range; a value of 10 implies a decimal base range.

Implementation Advice

2

Packed decimal should be used as the internal representation for objects of subtype S when S'Machine_Radix = 10.

Examples

3

<Example of Machine_Radix attribute definition clause:>

4

```
type Money is delta 0.01 digits 15;
for Money'Machine_Radix use 10;
```

20.2 F.2 The Package Decimal

Static Semantics

1

The library package Decimal has the following declaration:

2

```
package Ada.Decimal is
  pragma Pure(Decimal);
```

3

```
  Max_Scale : constant := <implementation-defined>;
  Min_Scale : constant := <implementation-defined>;
```

4

```
  Min_Delta : constant := 10.0**(-Max_Scale);
  Max_Delta : constant := 10.0**(-Min_Scale);
```

5

```
  Max_Decimal_Digits : constant := <implementation-defined>;
```

6

```
  generic
    type Dividend_Type is delta <> digits <>;
    type Divisor_Type is delta <> digits <>;
    type Quotient_Type is delta <> digits <>;
    type Remainder_Type is delta <> digits <>;
  procedure Divide (Dividend : in Dividend_Type;
                   Divisor : in Divisor_Type;
```

```
        Quotient  : out Quotient_Type;
        Remainder : out Remainder_Type);
pragma Convention(Intrinsic, Divide);
```

7

```
end Ada.Decimal;
```

8

Max_Scale is the largest N such that $10.0^{*(-N)}$ is allowed as a decimal type's delta. Its type is <universal_integer>.

9

Min_Scale is the smallest N such that $10.0^{*(-N)}$ is allowed as a decimal type's delta. Its type is <universal_integer>.

10

Min_Delta is the smallest value allowed for <delta> in a decimal_fixed_point_definition. Its type is <universal_real>.

11

Max_Delta is the largest value allowed for <delta> in a decimal_fixed_point_definition. Its type is <universal_real>.

12

Max_Decimal_Digits is the largest value allowed for <digits> in a decimal_fixed_point_definition. Its type is <universal_integer>.

Static Semantics

13

The effect of Divide is as follows. The value of Quotient is Quotient_Type(Dividend/Divisor). The value of Remainder is Remainder_Type(Intermediate), where Intermediate is the difference between Dividend and the product of Divisor and Quotient; this result is computed exactly.

Implementation Requirements

14

Decimal.Max_Decimal_Digits shall be at least 18.

15

Decimal.Max_Scale shall be at least 18.

16

Decimal.Min_Scale shall be at most 0.

NOTES

17

1 The effect of division yielding a quotient with control over rounding versus truncation is obtained by applying either the function attribute Quotient_Type'Round or the conversion Quotient_Type to the expression Dividend/Divisor.

20.3 F.3 Edited Output for Decimal Types

1/2

The child packages `Text_IO.Editing`, `Wide_Text_IO.Editing`, and `Wide_Wide_Text_IO.Editing` provide localizable formatted text output, known as <edited output>, for decimal types. An edited output string is a function of a numeric value, program–specifiable locale elements, and a format control value. The numeric value is of some decimal type. The locale elements are:

2

- the currency string;

3

- the digits group separator character;

4

- the radix mark character; and

5

- the fill character that replaces leading zeros of the numeric value.

6/2

For `Text_IO.Editing` the edited output and currency strings are of type `String`, and the locale characters are of type `Character`. For `Wide_Text_IO.Editing` their types are `Wide_String` and `Wide_Character`, respectively. For `Wide_Wide_Text_IO.Editing` their types are `Wide_Wide_String` and `Wide_Wide_Character`, respectively.

7

Each of the locale elements has a default value that can be replaced or explicitly overridden.

8

A format–control value is of the private type `Picture`; it determines the composition of the edited output string and controls the form and placement of the sign, the position of the locale elements and the decimal digits, the presence or absence of a radix mark, suppression of leading zeros, and insertion of particular character values.

9

A `Picture` object is composed from a `String` value, known as a <picture String>, that serves as a template for the edited output string, and a `Boolean` value that controls whether a string of all space characters is produced when the number’s value is zero. A picture `String` comprises a sequence of one– or two–`Character` symbols, each serving as a placeholder for a character or string at a corresponding position in the edited output string. The picture `String` symbols fall into several categories based on their effect on the edited output string:

10

Decimal Digit: '9'
Radix Control: '.' 'V'

Sign Control: '+' '-' '<' '>' "CR" "DB"
Currency Control: '\$' '#'
Zero Suppression: 'Z' '*'
Simple Insertion: '_' 'B' '0' '/'

11

The entries are not case-sensitive. Mixed- or lower-case forms for "CR" and "DB", and lower-case forms for 'V', 'Z', and 'B', have the same effect as the upper-case symbols shown.

12

An occurrence of a '9' Character in the picture String represents a decimal digit position in the edited output string.

13

A radix control Character in the picture String indicates the position of the radix mark in the edited output string: an actual character position for '.', or an assumed position for 'V'.

14

A sign control Character in the picture String affects the form of the sign in the edited output string. The '<' and '>' Character values indicate parentheses for negative values. A Character '+', '-', or '<' appears either singly, signifying a fixed-position sign in the edited output, or repeated, signifying a floating-position sign that is preceded by zero or more space characters and that replaces a leading 0.

15

A currency control Character in the picture String indicates an occurrence of the currency string in the edited output string. The '\$' Character represents the complete currency string; the '#' Character represents one character of the currency string. A '\$' Character appears either singly, indicating a fixed-position currency string in the edited output, or repeated, indicating a floating-position currency string that occurs in place of a leading 0. A sequence of '#' Character values indicates either a fixed- or floating-position currency string, depending on context.

16

A zero suppression Character in the picture String allows a leading zero to be replaced by either the space character (for 'Z') or the fill character (for '*').

17

A simple insertion Character in the picture String represents, in general, either itself (if '/' or '0'), the space character (if 'B'), or the digits group separator character (if '_'). In some contexts it is treated as part of a floating sign, floating currency, or zero suppression string.

18/2

An example of a picture String is "<###Z_ZZ9.99>". If the currency string is "kr", the separator character is ',', and the radix mark is '.' then the edited output string values for the decimal values 32.10 and -5432.10 are "bbkrbbb32.10b" and "(bkr5,432.10)", respectively, where 'b' indicates the space character.

19/2

The generic packages Text_IO.Decimal_IO, Wide_Text_IO.Decimal_IO, and Wide_Wide_Text_IO.Decimal_IO (see Section 15.10.9 [A.10.9], page 731, "Section 15.10.9 [A.10.9], page 731, Input-Output for Real Types") provide text input and non-edited text output for decimal types.

NOTES

20/2

2 A picture String is of type Standard.String, for all of Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing.

20.3.1 F.3.1 Picture String Formation

1

A <well-formed picture String>, or simply <picture String>, is a String value that conforms to the syntactic rules, composition constraints, and character replication conventions specified in this clause.

Dynamic Semantics

2/1

<This paragraph was deleted.>

3

```
picture_string ::=
    fixed_$_picture_string
  | fixed_#_picture_string
  | floating_currency_picture_string
  | non_currency_picture_string
```

4

```
fixed_$_picture_string ::=
    [fixed_LHS_sign] fixed_$_char {direct_insertion} [zero_suppression]
    number [RHS_sign]
  | [fixed_LHS_sign {direct_insertion}] [zero_suppression]
    number fixed_$_char {direct_insertion} [RHS_sign]
  | floating_LHS_sign number fixed_$_char {direct_insertion} [RHS_sign]
  | [fixed_LHS_sign] fixed_$_char {direct_insertion}
    all_zero_suppression_number {direct_insertion} [RHS_sign]
  | [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
    fixed_$_char {direct_insertion} [RHS_sign]
  | all_sign_number {direct_insertion} fixed_$_char {direct_insertion} [RHS_sign]
```

5

```
fixed_#_picture_string ::=
  [fixed_LHS_sign] single_#_currency {direct_insertion}
  [zero_suppression] number [RHS_sign]

| [fixed_LHS_sign] multiple_#_currency {direct_insertion}
  zero_suppression number [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] [zero_suppression]
  number fixed_#_currency {direct_insertion} [RHS_sign]

| floating_LHS_sign number fixed_#_currency {direct_insertion} [RHS_sign]

| [fixed_LHS_sign] single_#_currency {direct_insertion}
  all_zero_suppression_number {direct_insertion} [RHS_sign]

| [fixed_LHS_sign] multiple_#_currency {direct_insertion}
  all_zero_suppression_number {direct_insertion} [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
  fixed_#_currency {direct_insertion} [RHS_sign]

| all_sign_number {direct_insertion} fixed_#_currency {direct_insertion} [RHS_sign]
```

6

```
floating_currency_picture_string ::=
  [fixed_LHS_sign] {direct_insertion} floating_#_currency number [RHS_sign]

| [fixed_LHS_sign] {direct_insertion} floating_#_currency number [RHS_sign]

| [fixed_LHS_sign] {direct_insertion} all_currency_number {direct_insertion} [RHS_sign]
```

7

```
non_currency_picture_string ::=
  [fixed_LHS_sign {direct_insertion}] zero_suppression number [RHS_sign]
```

| [floating_LHS_sign] number [RHS_sign]
| [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}

[RHS_sign]
| all_sign_number {direct_insertion}
| fixed_LHS_sign direct_insertion {direct_insertion} number [RHS_sign]

8

fixed_LHS_sign ::= LHS_Sign

9

LHS_Sign ::= + | - | <

10

fixed_\$_char ::= \$

11

direct_insertion ::= simple_insertion

12

simple_insertion ::= - | B | 0 | /

13

zero_suppression ::= Z {Z | context_sensitive_insertion} | fill_string

14

context_sensitive_insertion ::= simple_insertion

15

fill_string ::= * { * | context_sensitive_insertion }

16

number ::=
fore_digits [radix [aft_digits] {direct_insertion}]
| radix aft_digits {direct_insertion}

17

fore_digits ::= 9 {9 | direct_insertion}

18

aft_digits ::= {9 | direct_insertion} 9

19

radix ::= . | V

20

RHS_sign ::= + | - | > | CR | DB

21

floating_LHS_sign ::=
LHS_Sign {context_sensitive_insertion} LHS_Sign {LHS_Sign | context_sensitive_insertion}

22

single_#_currency ::= #

23

multiple_#_currency ::= ## {#}

24

fixed_#_currency ::= single_#_currency | multiple_#_currency

25

floating_\$_currency ::=
\$ {context_sensitive_insertion} \$ {\$ | context_sensitive_insertion}

26

floating_#_currency ::=
{context_sensitive_insertion} # {# | context_sensitive_insertion}

27

all_sign_number ::= all_sign_fore [radix [all_sign_aft]] [>]

28

all_sign_fore ::=
sign_char {context_sensitive_insertion} sign_char {sign_char | context_sensitive_insertion}

29

all_sign_aft ::= {all_sign_aft_char} sign_char

all_sign_aft_char ::= sign_char | context_sensitive_insertion

30

sign_char ::= + | - | <

31

all_currency_number ::= all_currency_fore [radix [all_currency_aft]]

32

all_currency_fore ::=
currency_char {context_sensitive_insertion}
currency_char {currency_char | context_sensitive_insertion}

33

all_currency_aft ::= {all_currency_aft_char} currency_char

all_currency_aft_char ::= currency_char | context_sensitive_insertion ■

34

currency_char ::= \$ | #

35

all_zero_suppression_number ::= all_zero_suppression_fore [radix [all_zero_suppression_aft]]

36

all_zero_suppression_fore ::=
zero_suppression_char {zero_suppression_char | context_sensitive_insertion} ■

37

all_zero_suppression_aft ::= {all_zero_suppression_aft_char} zero_suppression_char

all_zero_suppression_aft_char ::= zero_suppression_char | context_sensitive_insertion

38

zero_suppression_char ::= Z | *

39

The following composition constraints apply to a picture String:

40

- A floating_LHS_sign does not have occurrences of different LHS_Sign Character values.

41

- If a picture String has '<' as fixed_LHS_sign, then it has '>' as RHS_sign.

42

- If a picture String has '<' in a floating_LHS_sign or in an all_sign_number, then it has an occurrence of '>'.

43/1

- If a picture String has '+' or '-' as fixed_LHS_sign, in a floating_LHS_sign, or in an all_sign_number, then it has no RHS_sign or '>' character.

44

- An instance of all_sign_number does not have occurrences of different sign_char Character values.

45

- An instance of all_currency_number does not have occurrences of different currency_char Character values.

46

- An instance of all_zero_suppression_number does not have occurrences of different zero_suppression_char Character values, except for possible case differences between 'Z' and 'z'.

47

A <replicable Character> is a Character that, by the above rules, can occur in two consecutive positions in a picture String.

48

A <Character replication> is a String

49

<char> & ' (' & <spaces> & <count_string> & ') '

50

where <char> is a replicable Character, <spaces> is a String (possibly empty) comprising only space Character values, and <count_string> is a String of one or more decimal digit Character values. A Character replication in a picture String has the same effect as (and is said to be <equivalent to>) a String comprising <n> consecutive occurrences of <char>, where <n>=Integer'Value(<count_string>).

51

An <expanded picture String> is a picture String containing no Character replications.

NOTES

52

3 Although a sign to the left of the number can float, a sign to the right of the number is in a fixed position.

20.3.2 F.3.2 Edited Output Generation

Dynamic Semantics

1

The contents of an edited output string are based on:

2

- A value, Item, of some decimal type Num,

3

- An expanded picture String Pic_String,

4

- A Boolean value, Blank_When_Zero,

5

- A Currency string,

6

- A Fill character,

7

- A Separator character, and

8

- A Radix_Mark character.

9

The combination of a True value for Blank_When_Zero and a '*' character in Pic_String is inconsistent; no edited output string is defined.

10

A layout error is identified in the rules below if leading non-zero digits of Item, character values of the Currency string, or a negative sign would be truncated; in such cases no edited output string is defined.

11

The edited output string has lower bound 1 and upper bound N where $N = \text{Pic_String}'\text{Length} + \text{Currency_Length_Adjustment} - \text{Radix_Adjustment}$, and

12

- $\text{Currency_Length_Adjustment} = \text{Currency}'\text{Length} - 1$ if there is some occurrence of '\$' in Pic_String, and 0 otherwise.

13

- $\text{Radix_Adjustment} = 1$ if there is an occurrence of 'V' or 'v' in Pic_Str, and 0 otherwise.

14

Let the magnitude of Item be expressed as a base-10 number $I_p \dots I_1.F_1 \dots F_q$, called the <displayed> <magnitude> of Item, where:

15

- $q = \text{Min}(\text{Max}(\text{Num}'\text{Scale}, 0), n)$ where n is 0 if Pic_String has no radix and is otherwise the number of digit positions following radix in Pic_String, where a digit position corresponds to an occurrence of '9', a zero_suppression_char (for an all_zero_suppression_number), a currency_char (for an all_currency_number), or a sign_char (for an all_sign_number).

16

- $I_p \neq 0$ if $p > 0$.

17

If $n < \text{Num}'\text{Scale}$, then the above number is the result of rounding (away from 0 if exactly midway between values).

18

If Blank_When_Zero = True and the displayed magnitude of Item is zero, then the edited output string comprises all space character values. Otherwise, the picture String is treated as a sequence of instances of syntactic categories based on the rules in Section 20.3.1 [F.3.1], page 1059, and the edited output string is the concatenation of string values derived from these categories according to the following mapping rules.

19

Table F-1 shows the mapping from a sign control symbol to a corresponding character or string in the edited output. In the columns showing the edited output, a lower-case 'b'

represents the space character. If there is no sign control symbol but the value of Item is negative, a layout error occurs and no edited output string is produced.

Table F–1: Edited Output for Sign Control Symbols

Sign Control Symbol	Edited Output for Non–Negative Number	Edited Output for Negative Number
'+'	'+'	'_'
'_'	'b'	'_'
'<'	'b'	'('
'>'	'b'	')'
"CR"	"bb"	"CR"
"DB"	"bb"	"DB"

20

An instance of `fixed_LHS_sign` maps to a character as shown in Table F–1.

21

An instance of `fixed_$.char` maps to Currency.

22

An instance of `direct_insertion` maps to Separator if `direct_insertion = '_'`, and to the `direct_insertion` Character otherwise.

23

An instance of `number` maps to a string `<integer_part> & <radix_part> & <fraction_part>` where:

24

- The string for `<integer_part>` is obtained as follows:

25

1. Occurrences of '9' in `fore_digits` of number are replaced from right to left with the decimal digit character values for `I1`, ..., `Ip`, respectively.

26

2. Each occurrence of '9' in `fore_digits` to the left of the leftmost '9' replaced according to rule 1 is replaced with '0'.

27

3. If `p` exceeds the number of occurrences of '9' in `fore_digits` of number, then the excess leftmost digits are eligible for use in the mapping of an instance of `zero_suppression`, `floating_LHS_sign`, `floating_$.currency`, or `floating_#.currency` to the left of number; if there is no such instance, then a layout error occurs and no edited output string is produced.

28

- The <radix_part> is:

29

- "" if number does not include a radix, if radix = 'V', or if radix = 'v'

30

- Radix_Mark if number includes '.' as radix

31

- The string for <fraction_part> is obtained as follows:

32

1. Occurrences of '9' in aft_digits of number are replaced from left to right with the decimal digit character values for F1, ... Fq.

33

2. Each occurrence of '9' in aft_digits to the right of the rightmost '9' replaced according to rule 1 is replaced by '0'.

34

An instance of zero_suppression maps to the string obtained as follows:

35

1. The rightmost 'Z', 'z', or '*' Character values are replaced with the excess digits (if any) from the <integer_part> of the mapping of the number to the right of the zero_suppression instance,

36

2. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of some 'Z', 'z', or '*' in zero_suppression that has been mapped to an excess digit,

37

3. Each Character to the left of the leftmost Character replaced according to rule 1 above is replaced by:

38

- the space character if the zero suppression Character is 'Z' or 'z', or

39

- the Fill character if the zero suppression Character is '*'.
40

4. A layout error occurs if some excess digits remain after all 'Z', 'z', and '*' Character values in zero_suppression have been replaced via rule 1; no edited output string is produced.

41

An instance of RHS_sign maps to a character or string as shown in Table F–1.

42

An instance of floating_LHS_sign maps to the string obtained as follows.

43

1. Up to all but one of the rightmost LHS_Sign Character values are replaced by the excess digits (if any) from the <integer_part> of the mapping of the number to the right of the floating_LHS_sign instance.

44

2. The next Character to the left is replaced with the character given by the entry in Table F–1 corresponding to the LHS_Sign Character.

45

3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost LHS_Sign character replaced according to rule 1.

46

4. Any other Character is replaced by the space character..

47

5. A layout error occurs if some excess digits remain after replacement via rule 1; no edited output string is produced.

48

An instance of fixed_#_currency maps to the Currency string with n space character values concatenated on the left (if the instance does not follow a radix) or on the right (if

the instance does follow a radix), where n is the difference between the length of the fixed-#-currency instance and Currency'Length. A layout error occurs if Currency'Length exceeds the length of the fixed-#-currency instance; no edited output string is produced.

49

An instance of floating-\$_currency maps to the string obtained as follows:

50

1. Up to all but one of the rightmost '\$' Character values are replaced with the excess digits (if any) from the <integer_part> of the mapping of the number to the right of the floating-\$_currency instance.

51

2. The next Character to the left is replaced by the Currency string.

52

3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost '\$' Character replaced via rule 1.

53

4. Each other Character is replaced by the space character.

54

5. A layout error occurs if some excess digits remain after replacement by rule 1; no edited output string is produced.

55

An instance of floating-#-currency maps to the string obtained as follows:

56

1. Up to all but one of the rightmost '#' Character values are replaced with the excess digits (if any) from the <integer_part> of the mapping of the number to the right of the floating-#-currency instance.

57

2. The substring whose last Character occurs at the position immediately preceding the leftmost Character replaced via rule 1, and whose length is Currency'Length, is replaced by the Currency string.

58

3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost '#' replaced via rule 1.

59

4. Any other Character is replaced by the space character.

60

5. A layout error occurs if some excess digits remain after replacement rule 1, or if there is no substring with the required length for replacement rule 2; no edited output string is produced.

61

An instance of `all_zero_suppression_number` maps to:

62

- a string of all spaces if the displayed magnitude of Item is zero, the `zero_suppression_char` is 'Z' or 'z', and the instance of `all_zero_suppression_number` does not have a radix at its last character position;

63

- a string containing the Fill character in each position except for the character (if any) corresponding to radix, if `zero_suppression_char = '*'` and the displayed magnitude of Item is zero;

64

- otherwise, the same result as if each `zero_suppression_char` in `all_zero_suppression_aft` were '9', interpreting the instance of `all_zero_suppression_number` as either zero_suppression number (if a radix and `all_zero_suppression_aft` are present), or as zero_suppression otherwise.

65

An instance of `all_sign_number` maps to:

66

- a string of all spaces if the displayed magnitude of Item is zero and the instance of `all_sign_number` does not have a radix at its last character position;

67

- otherwise, the same result as if each `sign_char` in `all_sign_number_aft` were '9', interpreting the instance of `all_sign_number` as either `floating_LHS_sign` number (if a radix and `all_sign_number_aft` are present), or as `floating_LHS_sign` otherwise.

68

An instance of `all_currency_number` maps to:

69

- a string of all spaces if the displayed magnitude of Item is zero and the instance of `all_currency_number` does not have a radix at its last character position;

70

- otherwise, the same result as if each `currency_char` in `all_currency_number_aft` were '9', interpreting the instance of `all_currency_number` as `floating_$.currency` number or `floating_#_currency` number (if a radix and `all_currency_number_aft` are present), or as `floating_$.currency` or `floating_#_currency` otherwise.

Examples

71

In the result string values shown below, 'b' represents the space character.

72

Item: Picture and Result Strings:

73

```
123456.78      Picture:  "-####*_**_**9.99"
               "bbb$***123,456.78"
               "bbFF***123.456,78" (currency = "FF",
                                   separator = '.',',
                                   radix mark = ',')
```

74/1

```
123456.78      Picture:  "-$**_**_**9.99"
               Result:  "b$***123,456.78"
               "bFF***123.456,78" (currency = "FF",
                                   separator = '.',',
                                   radix mark = ',')
```

75

```
0.0            Picture:  "-$$$$$.$$"
               Result:  "bbbbbbbbbb"
```

76

```
0.20           Picture:  "-$$$$$.$$"
               Result:  "bbbbbb$.20"
```

77

```
-1234.565      Picture:  "<<<<_<<<.<<###>"
               Result:  "bb(1,234.57DMb)" (currency = "DM")
```

78

```
12345.67       Picture:  "###_###_##9.99"
               Result:  "bbCHF12,345.67" (currency = "CHF")
```


20.3.3 F.3.3 The Package Text_IO.Editing

1

The package Text_IO.Editing provides a private type Picture with associated operations, and a generic package Decimal_Output. An object of type Picture is composed from a well-formed picture String (see Section 20.3.1 [F.3.1], page 1059) and a Boolean item indicating whether a zero numeric value will result in an edited output string of all space characters. The package Decimal_Output contains edited output subprograms implementing the effects defined in Section 20.3.2 [F.3.2], page 1065.

Static Semantics

2

The library package Text_IO.Editing has the following declaration:

3

```
package Ada.Text_IO.Editing is
```

4

```
    type Picture is private;
```

5

```
    function Valid (Pic_String      : in String;
                   Blank_When_Zero : in Boolean := False) return Boolean;
```

6

```
    function To_Picture (Pic_String      : in String;
                        Blank_When_Zero : in Boolean := False)
        return Picture;
```

7

```
    function Pic_String      (Pic : in Picture) return String;
    function Blank_When_Zero (Pic : in Picture) return Boolean;
```

8

```
    Max_Picture_Length : constant := <implementation_defined>;
```

9

```
    Picture_Error      : exception;
```

10

```
    Default_Currency      : constant String := "$";
    Default_Fill          : constant Character := '*';
    Default_Separator     : constant Character := ',';
    Default_Radix_Mark    : constant Character := '.';
```

11

```
generic
  type Num is delta <> digits <>;
  Default_Currency   : in String   := Text_IO.Editing.Default_Currency;
  Default_Fill       : in Character := Text_IO.Editing.Default_Fill;
  Default_Separator  : in Character :=
    Text_IO.Editing.Default_Separator;
  Default_Radix_Mark : in Character :=
    Text_IO.Editing.Default_Radix_Mark;
```

```
package Decimal_Output is
  function Length (Pic      : in Picture;
                  Currency : in String := Default_Currency)
    return Natural;
```

12

```
function Valid (Item      : in Num;
                Pic       : in Picture;
                Currency   : in String := Default_Currency)
  return Boolean;
```

13

```
function Image (Item      : in Num;
                Pic       : in Picture;
                Currency   : in String := Default_Currency;
                Fill       : in Character := Default_Fill;
                Separator  : in Character := Default_Separator;
                Radix_Mark : in Character := Default_Radix_Mark)
  return String;
```

14

```
procedure Put (File      : in File_Type;
               Item      : in Num;
               Pic       : in Picture;
               Currency   : in String := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);
```

15

```
procedure Put (Item      : in Num;
               Pic       : in Picture;
               Currency   : in String := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator);
```

```

Radix_Mark : in Character := Default_Radix_Mark);■
16

    procedure Put (To          : out String;
                  Item        : in Num;
                  Pic         : in Picture;
                  Currency    : in String := Default_Currency;
                  Fill        : in Character := Default_Fill;
                  Separator   : in Character := Default_Separator;■
                  Radix_Mark  : in Character := Default_Radix_Mark);■
    end Decimal_Output;
private
    ... <-- not specified by the language>
end Ada.Text_IO.Editing;

```

17

The exception `Constraint_Error` is raised if the `Image` function or any of the `Put` procedures is invoked with a null string for `Currency`.

18

```

function Valid (Pic_String      : in String;
               Blank_When_Zero : in Boolean := False) return Boolean;■

```

19

`Valid` returns `True` if `Pic_String` is a well-formed picture String (see Section 20.3.1 [F.3.1], page 1059) the length of whose expansion does not exceed `Max_Picture_Length`, and if either `Blank_When_Zero` is `False` or `Pic_String` contains no `'*`.

20

```

function To_Picture (Pic_String      : in String;
                   Blank_When_Zero : in Boolean := False)
    return Picture;

```

21

`To_Picture` returns a result `Picture` such that the application of the function `Pic_String` to this result yields an expanded picture String equivalent to `Pic_String`, and such that `Blank_When_Zero` applied to the result `Picture` is the same value as the parameter `Blank_When_Zero`. `Picture_Error` is raised if not `Valid(Pic_String, Blank_When_Zero)`.

22

```
function Pic_String      (Pic : in Picture) return String;  
function Blank_When_Zero (Pic : in Picture) return Boolean;
```

23

If Pic is To_Picture(String_Item,
Boolean_Item) for some String_Item and
Boolean_Item, then:

24

- Pic_String(Pic) returns an expanded picture String equivalent to String_Item and with any lower-case letter replaced with its corresponding upper-case form, and

25

- Blank_When_Zero(Pic) returns Boolean_Item.

26

If Pic_1 and Pic_2 are objects of type Picture,
then "="(Pic_1, Pic_2) is True when

27

- Pic_String(Pic_1) = Pic_String(Pic_2),
and

28

- Blank_When_Zero(Pic_1) =
Blank_When_Zero(Pic_2).

29

```
function Length (Pic      : in Picture;  
                Currency : in String := Default_Currency)  
return Natural;
```

30

Length returns Pic_String(Pic)'Length
+ Currency_Length_Adjustment -
Radix_Adjustment where

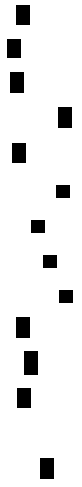
31

- Currency_Length_Adjustment =

32

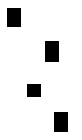
- Currency'Length

—
 1
 if
 there
 is
 some
 oc-
 cur-
 rence
 of
 '\$'
 in
 Pic_String(Pic),
 and



33

- 0
 oth-
 er-
 wise.

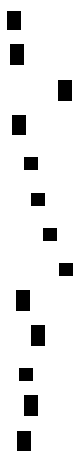


34

- Radix_Adjustment =

35

- 1
 if
 there
 is
 an
 oc-
 cur-
 rence
 of
 'V'
 or
 'v'
 in



36 Pic.Str(Pic),
and

• 0
oth-
er-
wise.

37

```
function Valid (Item      : in Num;  
               Pic       : in Picture;  
               Currency  : in String := Default_Currency)  
  return Boolean;
```

38

Valid returns True if Image(Item, Pic, Currency) does not raise Layout_Error, and returns False otherwise.

39

```
function Image (Item      : in Num;  
               Pic       : in Picture;  
               Currency  : in String := Default_Currency;  
               Fill      : in Character := Default_Fill;  
               Separator : in Character := Default_Separator;  
               Radix_Mark : in Character := Default_Radix_Mark)  
  return String;
```

40

Image returns the edited output String as defined in Section 20.3.2 [F.3.2], page 1065, for Item, Pic.String(Pic), Blank_When_Zero(Pic), Currency, Fill, Separator, and Radix_Mark. If these rules identify a layout error, then Image raises the exception Layout_Error.

41

```
procedure Put (File      : in File_Type;  
              Item      : in Num;  
              Pic       : in Picture;  
              Currency  : in String := Default_Currency;  
              Fill      : in Character := Default_Fill;
```

```

        Separator : in Character := Default_Separator;
        Radix_Mark : in Character := Default_Radix_Mark);

procedure Put (Item      : in Num;
              Pic        : in Picture;
              Currency   : in String := Default_Currency;
              Fill       : in Character := Default_Fill;
              Separator   : in Character := Default_Separator;
              Radix_Mark : in Character := Default_Radix_Mark);

```

42

Each of these Put procedures outputs Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) consistent with the conventions for Put for other real types in case of bounded line length (see Section 15.10.6 [A.10.6], page 721, "Section 15.10.6 [A.10.6], page 721, Get and Put Procedures").

43

```

procedure Put (To          : out String;
              Item        : in Num;
              Pic         : in Picture;
              Currency     : in String := Default_Currency;
              Fill         : in Character := Default_Fill;
              Separator    : in Character := Default_Separator;
              Radix_Mark  : in Character := Default_Radix_Mark);

```

44

Put copies Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) to the given string, right justified. Otherwise unassigned Character values in To are assigned the space character. If To'Length is less than the length of the string resulting from Image, then Layout_Error is raised.

Implementation Requirements

45

Max_Picture_Length shall be at least 30. The implementation shall support currency strings of length up to at least 10, both for Default_Currency in an instantiation of Decimal_Output, and for Currency in an invocation of Image or any of the Put procedures.

NOTES

46

4 The rules for edited output are based on COBOL (ANSI X3.23:1985, endorsed by ISO as ISO 1989–1985), with the following differences:

47

- The COBOL provisions for picture string localization and for 'P' format are absent from Ada.

48

- The following Ada facilities are not in COBOL:

49

- currency symbol placement after the number,

50

- localization of edited output string for multi-character currency string values, including support for both length-preserving and length-expanding currency symbols in picture strings

51

- localization of the radix mark, digits separator, and fill character, and

52

- parenthesization of negative values.

52.1

The value of 30 for Max_Picture_Length is the same limit as in COBOL.

20.3.4 F.3.4 The Package Wide_Text_IO.Editing

Static Semantics

1

The child package Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that:

2

- each occurrence of Character is replaced by Wide_Character,

3

- each occurrence of Text_IO is replaced by Wide_Text_IO,

4

- the subtype of Default_Currency is Wide_String rather than String, and

5

- each occurrence of String in the generic package Decimal_Output is replaced by Wide_String.

NOTES

6

5 Each of the functions Wide_Text_IO.Editing.Valid, To_Picture, and Pic_String has String (versus Wide_String) as its parameter or result subtype, since a picture String is not localizable.

20.3.5 F.3.5 The Package Wide_Wide_Text_IO.Editing

Static Semantics

1/2

The child package Wide_Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that:

2/2

- each occurrence of Character is replaced by Wide_Wide_Character,

3/2

- each occurrence of Text_IO is replaced by Wide_Wide_Text_IO,

4/2

- the subtype of Default_Currency is Wide_Wide_String rather than String, and

5/2

- each occurrence of `String` in the generic package `Decimal_Output` is replaced by `Wide_Wide_String`.

NOTES

6/2

6 Each of the functions `Wide_Wide_Text_IO.Editing.Valid`, `To_Picture`, and `Pic_String` has `String` (versus `Wide_Wide_String`) as its parameter or result subtype, since a picture `String` is not localizable.

21 Annex G Numerics

1

The Numerics Annex specifies

2

- features for complex arithmetic, including complex I/O;

3

- a mode ("strict mode"), in which the predefined arithmetic operations of floating point and fixed point types and the functions and operations of various predefined packages have to provide guaranteed accuracy or conform to other numeric performance requirements, which the Numerics Annex also specifies;

4

- a mode ("relaxed mode"), in which no accuracy or other numeric performance requirements need be satisfied, as for implementations not conforming to the Numerics Annex;

5/2

- models of floating point and fixed point arithmetic on which the accuracy requirements of strict mode are based;

6/2

- the definitions of the model-oriented attributes of floating point types that apply in the strict mode; and

6.1/2

- features for the manipulation of real and complex vectors and matrices.

Implementation Advice

7

If Fortran (respectively, C) is widely supported in the target environment, implementations supporting the Numerics Annex should provide the child package Interfaces.Fortran (respectively, Interfaces.C) specified in Chapter 16 [Annex B], page 894, and should support a <convention_>identifier of Fortran (respectively, C) in the interfacing pragmas (see Chapter 16 [Annex B], page 894), thus allowing Ada programs to interface with programs written in that language.

21.1 G.1 Complex Arithmetic

1

Types and arithmetic operations for complex arithmetic are provided in `Generic_Complex_Types`, which is defined in Section 21.1.1 [G.1.1], page 1084. Implementation-defined approximations to the complex analogs of the mathematical functions known as the "elementary functions" are provided by the subprograms in `Generic_Complex_Elementary_Functions`, which is defined in Section 21.1.2 [G.1.2], page 1091. Both of these library units are generic children of the predefined package `Numerics` (see Section 15.5 [A.5], page 648). Nongeneric equivalents of these generic packages for each of the predefined floating point types are also provided as children of `Numerics`.

21.1.1 G.1.1 Complex Types

Static Semantics

1

The generic library package `Numerics.Generic_Complex_Types` has the following declaration:

2/1

```
generic
  type Real is digits <>;
package Ada.Numerics.Generic_Complex_Types is
  pragma Pure(Generic_Complex_Types);
```

3

```
type Complex is
  record
    Re, Im : Real'Base;
  end record;
```

4/2

```
type Imaginary is private;
pragma Preelaborable_Initialization(Imaginary);
```

5

```
i : constant Imaginary;
j : constant Imaginary;
```

6

```
function Re (X : Complex) return Real'Base;
function Im (X : Complex) return Real'Base;
function Im (X : Imaginary) return Real'Base;
```

7

```
procedure Set_Re (X : in out Complex;
                 Re : in      Real'Base);
procedure Set_Im (X : in out Complex;
                 Im : in      Real'Base);
procedure Set_Im (X :      out Imaginary;
                 Im : in      Real'Base);
```

8

```
function Compose_From_Cartesian (Re, Im : Real'Base) return Complex;█
function Compose_From_Cartesian (Re      : Real'Base) return Complex;█
function Compose_From_Cartesian (Im      : Imaginary) return Complex;█
```

9

```
function Modulus (X      : Complex) return Real'Base;
function "abs"   (Right : Complex) return Real'Base renames Modulus;█
```

10

```
function Argument (X      : Complex) return Real'Base;
function Argument (X      : Complex;
                  Cycle : Real'Base) return Real'Base;
```

11

```
function Compose_From_Polar (Modulus, Argument      : Real'Base)█
    return Complex;
function Compose_From_Polar (Modulus, Argument, Cycle : Real'Base)█
    return Complex;
```

12

```
function "+"      (Right : Complex) return Complex;
function "-"      (Right : Complex) return Complex;
function Conjugate (X      : Complex) return Complex;
```

13

```
function "+" (Left, Right : Complex) return Complex;
function "-" (Left, Right : Complex) return Complex;
function "*" (Left, Right : Complex) return Complex;
function "/" (Left, Right : Complex) return Complex;
```

14

```
function "**" (Left : Complex; Right : Integer) return Complex;
```

15

```
function "+"      (Right : Imaginary) return Imaginary;
```

```
function "-" (Right : Imaginary) return Imaginary;
function Conjugate (X : Imaginary) return Imaginary renames "-";
function "abs" (Right : Imaginary) return Real'Base;
```

16

```
function "+" (Left, Right : Imaginary) return Imaginary;
function "-" (Left, Right : Imaginary) return Imaginary;
function "*" (Left, Right : Imaginary) return Real'Base;
function "/" (Left, Right : Imaginary) return Real'Base;
```

17

```
function "**" (Left : Imaginary; Right : Integer) return Complex;
```

18

```
function "<" (Left, Right : Imaginary) return Boolean;
function "<=" (Left, Right : Imaginary) return Boolean;
function ">" (Left, Right : Imaginary) return Boolean;
function ">=" (Left, Right : Imaginary) return Boolean;
```

19

```
function "+" (Left : Complex; Right : Real'Base) return Complex;
function "+" (Left : Real'Base; Right : Complex) return Complex;
function "-" (Left : Complex; Right : Real'Base) return Complex;
function "-" (Left : Real'Base; Right : Complex) return Complex;
function "*" (Left : Complex; Right : Real'Base) return Complex;
function "*" (Left : Real'Base; Right : Complex) return Complex;
function "/" (Left : Complex; Right : Real'Base) return Complex;
function "/" (Left : Real'Base; Right : Complex) return Complex;
```

20

```
function "+" (Left : Complex; Right : Imaginary) return Complex;
function "+" (Left : Imaginary; Right : Complex) return Complex;
function "-" (Left : Complex; Right : Imaginary) return Complex;
function "-" (Left : Imaginary; Right : Complex) return Complex;
function "*" (Left : Complex; Right : Imaginary) return Complex;
function "*" (Left : Imaginary; Right : Complex) return Complex;
function "/" (Left : Complex; Right : Imaginary) return Complex;
function "/" (Left : Imaginary; Right : Complex) return Complex;
```

21

```
function "+" (Left : Imaginary; Right : Real'Base) return Complex;
function "+" (Left : Real'Base; Right : Imaginary) return Complex;
function "-" (Left : Imaginary; Right : Real'Base) return Complex;
function "-" (Left : Real'Base; Right : Imaginary) return Complex;
```

```

function "*" (Left : Imaginary; Right : Real'Base) return Imaginary;
function "*" (Left : Real'Base; Right : Imaginary) return Imaginary;
function "/" (Left : Imaginary; Right : Real'Base) return Imaginary;
function "/" (Left : Real'Base; Right : Imaginary) return Imaginary;

```

22

```

private

```

23

```

type Imaginary is new Real'Base;
i : constant Imaginary := 1.0;
j : constant Imaginary := 1.0;

```

24

```

end Ada.Numerics.Generic_Complex_Types;

```

25/1

The library package `Numerics.Complex_Types` is declared pure and defines the same types, constants, and subprograms as `Numerics.Generic_Complex_Types`, except that the predefined type `Float` is systematically substituted for `Real'Base` throughout. Nongeneric equivalents of `Numerics.Generic_Complex_Types` for each of the other predefined floating point types are defined similarly, with the names `Numerics.Short_Complex_Types`, `Numerics.Long_Complex_Types`, etc.

26/2

`Complex` is a visible type with Cartesian components.

27

`Imaginary` is a private type; its full type is derived from `Real'Base`.

28

The arithmetic operations and the `Re`, `Im`, `Modulus`, `Argument`, and `Conjugate` functions have their usual mathematical meanings. When applied to a parameter of pure-imaginary type, the "imaginary-part" function `Im` yields the value of its parameter, as the corresponding real value. The remaining subprograms have the following meanings:

29

- The `Set_Re` and `Set_Im` procedures replace the designated component of a complex parameter with the given real value; applied to a parameter of pure-imaginary type, the `Set_Im` procedure replaces the value of that parameter with the imaginary value corresponding to the given real value.

30

- The `Compose_From_Cartesian` function constructs a complex value from the given real and imaginary components. If only one component is given, the other component is implicitly zero.

31

- The `Compose_From_Polar` function constructs a complex value from the given modulus (radius) and argument (angle). When the value of the parameter `Modulus` is positive (resp., negative), the result is the complex value represented by the point in the complex plane lying at a distance from the origin given by the absolute value of `Modulus` and forming an angle measured counterclockwise from the positive (resp., negative) real axis given by the value of the parameter `Argument`.

32

When the `Cycle` parameter is specified, the result of the `Argument` function and the parameter `Argument` of the `Compose_From_Polar` function are measured in units such that a full cycle of revolution has the given value; otherwise, they are measured in radians.

33

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

34

- The result of the `Modulus` function is nonnegative.

35

- The result of the `Argument` function is in the quadrant containing the point in the complex plane represented by the parameter `X`. This may be any quadrant (I through IV); thus, the range of the `Argument` function is approximately $-\text{PI}$ to PI ($-\text{Cycle}/2.0$ to $\text{Cycle}/2.0$, if the parameter `Cycle` is specified). When the point represented by the parameter `X` lies on the negative real axis, the result approximates

36

- PI (resp., $-\text{PI}$) when the sign of the imaginary component of `X` is positive (resp., negative), if `Real'Signed_Zeros` is `True`;

37

- PI , if `Real'Signed_Zeros` is `False`.

38

- Because a result lying on or near one of the axes may not be exactly representable, the approximation inherent in computing the result may place it in an adjacent quadrant, close to but on the wrong side of the axis.

Dynamic Semantics

39

The exception `Numerics.Argument_Error` is raised by the `Argument` and `Compose_From_Polar` functions with specified cycle, signaling a parameter value outside the

domain of the corresponding mathematical function, when the value of the parameter `Cycle` is zero or negative.

40

The exception `Constraint_Error` is raised by the division operator when the value of the right operand is zero, and by the exponentiation operator when the value of the left operand is zero and the value of the exponent is negative, provided that `Real'Machine_Overflows` is `True`; when `Real'Machine_Overflows` is `False`, the result is unspecified. `Constraint_Error` can also be raised when a finite result overflows (see Section 21.2.6 [G.2.6], page 1116).

Implementation Requirements

41

In the implementation of `Numerics.Generic_Complex_Types`, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype `Real`.

42

In the following cases, evaluation of a complex arithmetic operation shall yield the <prescribed result>, provided that the preceding rules do not call for an exception to be raised:

43

- The results of the `Re`, `Im`, and `Compose_From_Cartesian` functions are exact.

44

- The real (resp., imaginary) component of the result of a binary addition operator that yields a result of complex type is exact when either of its operands is of pure-imaginary (resp., real) type.

45

- The real (resp., imaginary) component of the result of a binary subtraction operator that yields a result of complex type is exact when its right operand is of pure-imaginary (resp., real) type.

46

- The real component of the result of the `Conjugate` function for the complex type is exact.

47

- When the point in the complex plane represented by the parameter `X` lies on the nonnegative real axis, the `Argument` function yields a result of zero.

48

- When the value of the parameter `Modulus` is zero, the `Compose_From_Polar` function yields a result of zero.

49

- When the value of the parameter `Argument` is equal to a multiple of the quarter cycle, the result of the `Compose_From_Polar` function with specified cycle lies on one of the axes. In this case, one of its components is zero, and the other has the magnitude of the parameter `Modulus`.

50

- Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand. Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero, provided that the exponent is nonzero. When the left operand is of pure–imaginary type, one component of the result of the exponentiation operator is zero.

51

When the result, or a result component, of any operator of `Numerics.Generic_Complex_Types` has a mathematical definition in terms of a single arithmetic or relational operation, that result or result component exhibits the accuracy of the corresponding operation of the type `Real`.

52

Other accuracy requirements for the `Modulus`, `Argument`, and `Compose_From_Polar` functions, and accuracy requirements for the multiplication of a pair of complex operands or for division by a complex operand, all of which apply only in the strict mode, are given in Section 21.2.6 [G.2.6], page 1116.

53

The sign of a zero result or zero result component yielded by a complex arithmetic operation or function is implementation defined when `Real.Signed_Zeros` is `True`.

Implementation Permissions

54

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

55/2

Implementations may obtain the result of exponentiation of a complex or pure–imaginary operand by repeated complex multiplication, with arbitrary association of the factors and with a possible final complex reciprocation (when the exponent is negative). Implementations are also permitted to obtain the result of exponentiation of a complex operand, but not of a pure–imaginary operand, by converting the left operand to a polar representation; exponentiating the modulus by the given exponent; multiplying the argument by the given exponent; and reconvertng to a Cartesian representation. Because of this implementation freedom, no accuracy requirement is imposed on complex exponentiation (except for the prescribed results given above, which apply regardless of the implementation method chosen).

Implementation Advice

56

Because the usual mathematical meaning of multiplication of a complex operand and a

real operand is that of the scaling of both components of the former by the latter, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex multiplication. In systems that, in the future, support an Ada binding to IEC 559:1989, the latter technique will not generate the required result when one of the components of the complex operand is infinite. (Explicit multiplication of the infinite component by the zero component obtained during promotion yields a NaN that propagates into the final result.) Analogous advice applies in the case of multiplication of a complex operand and a pure-imaginary operand, and in the case of division of a complex operand by a real or pure-imaginary operand.

57

Likewise, because the usual mathematical meaning of addition of a complex operand and a real operand is that the imaginary operand remains unchanged, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex addition. In implementations in which the `Signed_Zeros` attribute of the component type is `True` (and which therefore conform to IEC 559:1989 in regard to the handling of the sign of zero in predefined arithmetic operations), the latter technique will not generate the required result when the imaginary component of the complex operand is a negatively signed zero. (Explicit addition of the negative zero to the zero obtained during promotion yields a positive zero.) Analogous advice applies in the case of addition of a complex operand and a pure-imaginary operand, and in the case of subtraction of a complex operand and a real or pure-imaginary operand.

58

Implementations in which `Real'Signed_Zeros` is `True` should attempt to provide a rational treatment of the signs of zero results and result components. As one example, the result of the `Argument` function should have the sign of the imaginary component of the parameter `X` when the point represented by that parameter lies on the positive real axis; as another, the sign of the imaginary component of the `Compose_From_Polar` function should be the same as (resp., the opposite of) that of the `Argument` parameter when that parameter has a value of zero and the `Modulus` parameter has a nonnegative (resp., negative) value.

21.1.2 G.1.2 Complex Elementary Functions

Static Semantics

1

The generic library package `Numerics.Generic_Complex_Elementary_Functions` has the following declaration:

2/2

```
with Ada.Numerics.Generic_Complex_Types;
generic
  with package Complex_Types is
    new Ada.Numerics.Generic_Complex_Types (<>);
  use Complex_Types;
package Ada.Numerics.Generic_Complex_Elementary_Functions is
  pragma Pure(Generic_Complex_Elementary_Functions);
```

3

```

function Sqrt (X : Complex)    return Complex;
function Log   (X : Complex)    return Complex;
function Exp   (X : Complex)    return Complex;
function Exp   (X : Imaginary)  return Complex;
function "**"   (Left : Complex; Right : Complex)  return Complex;
function "**"   (Left : Complex; Right : Real'Base) return Complex;
function "**"   (Left : Real'Base; Right : Complex) return Complex;

```

4

```

function Sin (X : Complex) return Complex;
function Cos (X : Complex) return Complex;
function Tan (X : Complex) return Complex;
function Cot (X : Complex) return Complex;

```

5

```

function Arcsin (X : Complex) return Complex;
function Arccos (X : Complex) return Complex;
function Arctan (X : Complex) return Complex;
function Arccot (X : Complex) return Complex;

```

6

```

function Sinh (X : Complex) return Complex;
function Cosh (X : Complex) return Complex;
function Tanh (X : Complex) return Complex;
function Coth (X : Complex) return Complex;

```

7

```

function Arcsinh (X : Complex) return Complex;
function Arccosh (X : Complex) return Complex;
function Arctanh (X : Complex) return Complex;
function Arccoth (X : Complex) return Complex;

```

8

```

end Ada.Numerics.Generic_Complex_Elementary_Functions;

```

9/1

The library package `Numerics.Complex_Elementary_Functions` is declared pure and defines the same subprograms as `Numerics.Generic_Complex_Elementary_Functions`, except that the predefined type `Float` is systematically substituted for `Real'Base`, and the `Complex` and `Imaginary` types exported by `Numerics.Complex.Types` are systematically substituted for `Complex` and `Imaginary`, throughout. Nongeneric equivalents of `Numerics.Generic_Complex_Elementary_Functions` corresponding to each of the other predefined floating point types are defined similarly, with the names `Numerics.Short_Complex_Elementary_Functions`, `Numerics.Long_Complex_Elementary_Functions`, etc.

10

The overloading of the `Exp` function for the pure-imaginary type is provided to give the user an alternate way to compose a complex value from a given modulus and argument. In addition to `Compose_From_Polar(Rho, Theta)` (see Section 21.1.1 [G.1.1], page 1084), the programmer may write `Rho * Exp(i * Theta)`.

11

The imaginary (resp., real) component of the parameter `X` of the forward hyperbolic (resp., trigonometric) functions and of the `Exp` function (and the parameter `X`, itself, in the case of the overloading of the `Exp` function for the pure-imaginary type) represents an angle measured in radians, as does the imaginary (resp., real) component of the result of the `Log` and inverse hyperbolic (resp., trigonometric) functions.

12

The functions have their usual mathematical meanings. However, the arbitrariness inherent in the placement of branch cuts, across which some of the complex elementary functions exhibit discontinuities, is eliminated by the following conventions:

13

- The imaginary component of the result of the `Sqrt` and `Log` functions is discontinuous as the parameter `X` crosses the negative real axis.

14

- The result of the exponentiation operator when the left operand is of complex type is discontinuous as that operand crosses the negative real axis.

15/2

- The imaginary component of the result of the `Arcsin`, `Arccos`, and `Arctanh` functions is discontinuous as the parameter `X` crosses the real axis to the left of -1.0 or the right of 1.0 .

16/2

- The real component of the result of the `Arctan` and `Arcsinh` functions is discontinuous as the parameter `X` crosses the imaginary axis below $-<i>$ or above $<i>$.

17/2

- The real component of the result of the `Arccot` function is discontinuous as the parameter `X` crosses the imaginary axis below $-<i>$ or above $<i>$.

18

- The imaginary component of the `Arccosh` function is discontinuous as the parameter `X` crosses the real axis to the left of 1.0 .

19

- The imaginary component of the result of the `Arccoth` function is discontinuous as the parameter `X` crosses the real axis between -1.0 and 1.0 .

20/2

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply that the principal branch is an analytic continuation of the corresponding real-valued function in `Numerics.Generic_Elementary_Functions`. (For `Arctan` and `Arccot`, the single-argument function in question is that obtained from the two-argument version by fixing the second argument to be its default value.)

21

- The real component of the result of the `Sqrt` and `Arccosh` functions is nonnegative.

22

- The same convention applies to the imaginary component of the result of the `Log` function as applies to the result of the natural-cycle version of the `Argument` function of `Numerics.Generic_Complex_Types` (see Section 21.1.1 [G.1.1], page 1084).

23

- The range of the real (resp., imaginary) component of the result of the `Arcsin` and `Arctan` (resp., `Arcsinh` and `Arctanh`) functions is approximately $-\text{PI}/2.0$ to $\text{PI}/2.0$.

24

- The real (resp., imaginary) component of the result of the `Arccos` and `Arccot` (resp., `Arccoth`) functions ranges from 0.0 to approximately PI .

25

- The range of the imaginary component of the result of the `Arccosh` function is approximately $-\text{PI}$ to PI .

26

In addition, the exponentiation operator inherits the single-valuedness of the `Log` function.

Dynamic Semantics

27

The exception `Numerics.Argument_Error` is raised by the exponentiation operator, signaling a parameter value outside the domain of the corresponding mathematical function, when the value of the left operand is zero and the real component of the exponent (or the exponent itself, when it is of real type) is zero.

28

The exception `Constraint_Error` is raised, signaling a pole of the mathematical

function (analogous to dividing by zero), in the following cases, provided that `Complex.Types.Real'Machine.Overflows` is `True`:

29

- by the `Log`, `Cot`, and `Coth` functions, when the value of the parameter `X` is zero;

30

- by the exponentiation operator, when the value of the left operand is zero and the real component of the exponent (or the exponent itself, when it is of real type) is negative;

31

- by the `Arctan` and `Arccot` functions, when the value of the parameter `X` is $\pm <i>$;

32

- by the `Arctanh` and `Arccoth` functions, when the value of the parameter `X` is ± 1.0 .

33

`Constraint_Error` can also be raised when a finite result overflows (see Section 21.2.6 [G.2.6], page 1116); this may occur for parameter values sufficiently `<near>` poles, and, in the case of some of the functions, for parameter values having components of sufficiently large magnitude. When `Complex.Types.Real'Machine.Overflows` is `False`, the result at poles is unspecified.

Implementation Requirements

34

In the implementation of `Numerics.Generic.Complex.Elementary.Functions`, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype `Complex.Types.Real`.

35

In the following cases, evaluation of a complex elementary function shall yield the `<prescribed result>` (or a result having the prescribed component), provided that the preceding rules do not call for an exception to be raised:

36

- When the parameter `X` has the value zero, the `Sqrt`, `Sin`, `Arcsin`, `Tan`, `Arctan`, `Sinh`, `Arcsinh`, `Tanh`, and `Arctanh` functions yield a result of zero; the `Exp`, `Cos`, and `Cosh` functions yield a result of one; the `Arccos` and `Arccot` functions yield a real result; and the `Arccoth` function yields an imaginary result.

37

- When the parameter `X` has the value one, the `Sqrt` function yields a result of one; the `Log`, `Arccos`, and `Arccosh` functions yield a result of zero; and the `Arcsin` function yields a real result.

38

- When the parameter X has the value -1.0 , the `Sqrt` function yields the result

39

- $\langle i \rangle$ (resp., $-\langle i \rangle$), when the sign of the imaginary component of X is positive (resp., negative), if `Complex.Types.Real'Signed_Zeros` is `True`;

40

- $\langle i \rangle$, if `Complex.Types.Real'Signed_Zeros` is `False`; ■

41/2

- When the parameter X has the value -1.0 , the `Log` function yields an imaginary result; and the `Arcsin` and `Arccos` functions yield a real result.

42

- When the parameter X has the value $\pm \langle i \rangle$, the `Log` function yields an imaginary result.

43

- Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand (as a complex value). Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero.

44

Other accuracy requirements for the complex elementary functions, which apply only in the strict mode, are given in Section 21.2.6 [G.2.6], page 1116.

45

The sign of a zero result or zero result component yielded by a complex elementary function is implementation defined when `Complex.Types.Real'Signed_Zeros` is `True`.

Implementation Permissions

46

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package with the appropriate predefined nongeneric equivalent of `Numerics.Generic_Complex_Types`; if they are, then the latter shall have been obtained by actual instantiation of `Numerics.Generic_Complex_Types`.

47

The exponentiation operator may be implemented in terms of the `Exp` and `Log` functions.

Because this implementation yields poor accuracy in some parts of the domain, no accuracy requirement is imposed on complex exponentiation.

48

The implementation of the `Exp` function of a complex parameter `X` is allowed to raise the exception `Constraint_Error`, signaling overflow, when the real component of `X` exceeds an unspecified threshold that is approximately $\log(\text{Complex_Types.Real'Safe_Last})$. This permission recognizes the impracticality of avoiding overflow in the marginal case that the exponential of the real component of `X` exceeds the safe range of `Complex.Types.Real` but both components of the final result do not. Similarly, the `Sin` and `Cos` (resp., `Sinh` and `Cosh`) functions are allowed to raise the exception `Constraint_Error`, signaling overflow, when the absolute value of the imaginary (resp., real) component of the parameter `X` exceeds an unspecified threshold that is approximately $\log(\text{Complex_Types.Real'Safe_Last}) + \log(2.0)$. This permission recognizes the impracticality of avoiding overflow in the marginal case that the hyperbolic sine or cosine of the imaginary (resp., real) component of `X` exceeds the safe range of `Complex.Types.Real` but both components of the final result do not.

Implementation Advice

49

Implementations in which `Complex.Types.Real'Signed_Zeros` is `True` should attempt to provide a rational treatment of the signs of zero results and result components. For example, many of the complex elementary functions have components that are odd functions of one of the parameter components; in these cases, the result component should have the sign of the parameter component at the origin. Other complex elementary functions have zero components whose sign is opposite that of a parameter component at the origin, or is always positive or always negative.

21.1.3 G.1.3 Complex Input-Output

1

The generic package `Text_IO.Complex_IO` defines procedures for the formatted input and output of complex values. The generic actual parameter in an instantiation of `Text_IO.Complex_IO` is an instance of `Numerics.Generic_Complex_Types` for some floating point subtype. Exceptional conditions are reported by raising the appropriate exception defined in `Text_IO`.

Static Semantics

2

The generic library package `Text_IO.Complex_IO` has the following declaration:

3

```
with Ada.Numerics.Generic_Complex_Types;
generic
  with package Complex_Types is
    new Ada.Numerics.Generic_Complex_Types (<>);
package Ada.Text_IO.Complex_IO is
```

4

```
  use Complex_Types;
```

5

```
Default_Fore : Field := 2;  
Default_Aft  : Field := Real'Digits - 1;  
Default_Exp  : Field := 3;
```

6

```
procedure Get (File : in File_Type;  
              Item  : out Complex;  
              Width : in Field := 0);  
procedure Get (Item  : out Complex;  
              Width : in Field := 0);
```

7

```
procedure Put (File : in File_Type;  
              Item  : in Complex;  
              Fore  : in Field := Default_Fore;  
              Aft   : in Field := Default_Aft;  
              Exp   : in Field := Default_Exp);  
procedure Put (Item  : in Complex;  
              Fore  : in Field := Default_Fore;  
              Aft   : in Field := Default_Aft;  
              Exp   : in Field := Default_Exp);
```

8

```
procedure Get (From : in String;  
              Item  : out Complex;  
              Last  : out Positive);  
procedure Put (To   : out String;  
              Item  : in Complex;  
              Aft   : in Field := Default_Aft;  
              Exp   : in Field := Default_Exp);
```

9

```
end Ada.Text_IO.Complex_IO;
```

9.1/2

The library package `Complex_Text_IO` defines the same subprograms as `Text_IO.Complex_IO`, except that the predefined type `Float` is systematically substituted for `Real`, and the type `Numerics.Complex_Types.Complex` is systematically substituted for `Complex` throughout. Non-generic equivalents of `Text_IO.Complex_IO` corresponding to each of the other predefined floating point types are defined similarly, with the names `Short_Complex_Text_IO`, `Long_Complex_Text_IO`, etc.

10

The semantics of the `Get` and `Put` procedures are as follows:

11

```
procedure Get (File : in File_Type;  
              Item  : out Complex;  
              Width : in Field := 0);  
procedure Get (Item  : out Complex;  
              Width : in Field := 0);
```

12/1

The input sequence is a pair of optionally signed real literals representing the real and imaginary components of a complex value. These components have the format defined for the corresponding Get procedure of an instance of `Text_IO.Float_IO` (see Section 15.10.9 [A.10.9], page 731) for the base subtype of `Complex.Types.Real`. The pair of components may be separated by a comma or surrounded by a pair of parentheses or both. Blanks are freely allowed before each of the components and before the parentheses and comma, if either is used. If the value of the parameter `Width` is zero, then

13

- line and page terminators are also allowed in these places;

14

- the components shall be separated by at least one blank or line terminator if the comma is omitted; and

15

- reading stops when the right parenthesis has been read, if the input sequence includes a left parenthesis, or when the imaginary component has been read, otherwise.

15.1

If a nonzero value of `Width` is supplied, then

16

- the components shall be separated by at least one blank if the comma is omitted; and

17

- exactly `Width` characters are read, or the characters (possibly none) up to a line terminator, whichever comes first (blanks are included in the count).

18

Returns, in the parameter `Item`, the value of type `Complex` that corresponds to the input sequence.

19

The exception `Text_IO.Data_Error` is raised if the input sequence does not have the required syntax or if the components of the complex value obtained are not of the base subtype of `Complex_Types.Real`.

20

```
procedure Put (File : in File_Type;
              Item : in Complex;
              Fore : in Field := Default_Fore;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
procedure Put (Item : in Complex;
              Fore : in Field := Default_Fore;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
```

21

Outputs the value of the parameter `Item` as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically,

22

- outputs a left parenthesis;

23

- outputs the value of the real component of the parameter `Item` with the format defined by the corresponding `Put` procedure of an instance of `Text_IO.Float_IO` for the base subtype of `Complex.Types.Real`, using the given values of `Fore`, `Aft`, and `Exp`;

24

- outputs a comma;

25

- outputs the value of the imaginary component of the parameter `Item` with the format defined by the corresponding `Put` procedure of an instance of `Text_IO.Float_IO` for the base subtype of `Complex.Types.Real`, using the given values of `Fore`, `Aft`, and `Exp`;

26

- outputs a right parenthesis.

27

```
procedure Get (From : in String;  
              Item  : out Complex;  
              Last  : out Positive);
```

28/2

Reads a complex value from the beginning of the given string, following the same rule as the `Get` procedure that reads a complex value from a file, but treating the end of the string as a file terminator. Returns, in the parameter `Item`, the value of type `Complex` that corresponds to the input sequence. Returns in `Last` the index value such that `From(Last)` is the last character read.

29

The exception `Text_IO.Data_Error` is raised if the input sequence does not have the required syntax or if the components of the complex value obtained are not of the base subtype of `Complex_Types.Real`.

30

```
procedure Put (To    : out String;
              Item  : in  Complex;
              Aft   : in  Field := Default_Aft;
              Exp   : in  Field := Default_Exp);
```

31

Outputs the value of the parameter `Item` to the given string as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically,

32

- a left parenthesis, the real component, and a comma are left justified in the given string, with the real component having the format defined by the `Put` procedure (for output to a file) of an instance of `Text_IO.Float_IO` for the base subtype of `Complex_Types.Real`, using a value of zero for `Fore` and the given values of `Aft` and `Exp`;

33

- the imaginary component and a right parenthesis are right justified in the given string, with the imaginary component having the format defined by the `Put` procedure (for output to a file) of an instance of `Text_IO.Float_IO` for the base subtype of `Complex_Types.Real`, using a value for `Fore` that completely fills the remainder of the string, together with the given values of `Aft` and `Exp`.

34

The exception `Text_IO.Layout_Error` is raised if the given string is too short to hold the formatted output.

Implementation Permissions

35

Other exceptions declared (by renaming) in `Text_IO` may be raised by the preceding procedures in the appropriate circumstances, as for the corresponding procedures of `Text_IO.Float_IO`.

21.1.4 G.1.4 The Package `Wide_Text_IO.Complex_IO`

Static Semantics

1

Implementations shall also provide the generic library package `Wide_Text_IO.Complex_IO`. Its declaration is obtained from that of `Text_IO.Complex_IO` by systematically replacing `Text_IO` by `Wide_Text_IO` and `String` by `Wide_String`; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide characters.

21.1.5 G.1.5 The Package `Wide_Wide_Text_IO.Complex_IO`

Static Semantics

1/2

Implementations shall also provide the generic library package `Wide_Wide_Text_IO.Complex_IO`. Its declaration is obtained from that of `Text_IO.Complex_IO` by systematically replacing `Text_IO` by `Wide_Wide_Text_IO` and `String` by `Wide_Wide_String`; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide wide characters.

21.2 G.2 Numeric Performance Requirements

Implementation Requirements

1

Implementations shall provide a user-selectable mode in which the accuracy and other numeric performance requirements detailed in the following subclauses are observed. This mode, referred to as the `<strict mode>`, may or may not be the default mode; it directly affects the results of the predefined arithmetic operations of real types and the results of the subprograms in children of the Numerics package, and indirectly affects the operations in other language defined packages. Implementations shall also provide the opposing mode, which is known as the `<relaxed mode>`.

Implementation Permissions

2

Either mode may be the default mode.

3

The two modes need not actually be different.

21.2.1 G.2.1 Model of Floating Point Arithmetic

1

In the strict mode, the predefined operations of a floating point type shall satisfy the accuracy requirements specified here and shall avoid or signal overflow in the situations described. This behavior is presented in terms of a model of floating point arithmetic that builds on the concept of the canonical form (see Section 15.5.3 [A.5.3], page 663).

Static Semantics

2

Associated with each floating point type is an infinite set of model numbers. The model numbers of a type are used to define the accuracy requirements that have to be satisfied by certain predefined operations of the type; through certain attributes of the model numbers, they are also used to explain the meaning of a user-declared floating point type declaration. The model numbers of a derived type are those of the parent type; the model numbers of a subtype are those of its type.

3

The <model numbers> of a floating point type T are zero and all the values expressible in the canonical form (for the type T), in which <mantissa> has T'Model_Mantissa digits and <exponent> has a value greater than or equal to T'Model_Emin. (These attributes are defined in Section 21.2.2 [G.2.2], page 1105.)

4

A <model interval> of a floating point type is any interval whose bounds are model numbers of the type. The <model interval> of a type T <associated with a value> <v> is the smallest model interval of T that includes <v>. (The model interval associated with a model number of a type consists of that number only.)

Implementation Requirements

5

The accuracy requirements for the evaluation of certain predefined operations of floating point types are as follows.

6

An <operand interval> is the model interval, of the type specified for the operand of an operation, associated with the value of the operand.

7

For any predefined arithmetic operation that yields a result of a floating point type T, the required bounds on the result are given by a model interval of T (called the <result interval>) defined in terms of the operand values as follows:

8

- The result interval is the smallest model interval of T that includes the minimum and the maximum of all the values obtained by applying the (exact) mathematical operation to values arbitrarily selected from the respective operand intervals.

9

The result interval of an exponentiation is obtained by applying the above rule to the sequence of multiplications defined by the exponent, assuming arbitrary association of the factors, and to the final division in the case of a negative exponent.

10

The result interval of a conversion of a numeric value to a floating point type `T` is the model interval of `T` associated with the operand value, except when the source expression is of a fixed point type with a `<small>` that is not a power of `T'Machine_Radix` or is a fixed point multiplication or division either of whose operands has a `<small>` that is not a power of `T'Machine_Radix`; in these cases, the result interval is implementation defined.

11

For any of the foregoing operations, the implementation shall deliver a value that belongs to the result interval when both bounds of the result interval are in the safe range of the result type `T`, as determined by the values of `T'Safe_First` and `T'Safe_Last`; otherwise,

12

- if `T'Machine_Overflows` is `True`, the implementation shall either deliver a value that belongs to the result interval or raise `Constraint_Error`;

13

- if `T'Machine_Overflows` is `False`, the result is implementation defined.

14

For any predefined relation on operands of a floating point type `T`, the implementation may deliver any value (i.e., either `True` or `False`) obtained by applying the (exact) mathematical comparison to values arbitrarily chosen from the respective operand intervals.

15

The result of a membership test is defined in terms of comparisons of the operand value with the lower and upper bounds of the given range or type mark (the usual rules apply to these comparisons).

Implementation Permissions

16

If the underlying floating point hardware implements division as multiplication by a reciprocal, the result interval for division (and exponentiation by a negative exponent) is implementation defined.

21.2.2 G.2.2 Model-Oriented Attributes of Floating Point Types

1

In implementations that support the Numerics Annex, the model-oriented attributes of floating point types shall yield the values defined here, in both the strict and the relaxed modes. These definitions add conditions to those in Section 15.5.3 [A.5.3], page 663.

Static Semantics

2

For every subtype `S` of a floating point type `<T>`:

3/2

`S'Model_Mantissa`

Yields the number of
digits in the mantissa
of the canonical

form of the model numbers of <T> (see Section 15.5.3 [A.5.3], page 663). The value of this attribute shall be greater than or equal to

3.1/2

$$\text{ceiling}(\langle d \rangle \cdot \log(10) / \log(\langle T \rangle \text{'Machine_Radix})) + \langle g \rangle$$

3.2/2

where <d> is the requested decimal precision of <T>, and <g> is 0 if <T>'Machine.Radix is a positive power of 10 and 1 otherwise. In addition, <T>'Model.Mantissa shall be less than or equal to the value of <T>'Machine.Mantissa. This attribute yields a value of the type <universal_integer>.

4
S'Model_Emin

Yields the minimum exponent of the canonical form of the model numbers of <T> (see Section 15.5.3 [A.5.3], page 663). The value of this attribute shall be greater than or equal to the value of <T>'Machine.Emin. This attribute yields a value of the type <universal_integer>.

5

S'Safe.First

Yields the lower bound of the safe range of <T>. The value of this attribute shall be a model number of <T> and greater than or equal to the lower bound of the base range of <T>. In addition, if <T> is declared by a floating_point_definition or is derived from such a type, and the floating_point_definition includes a real_range_specification specifying a lower bound of <lb>, then the value of this attribute shall be less than or equal to <lb>; otherwise, it shall be less than or equal to $-10.04 \cdot \langle d \rangle$, where <d> is the requested decimal precision of <T>. This attribute yields a value of the type <universal_real>.

6

S'Safe.Last

Yields the upper bound of the safe range of <T>. The value of this attribute shall be a model number of <T> and less than or equal to the upper bound of the base range of <T>. In

addition, if `<T>` is declared by a `floating_point_definition` or is derived from such a type, and the `floating_point_definition` includes a `real_range_specification` specifying an upper bound of `<ub>`, then the value of this attribute shall be greater than or equal to `<ub>`; otherwise, it shall be greater than or equal to $10.0 \cdot 4 \cdot \langle d \rangle$, where `d` is the requested decimal precision of `<T>`. This attribute yields a value of the type `<universal_real>`.

7
S'Model

Denotes a function (of a parameter `<X>`) whose specification is given in Section 15.5.3 [A.5.3], page 663. If `<X>` is a model number of `<T>`, the function yields `<X>`; otherwise, it yields the value obtained by rounding or truncating `<X>` to either one of the adjacent model numbers of `<T>`. `Constraint_Error` is raised if the resulting model number is outside the safe range of `S`. A zero result has the sign of `<X>`

when S'Signed_Zeros
is True.

8

Subject to the constraints given above, the values of S'Model_Mantissa and S'Safe_Last are to be maximized, and the values of S'Model_Emin and S'Safe_First minimized, by the implementation as follows:

9

- First, S'Model_Mantissa is set to the largest value for which values of S'Model_Emin, S'Safe_First, and S'Safe_Last can be chosen so that the implementation satisfies the strict-mode requirements of Section 21.2.1 [G.2.1], page 1104, in terms of the model numbers and safe range induced by these attributes.

10

- Next, S'Model_Emin is set to the smallest value for which values of S'Safe_First and S'Safe_Last can be chosen so that the implementation satisfies the strict-mode requirements of Section 21.2.1 [G.2.1], page 1104, in terms of the model numbers and safe range induced by these attributes and the previously determined value of S'Model_Mantissa.

11

- Finally, S'Safe_First and S'Safe_Last are set (in either order) to the smallest and largest values, respectively, for which the implementation satisfies the strict-mode requirements of Section 21.2.1 [G.2.1], page 1104, in terms of the model numbers and safe range induced by these attributes and the previously determined values of S'Model_Mantissa and S'Model_Emin.

21.2.3 G.2.3 Model of Fixed Point Arithmetic

1

In the strict mode, the predefined arithmetic operations of a fixed point type shall satisfy the accuracy requirements specified here and shall avoid or signal overflow in the situations described.

Implementation Requirements

2

The accuracy requirements for the predefined fixed point arithmetic operations and conversions, and the results of relations on fixed point operands, are given below.

3

The operands of the fixed point adding operators, absolute value, and comparisons have the same type. These operations are required to yield exact results, unless they overflow.

4

Multiplications and divisions are allowed between operands of any two fixed point types; the result has to be (implicitly or explicitly) converted to some other numeric type. For purposes of defining the accuracy rules, the multiplication or division and the conversion are treated as a single operation whose accuracy depends on three types (those of the operands

and the result). For decimal fixed point types, the attribute T'Round may be used to imply explicit conversion with rounding (see Section 4.5.10 [3.5.10], page 109).

5

When the result type is a floating point type, the accuracy is as given in Section 21.2.1 [G.2.1], page 1104. For some combinations of the operand and result types in the remaining cases, the result is required to belong to a small set of values called the <perfect result set>; for other combinations, it is required merely to belong to a generally larger and implementation–defined set of values called the <close result set>. When the result type is a decimal fixed point type, the perfect result set contains a single value; thus, operations on decimal types are always fully specified.

6

When one operand of a fixed–fixed multiplication or division is of type <universal_real>, that operand is not implicitly converted in the usual sense, since the context does not determine a unique target type, but the accuracy of the result of the multiplication or division (i.e., whether the result has to belong to the perfect result set or merely the close result set) depends on the value of the operand of type <universal_real> and on the types of the other operand and of the result.

7

For a fixed point multiplication or division whose (exact) mathematical result is <v>, and for the conversion of a value <v> to a fixed point type, the perfect result set and close result set are defined as follows:

8

- If the result type is an ordinary fixed point type with a <small> of <s>,

9

- if <v> is an integer multiple of <s>, then the perfect result set contains only the value <v>;

10

- otherwise, it contains the integer multiple of <s> just below <v> and the integer multiple of <s> just above <v>.

11

The close result set is an implementation–defined set of consecutive integer multiples of <s> containing the perfect result set as a subset.

12

- If the result type is a decimal type with a <small> of <s>,

13

- if $\langle v \rangle$ is an integer multiple of $\langle s \rangle$, then the perfect result set contains only the value $\langle v \rangle$;

14

- otherwise, if truncation applies then it contains only the integer multiple of $\langle s \rangle$ in the direction toward zero, whereas if rounding applies then it contains only the nearest integer multiple of $\langle s \rangle$ (with ties broken by rounding away from zero).

15

The close result set is an implementation—defined set of consecutive integer multiples of $\langle s \rangle$ containing the perfect result set as a subset.

16

- If the result type is an integer type,

17

- if $\langle v \rangle$ is an integer, then the perfect result set contains only the value $\langle v \rangle$;

18

- otherwise, it contains the integer nearest to the value $\langle v \rangle$ (if $\langle v \rangle$ lies equally distant from two consecutive integers, the perfect result set contains the one that is further from zero).

19

The close result set is an implementation—defined set of consecutive integers containing the perfect result set as a subset.

20

The result of a fixed point multiplication or division shall belong either to the perfect result set or to the close result set, as described below, if overflow does not occur. In the following cases, if the result type is a fixed point type, let $\langle s \rangle$ be its $\langle \text{small} \rangle$; otherwise, i.e. when the result type is an integer type, let $\langle s \rangle$ be 1.0.

21

- For a multiplication or division neither of whose operands is of type `<universal_real>`, let `<l>` and `<r>` be the `<smalls>` of the left and right operands. For a multiplication, if $(\langle l \rangle \cdot \langle r \rangle) / \langle s \rangle$ is an integer or the reciprocal of an integer (the `<smalls>` are said to be "compatible" in this case), the result shall belong to the perfect result set; otherwise, it belongs to the close result set. For a division, if $\langle l \rangle / (\langle r \rangle \cdot \langle s \rangle)$ is an integer or the reciprocal of an integer (i.e., the `<smalls>` are compatible), the result shall belong to the perfect result set; otherwise, it belongs to the close result set.

22

- For a multiplication or division having one `<universal_real>` operand with a value of `<v>`, note that it is always possible to factor `<v>` as an integer multiple of a "compatible" `<small>`, but the integer multiple may be "too big." If there exists a factorization in which that multiple is less than some implementation-defined limit, the result shall belong to the perfect result set; otherwise, it belongs to the close result set.

23

A multiplication $P * Q$ of an operand of a fixed point type `F` by an operand of an integer type `I`, or vice-versa, and a division P / Q of an operand of a fixed point type `F` by an operand of an integer type `I`, are also allowed. In these cases, the result has a type of `F`; explicit conversion of the result is never required. The accuracy required in these cases is the same as that required for a multiplication $F(P * Q)$ or a division $F(P / Q)$ obtained by interpreting the operand of the integer type to have a fixed point type with a `<small>` of 1.0.

24

The accuracy of the result of a conversion from an integer or fixed point type to a fixed point type, or from a fixed point type to an integer type, is the same as that of a fixed point multiplication of the source value by a fixed point operand having a `<small>` of 1.0 and a value of 1.0, as given by the foregoing rules. The result of a conversion from a floating point type to a fixed point type shall belong to the close result set. The result of a conversion of a `<universal_real>` operand to a fixed point type shall belong to the perfect result set.

25

The possibility of overflow in the result of a predefined arithmetic operation or conversion yielding a result of a fixed point type `T` is analogous to that for floating point types, except for being related to the base range instead of the safe range. If all of the permitted results belong to the base range of `T`, then the implementation shall deliver one of the permitted results; otherwise,

26

- if `T'Machine_Overflows` is `True`, the implementation shall either deliver one of the permitted results or raise `Constraint_Error`;

27

- if `T'Machine_Overflows` is `False`, the result is implementation defined.

21.2.4 G.2.4 Accuracy Requirements for the Elementary Functions

1

In the strict mode, the performance of `Numerics.Generic.Elementary.Functions` shall be as specified here.

Implementation Requirements

2

When an exception is not raised, the result of evaluating a function in an instance `<EF>` of `Numerics.Generic.Elementary.Functions` belongs to a `<result interval>`, defined as the smallest model interval of `<EF>.Float_Type` that contains all the values of the form $\langle f \rangle \cdot (1.0 + \langle d \rangle)$, where $\langle f \rangle$ is the exact value of the corresponding mathematical function at the given parameter values, $\langle d \rangle$ is a real number, and $|\langle d \rangle|$ is less than or equal to the function's `<maximum relative error>`. The function delivers a value that belongs to the result interval when both of its bounds belong to the safe range of `<EF>.Float_Type`; otherwise,

3

- if `<EF>.Float_Type'Machine_Overflows` is `True`, the function either delivers a value that belongs to the result interval or raises `Constraint_Error`, signaling overflow;

4

- if `<EF>.Float_Type'Machine_Overflows` is `False`, the result is implementation defined.

5

The maximum relative error exhibited by each function is as follows:

6

- $2.0 \cdot \langle \text{EF} \rangle.\text{Float_Type}'\text{Model_Epsilon}$, in the case of the `Sqrt`, `Sin`, and `Cos` functions;

7

- $4.0 \cdot \langle \text{EF} \rangle.\text{Float_Type}'\text{Model_Epsilon}$, in the case of the `Log`, `Exp`, `Tan`, `Cot`, and inverse trigonometric functions; and

8

- $8.0 \cdot \langle \text{EF} \rangle.\text{Float_Type}'\text{Model_Epsilon}$, in the case of the forward and inverse hyperbolic functions.

9

The maximum relative error exhibited by the exponentiation operator, which depends on the values of the operands, is $(4.0 + |\text{Right} \cdot \log(\text{Left})| / 32.0) \cdot \langle \text{EF} \rangle.\text{Float_Type}'\text{Model_Epsilon}$.

10

The maximum relative error given above applies throughout the domain of the forward trigonometric functions when the `Cycle` parameter is specified. When the `Cycle` parameter is omitted, the maximum relative error given above applies only when the absolute value

of the angle parameter X is less than or equal to some implementation-defined <angle threshold>, which shall be at least <EF>.Float_Type'Machine_Radix floor(<EF>.Float_Type'Machine_Mantissa/2). Beyond the angle threshold, the accuracy of the forward trigonometric functions is implementation defined.

11/2

The prescribed results specified in Section 15.5.1 [A.5.1], page 648, for certain functions at particular parameter values take precedence over the maximum relative error bounds; effectively, they narrow to a single value the result interval allowed by the maximum relative error bounds. Additional rules with a similar effect are given by table G-1 for the inverse trigonometric functions, at particular parameter values for which the mathematical result is possibly not a model number of <EF>.Float_Type (or is, indeed, even transcendental). In each table entry, the values of the parameters are such that the result lies on the axis between two quadrants; the corresponding accuracy rule, which takes precedence over the maximum relative error bounds, is that the result interval is the model interval of <EF>.Float_Type associated with the exact mathematical result given in the table.

12/1

<This paragraph was deleted.>

13

The last line of the table is meant to apply when <EF>.Float_Type'Signed_Zeros is False; the two lines just above it, when <EF>.Float_Type'Signed_Zeros is True and the parameter Y has a zero value with the indicated sign.

Table G-1: Tightly Approximated Elementary Function Results

Function	Value of X	Value of Y	Exact Result when Cycle Specified	Exact Result when Cycle Omitted
Arcsin	1.0	n.a.	Cycle/4.0	PI/2.0
Arcsin	-1.0	n.a.	-Cycle/4.0	-PI/2.0
Arccos	0.0	n.a.	Cycle/4.0	PI/2.0
Arccos	-1.0	n.a.	Cycle/2.0	PI
Arctan and Arccot	0.0	positive	Cycle/4.0	PI/2.0
Arctan and Arccot	0.0	negative	-Cycle/4.0	-PI/2.0
Arctan and Arccot	negative	+0.0	Cycle/2.0	PI
Arctan and Arccot	negative	-0.0	-Cycle/2.0	-PI
Arctan and Arccot	negative	0.0	Cycle/2.0	PI

14

The amount by which the result of an inverse trigonometric function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in Section 15.5.1 [A.5.1], page 648, is limited. The rule is that the result belongs to the smallest model interval of `<EF>.Float_Type` that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence over the maximum relative error bounds, effectively narrowing the result interval allowed by them.

15

Finally, the following specifications also take precedence over the maximum relative error bounds:

16

- The absolute value of the result of the Sin, Cos, and Tanh functions never exceeds one.

17

- The absolute value of the result of the Coth function is never less than one.

18

- The result of the Cosh function is never less than one.

Implementation Advice

19

The versions of the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of $2.0 * \text{Numerics.Pi}$, since this will not provide the required accuracy in some portions of the domain. For the same reason, the version of Log without a Base parameter should not be implemented by calling the corresponding version with a Base parameter of `Numerics.e`.

21.2.5 G.2.5 Performance Requirements for Random Number Generation

1

In the strict mode, the performance of `Numerics.Float_Random` and `Numerics.Discrete_Random` shall be as specified here.

Implementation Requirements

2

Two different calls to the time-dependent Reset procedure shall reset the generator to different states, provided that the calls are separated in time by at least one second and not more than fifty years.

3

The implementation's representations of generator states and its algorithms for generating random numbers shall yield a period of at least $2^{31}-2$; much longer periods are desirable but not required.

4

The implementations of `Numerics.Float_Random.Random` and `Numerics.Discrete_Random.Random` shall pass at least 85% of the individual trials in a

suite of statistical tests. For `Numerics.Float_Random`, the tests are applied directly to the floating point values generated (i.e., they are not converted to integers first), while for `Numerics.Discrete_Random` they are applied to the generated values of various discrete types. Each test suite performs 6 different tests, with each test repeated 10 times, yielding a total of 60 individual trials. An individual trial is deemed to pass if the chi-square value (or other statistic) calculated for the observed counts or distribution falls within the range of values corresponding to the 2.5 and 97.5 percentage points for the relevant degrees of freedom (i.e., it shall be neither too high nor too low). For the purpose of determining the degrees of freedom, measurement categories are combined whenever the expected counts are fewer than 5.

21.2.6 G.2.6 Accuracy Requirements for Complex Arithmetic

1

In the strict mode, the performance of `Numerics.Generic_Complex_Types` and `Numerics.Generic_Complex_Elementary_Functions` shall be as specified here.

Implementation Requirements

2

When an exception is not raised, the result of evaluating a real function of an instance `<CT>` of `Numerics.Generic_Complex_Types` (i.e., a function that yields a value of subtype `<CT>.Real_Base` or `<CT>.Imaginary`) belongs to a result interval defined as for a real elementary function (see Section 21.2.4 [G.2.4], page 1113).

3

When an exception is not raised, each component of the result of evaluating a complex function of such an instance, or of an instance of `Numerics.Generic_Complex_Elementary_Functions` obtained by instantiating the latter with `<CT>` (i.e., a function that yields a value of subtype `<CT>.Complex`), also belongs to a `<result interval>`. The result intervals for the components of the result are either defined by a `<maximum relative error>` bound or by a `<maximum box error>` bound. When the result interval for the real (resp., imaginary) component is defined by maximum relative error, it is defined as for that of a real function, relative to the exact value of the real (resp., imaginary) part of the result of the corresponding mathematical function. When defined by maximum box error, the result interval for a component of the result is the smallest model interval of `<CT>.Real` that contains all the values of the corresponding part of $\langle f \rangle \cdot (1.0 + \langle d \rangle)$, where $\langle f \rangle$ is the exact complex value of the corresponding mathematical function at the given parameter values, $\langle d \rangle$ is complex, and $|\langle d \rangle|$ is less than or equal to the given maximum box error. The function delivers a value that belongs to the result interval (or a value both of whose components belong to their respective result intervals) when both bounds of the result interval(s) belong to the safe range of `<CT>.Real`; otherwise,

4

- if `<CT>.Real_Machine_Overflows` is `True`, the function either delivers a value that belongs to the result interval (or a value both of whose components belong to their respective result intervals) or raises `Constraint_Error`, signaling overflow;

5

- if <CT>.Real'Machine_Overflows is False, the result is implementation defined.

6/2

The error bounds for particular complex functions are tabulated in table G–2. In the table, the error bound is given as the coefficient of <CT>.Real'Model_Epsilon.

7/1

<This paragraph was deleted.>

Table G–2: Error Bounds for Particular Complex Functions

Function or Operator	Nature of Result	Nature of Bound	Error Bound
Modulus	real	max. rel. error	3.0
Argument	real	max. rel. error	4.0
Compose_From_Polar	complex	max. rel. error	3.0
"*" (both operands complex)	complex	max. box error	5.0
"/" (right operand complex)	complex	max. box error	13.0
Sqrt	complex	max. rel. error	6.0
Log	complex	max. box error	13.0
Exp (complex parameter)	complex	max. rel. error	7.0
Exp (imaginary parameter)	complex	max. rel. error	2.0
Sin, Cos, Sinh, and Cosh	complex	max. rel. error	11.0
Tan, Cot, Tanh, and Coth	complex	max. rel. error	35.0
inverse trigonometric	complex	max. rel. error	14.0

inverse hyperbolic complex max. 14.0
rel.
error

8

The maximum relative error given above applies throughout the domain of the `Compose_From_Polar` function when the `Cycle` parameter is specified. When the `Cycle` parameter is omitted, the maximum relative error applies only when the absolute value of the parameter `Argument` is less than or equal to the angle threshold (see Section 21.2.4 [G.2.4], page 1113). For the `Exp` function, and for the forward hyperbolic (resp., trigonometric) functions, the maximum relative error given above likewise applies only when the absolute value of the imaginary (resp., real) component of the parameter `X` (or the absolute value of the parameter itself, in the case of the `Exp` function with a parameter of pure-imaginary type) is less than or equal to the angle threshold. For larger angles, the accuracy is implementation defined.

9

The prescribed results specified in Section 21.1.2 [G.1.2], page 1091, for certain functions at particular parameter values take precedence over the error bounds; effectively, they narrow to a single value the result interval allowed by the error bounds for a component of the result. Additional rules with a similar effect are given below for certain inverse trigonometric and inverse hyperbolic functions, at particular parameter values for which a component of the mathematical result is transcendental. In each case, the accuracy rule, which takes precedence over the error bounds, is that the result interval for the stated result component is the model interval of `<CT>.Real` associated with the component's exact mathematical value. The cases in question are as follows:

10

- When the parameter `X` has the value zero, the real (resp., imaginary) component of the result of the `Arccot` (resp., `Arccoth`) function is in the model interval of `<CT>.Real` associated with the value $\text{PI}/2.0$.

11

- When the parameter `X` has the value one, the real component of the result of the `Arcsin` function is in the model interval of `<CT>.Real` associated with the value $\text{PI}/2.0$.

12

- When the parameter `X` has the value -1.0 , the real component of the result of the `Arcsin` (resp., `Arccos`) function is in the model interval of `<CT>.Real` associated with the value $-\text{PI}/2.0$ (resp., PI).

13/2

The amount by which a component of the result of an inverse trigonometric or inverse hyperbolic function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in Section 21.1.2 [G.1.2], page 1091, is limited. The rule is that the result belongs to the smallest model interval of `<CT>.Real` that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes

precedence over the maximum error bounds, effectively narrowing the result interval allowed by them.

14

Finally, the results allowed by the error bounds are narrowed by one further rule: The absolute value of each component of the result of the Exp function, for a pure-imaginary parameter, never exceeds one.

Implementation Advice

15

The version of the Compose_From_Polar function without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of $2.0 * \text{Numerics.Pi}$, since this will not provide the required accuracy in some portions of the domain.

21.3 G.3 Vector and Matrix Manipulation

1/2

Types and operations for the manipulation of real vectors and matrices are provided in `Generic_Real_Arrays`, which is defined in Section 21.3.1 [G.3.1], page 1119. Types and operations for the manipulation of complex vectors and matrices are provided in `Generic_Complex_Arrays`, which is defined in Section 21.3.2 [G.3.2], page 1130. Both of these library units are generic children of the predefined package `Numerics` (see Section 15.5 [A.5], page 648). Nongeneric equivalents of these packages for each of the predefined floating point types are also provided as children of `Numerics`.

21.3.1 G.3.1 Real Vectors and Matrices

Static Semantics

1/2

The generic library package `Numerics.Generic_Real_Arrays` has the following declaration:

2/2

```
generic
  type Real is digits <>;
package Ada.Numerics.Generic_Real_Arrays is
  pragma Pure(Generic_Real_Arrays);
```

3/2

```
-- <Types>
```

4/2

```
type Real_Vector is array (Integer range <>) of Real'Base;
type Real_Matrix is array (Integer range <>, Integer range <>)
                        of Real'Base;
```

5/2

```
-- <Subprograms for Real_Vector types>
```

6/2

```
-- <Real_Vector arithmetic operations>
```

7/2

```
function "+" (Right : Real_Vector)      return Real_Vector;
function "-" (Right : Real_Vector)      return Real_Vector;
function "abs" (Right : Real_Vector)    return Real_Vector;
```

8/2

```
function "+" (Left, Right : Real_Vector) return Real_Vector;
function "-" (Left, Right : Real_Vector) return Real_Vector;
```

9/2

```
function "*" (Left, Right : Real_Vector) return Real'Base;
```

10/2

```
function "abs" (Right : Real_Vector)      return Real'Base;
```

11/2

```
-- <Real_Vector scaling operations>
```

12/2

```
function "*" (Left : Real'Base; Right : Real_Vector)
  return Real_Vector;
function "*" (Left : Real_Vector; Right : Real'Base)
  return Real_Vector;
function "/" (Left : Real_Vector; Right : Real'Base)
  return Real_Vector;
```

13/2

```
-- <Other Real_Vector operations>
```

14/2

```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Real_Vector;
```

15/2

```
-- <Subprograms for Real_Matrix types>
```

16/2

```
-- <Real_Matrix arithmetic operations>
```


17/2

```
function "+"      (Right : Real_Matrix) return Real_Matrix;
function "-"      (Right : Real_Matrix) return Real_Matrix;
function "abs"    (Right : Real_Matrix) return Real_Matrix;
function Transpose (X      : Real_Matrix) return Real_Matrix;
```

18/2

```
function "+" (Left, Right : Real_Matrix) return Real_Matrix;
function "-" (Left, Right : Real_Matrix) return Real_Matrix;
function "*" (Left, Right : Real_Matrix) return Real_Matrix;
```

19/2

```
function "*" (Left, Right : Real_Vector) return Real_Matrix;
```

20/2

```
function "*" (Left : Real_Vector; Right : Real_Matrix)
  return Real_Vector;
function "*" (Left : Real_Matrix; Right : Real_Vector)
  return Real_Vector;
```

21/2

```
-- <Real_Matrix scaling operations>
```

22/2

```
function "*" (Left : Real'Base; Right : Real_Matrix)
  return Real_Matrix;
function "*" (Left : Real_Matrix; Right : Real'Base)
  return Real_Matrix;
function "/" (Left : Real_Matrix; Right : Real'Base)
  return Real_Matrix;
```

23/2

```
-- <Real_Matrix inversion and related operations>
```

24/2

```
function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;
function Solve (A, X : Real_Matrix) return Real_Matrix;
function Inverse (A : Real_Matrix) return Real_Matrix;
function Determinant (A : Real_Matrix) return Real'Base;
```

25/2

```
-- <Eigenvalues and vectors of a real symmetric matrix>
```

26/2

```
function Eigenvalues (A : Real_Matrix) return Real_Vector;
```

27/2

```
procedure Eigensystem (A          : in  Real_Matrix;  
                      Values     : out Real_Vector;  
                      Vectors    : out Real_Matrix);
```

28/2

```
-- <Other Real_Matrix operations>
```

29/2

```
function Unit_Matrix (Order          : Positive;  
                     First_1, First_2 : Integer := 1)  
                    return Real_Matrix;
```

30/2

```
end Ada.Numerics.Generic_Real_Arrays;
```

31/2

The library package Numerics.Real_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Real_Arrays, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Real_Arrays, Numerics.Long_Real_Arrays, etc.

32/2

Two types are defined and exported by Numerics.Generic_Real_Arrays. The composite type Real_Vector is provided to represent a vector with components of type Real; it is defined as an unconstrained, one-dimensional array with an index of type Integer. The composite type Real_Matrix is provided to represent a matrix with components of type Real; it is defined as an unconstrained, two-dimensional array with indices of type Integer.

33/2

The effect of the various subprograms is as described below. In most cases the subprograms are described in terms of corresponding scalar operations of the type Real; any exception raised by those operations is propagated by the array operation. Moreover, the accuracy of the result for each individual component is as defined for the scalar operation unless stated otherwise.

34/2

In the case of those operations which are defined to <involve an inner product>, Constraint_Error may be raised if an intermediate result is outside the range of Real'Base even though the mathematical final result would not be.

35/2

```
function "+" (Right : Real_Vector) return Real_Vector;
```

```
function "-" (Right : Real_Vector) return Real_Vector;  
function "abs" (Right : Real_Vector) return Real_Vector;
```

36/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index range of the result is Right'Range.

37/2

```
function "+" (Left, Right : Real_Vector) return Real_Vector;  
function "-" (Left, Right : Real_Vector) return Real_Vector;
```

38/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

39/2

```
function "*" (Left, Right : Real_Vector) return Real'Base;
```

40/2

This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product.

41/2

```
function "abs" (Right : Real_Vector) return Real'Base;
```

42/2

This operation returns the L2-norm of Right (the square root of the inner product of the vector with itself).

43/2

```
function "*" (Left : Real'Base; Right : Real_Vector) return Real_Vector;■
```

44/2

This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index range of the result is Right'Range.

45/2

```
function "*" (Left : Real_Vector; Right : Real'Base) return Real_Vector;█  
function "/" (Left : Real_Vector; Right : Real'Base) return Real_Vector;█
```

46/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index range of the result is Left'Range.

47/2

```
function Unit_Vector (Index : Integer;  
                      Order : Positive;  
                      First : Integer := 1) return Real_Vector;
```

48/2

This function returns a <unit vector> with Order components and a lower bound of First. All components are set to 0.0 except for the Index component which is set to 1.0. Constraint_Error is raised if Index < First, Index > First + Order - 1 or if First + Order - 1 > Integer'Last.

49/2

```
function "+" (Right : Real_Matrix) return Real_Matrix;  
function "-" (Right : Real_Matrix) return Real_Matrix;  
function "abs" (Right : Real_Matrix) return Real_Matrix;
```

50/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index ranges of the result are those of Right.

51/2

```
function Transpose (X : Real_Matrix) return Real_Matrix;
```

52/2

This function returns the transpose of a matrix X . The first and second index ranges of the result are X' Range(2) and X' Range(1) respectively.

53/2

```
function "+" (Left, Right : Real_Matrix) return Real_Matrix;  
function "-" (Left, Right : Real_Matrix) return Real_Matrix;
```

54/2

Each operation returns the result of applying the corresponding operation of the type `Real` to each component of `Left` and the matching component of `Right`. The index ranges of the result are those of `Left`. `Constraint_Error` is raised if `Left'Length(1)` is not equal to `Right'Length(1)` or `Left'Length(2)` is not equal to `Right'Length(2)`.

55/2

```
function "*" (Left, Right : Real_Matrix) return Real_Matrix;
```

56/2

This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are `Left'Range(1)` and `Right'Range(2)` respectively. `Constraint_Error` is raised if `Left'Length(2)` is not equal to `Right'Length(1)`. This operation involves inner products.

57/2

```
function "*" (Left, Right : Real_Vector) return Real_Matrix;
```

58/2

This operation returns the outer product of a (column) vector `Left` by a (row) vector `Right` using the operation `"*"` of the type `Real` for computing the individual components. The first and second index ranges of the result are `Left'Range` and `Right'Range` respectively.

59/2

```
function "*" (Left : Real_Vector; Right : Real_Matrix) return Real_Vector;■
```

60/2

This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.

61/2

```
function "*" (Left : Real_Matrix; Right : Real_Vector) return Real_Vector;■
```

62/2

This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.

63/2

```
function "*" (Left : Real'Base; Right : Real_Matrix) return Real_Matrix;■
```

64/2

This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index ranges of the result are those of Right.

65/2

```
function "*" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;■  
function "/" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;■
```

66/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the

scalar Right. The index ranges of the result are those of Left.

67/2

```
function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;■
```

68/2

This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

69/2

```
function Solve (A, X : Real_Matrix) return Real_Matrix;
```

70/2

This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

71/2

```
function Inverse (A : Real_Matrix) return Real_Matrix;
```

72/2

This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.

73/2

```
function Determinant (A : Real_Matrix) return Real_Base;
```

74/2

This function returns the determinant of the matrix A. `Constraint_Error` is raised if `A'Length(1)` is not equal to `A'Length(2)`.

75/2

```
function Eigenvalues(A : Real_Matrix) return Real_Vector;
```

76/2

This function returns the eigenvalues of the symmetric matrix A as a vector sorted into order with the largest first. `Constraint_Error` is raised if `A'Length(1)` is not equal to `A'Length(2)`. The index range of the result is `A'Range(1)`. `Argument_Error` is raised if the matrix A is not symmetric.

77/2

```
procedure Eigensystem(A      : in Real_Matrix;  
                      Values  : out Real_Vector;  
                      Vectors : out Real_Matrix);
```

78/2

This procedure computes both the eigenvalues and eigenvectors of the symmetric matrix A. The out parameter `Values` is the same as that obtained by calling the function `Eigenvalues`. The out parameter `Vectors` is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are normalized and mutually orthogonal (they are orthonormal), including when there are repeated eigenvalues. `Constraint_Error` is raised if `A'Length(1)` is not equal to `A'Length(2)`. The index ranges of the parameter `Vectors` are those of A. `Argument_Error` is raised if the matrix A is not symmetric.

79/2

```
function Unit_Matrix (Order      : Positive;
```


First_1, First_2 : Integer := 1) return Real_Matrix;■

80/2

This function returns a square <unit matrix> with Order^2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to 0.0 except for the main diagonal, whose components are set to 1.0. Constraint_Error is raised if $\text{First}_1 + \text{Order} - 1 > \text{Integer}'\text{Last}$ or $\text{First}_2 + \text{Order} - 1 > \text{Integer}'\text{Last}$.

Implementation Requirements

81/2

Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.

82/2

For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real in both the strict mode and the relaxed mode (see Section 21.2 [G.2], page 1103).

83/2

For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product $\langle X \rangle * \langle Y \rangle$ shall not exceed $\langle g \rangle * \text{abs}(\langle X \rangle) * \text{abs}(\langle Y \rangle)$ where $\langle g \rangle$ is defined as

84/2

$$\langle g \rangle = \langle X \rangle.\text{Length} * \text{Real}'\text{Machine_Radix}^{(1 - \text{Real}'\text{Model_Mantissa})}$$
 ■

85/2

For the L2–norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed $\langle g \rangle / 2.0 + 3.0 * \text{Real}'\text{Model_Epsilon}$ where $\langle g \rangle$ is defined as above.

Documentation Requirements

86/2

Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

Implementation Permissions

87/2

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

Implementation Advice

88/2

Implementations should implement the Solve and Inverse functions using established techniques such as LU decomposition with row interchanges followed by back and forward substitution. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done then it should be documented.

89/2

It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise `Constraint_Error` is sufficient.

90/2

The test that a matrix is symmetric should be performed by using the equality operator to compare the relevant components.

21.3.2 G.3.2 Complex Vectors and Matrices

Static Semantics

1/2

The generic library package `Numerics.Generic_Complex_Arrays` has the following declaration:

2/2

```
with Ada.Numerics.Generic_Real_Arrays, Ada.Numerics.Generic_Complex_Types;
generic
  with package Real_Arrays is new
    Ada.Numerics.Generic_Real_Arrays (<>);
  use Real_Arrays;
  with package Complex_Types is new
    Ada.Numerics.Generic_Complex_Types (Real);
  use Complex_Types;
package Ada.Numerics.Generic_Complex_Arrays is
  pragma Pure(Generic_Complex_Arrays);
```

3/2

```
-- <Types>
```

4/2

```
type Complex_Vector is array (Integer range <>) of Complex;
type Complex_Matrix is array (Integer range <>,
                               Integer range <>) of Complex;
```

5/2

```
-- <Subprograms for Complex_Vector types>
```

6/2

```
-- <Complex_Vector selection, conversion and composition operations>
```

7/2

```
function Re (X : Complex_Vector) return Real_Vector;
function Im (X : Complex_Vector) return Real_Vector;
```

8/2

```
procedure Set_Re (X : in out Complex_Vector;
                 Re : in   Real_Vector);
procedure Set_Im (X : in out Complex_Vector;
                 Im : in   Real_Vector);
```

9/2

```
function Compose_From_Cartesian (Re      : Real_Vector)
  return Complex_Vector;
function Compose_From_Cartesian (Re, Im : Real_Vector)
  return Complex_Vector;
```

10/2

```
function Modulus (X      : Complex_Vector) return Real_Vector;
function "abs"    (Right : Complex_Vector) return Real_Vector
  renames Modulus;
function Argument (X      : Complex_Vector) return Real_Vector;
function Argument (X      : Complex_Vector;
                  Cycle : Real'Base)      return Real_Vector;
```

11/2

```
function Compose_From_Polar (Modulus, Argument : Real_Vector)
  return Complex_Vector;
function Compose_From_Polar (Modulus, Argument : Real_Vector;
                             Cycle             : Real'Base)
  return Complex_Vector;
```

12/2

```
-- <Complex_Vector arithmetic operations>
```

13/2

```
function "+"      (Right : Complex_Vector) return Complex_Vector;█
function "-"      (Right : Complex_Vector) return Complex_Vector;█
function Conjugate (X      : Complex_Vector) return Complex_Vector;█
```

14/2

```
function "+" (Left, Right : Complex_Vector) return Complex_Vector;█
function "-" (Left, Right : Complex_Vector) return Complex_Vector;█
```

15/2

```
function "*" (Left, Right : Complex_Vector) return Complex;
```

16/2

```
function "abs"      (Right : Complex_Vector) return Complex;
```

17/2

```
-- <Mixed Real_Vector and Complex_Vector arithmetic operations>
```

18/2

```
function "+" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "+" (Left  : Complex_Vector;
              Right : Real_Vector)   return Complex_Vector;
function "-" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "-" (Left  : Complex_Vector;
              Right : Real_Vector)   return Complex_Vector;
```

19/2

```
function "*" (Left  : Real_Vector;   Right : Complex_Vector)
  return Complex;
function "*" (Left  : Complex_Vector; Right : Real_Vector)
  return Complex;
```

20/2

```
-- <Complex_Vector scaling operations>
```

21/2

```
function "*" (Left  : Complex;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Complex)       return Complex_Vector;
function "/" (Left  : Complex_Vector;
              Right : Complex)       return Complex_Vector;
```

22/2

```
function "*" (Left  : Real'Base;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real'Base)     return Complex_Vector;
function "/" (Left  : Complex_Vector;
              Right : Real'Base)     return Complex_Vector;
```

23/2

```
-- <Other Complex_Vector operations>
```



```

    return Complex_Matrix;
33/2

-- <Complex_Matrix arithmetic operations>
34/2

function "+"      (Right : Complex_Matrix) return Complex_Matrix;█
function "-"      (Right : Complex_Matrix) return Complex_Matrix;█
function Conjugate (X      : Complex_Matrix) return Complex_Matrix;█
function Transpose (X      : Complex_Matrix) return Complex_Matrix;█
35/2

function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;█
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;█
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;█
36/2

function "*" (Left, Right : Complex_Vector) return Complex_Matrix;█
37/2

function "*" (Left  : Complex_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;
38/2

-- <Mixed Real_Matrix and Complex_Matrix arithmetic operations>
39/2

function "+" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)  return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)  return Complex_Matrix;
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)  return Complex_Matrix;
40/2

function "*" (Left  : Real_Vector;

```

```
function "*" (Left : Complex_Vector;
              Right : Complex_Vector) return Complex_Matrix;
function "*" (Left : Complex_Vector;
              Right : Real_Vector)   return Complex_Matrix;
```

41/2

```
function "*" (Left : Real_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left : Complex_Vector;
              Right : Real_Matrix)   return Complex_Vector;
function "*" (Left : Real_Matrix;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left : Complex_Matrix;
              Right : Real_Vector)   return Complex_Vector;
```

42/2

```
-- <Complex_Matrix scaling operations>
```

43/2

```
function "*" (Left : Complex;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left : Complex_Matrix;
              Right : Complex)       return Complex_Matrix;
function "/" (Left : Complex_Matrix;
              Right : Complex)       return Complex_Matrix;
```

44/2

```
function "*" (Left : Real'Base;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left : Complex_Matrix;
              Right : Real'Base)     return Complex_Matrix;
function "/" (Left : Complex_Matrix;
              Right : Real'Base)     return Complex_Matrix;
```

45/2

```
-- <Complex_Matrix inversion and related operations>
```

46/2

```
function Solve (A : Complex_Matrix; X : Complex_Vector)
  return Complex_Vector;
function Solve (A, X : Complex_Matrix) return Complex_Matrix;
function Inverse (A : Complex_Matrix) return Complex_Matrix;
function Determinant (A : Complex_Matrix) return Complex;
```

47/2

```

-- <Eigenvalues and vectors of a Hermitian matrix>
48/2

function Eigenvalues(A : Complex_Matrix) return Real_Vector;

49/2

procedure Eigensystem(A      : in Complex_Matrix;
                      Values : out Real_Vector;
                      Vectors : out Complex_Matrix);

50/2

-- <Other Complex_Matrix operations>

51/2

function Unit_Matrix (Order      : Positive;
                     First_1, First_2 : Integer := 1)
                    return Complex_Matrix;

52/2

end Ada.Numerics.Generic_Complex_Arrays;

```

53/2

The library package `Numerics.Complex_Arrays` is declared pure and defines the same types and subprograms as `Numerics.Generic_Complex_Arrays`, except that the predefined type `Float` is systematically substituted for `Real_Base`, and the `Real_Vector` and `Real_Matrix` types exported by `Numerics.Real_Arrays` are systematically substituted for `Real_Vector` and `Real_Matrix`, and the `Complex` type exported by `Numerics.Complex_Types` is systematically substituted for `Complex`, throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names `Numerics.Short_Complex_Arrays`, `Numerics.Long_Complex_Arrays`, etc.

54/2

Two types are defined and exported by `Numerics.Generic_Complex_Arrays`. The composite type `Complex_Vector` is provided to represent a vector with components of type `Complex`; it is defined as an unconstrained one-dimensional array with an index of type `Integer`. The composite type `Complex_Matrix` is provided to represent a matrix with components of type `Complex`; it is defined as an unconstrained, two-dimensional array with indices of type `Integer`.

55/2

The effect of the various subprograms is as described below. In many cases they are described in terms of corresponding scalar operations in `Numerics.Generic_Complex_Types`. Any exception raised by those operations is propagated by the array subprogram. Moreover, any constraints on the parameters and the accuracy of the result for each individual component are as defined for the scalar operation.

56/2

In the case of those operations which are defined to <involve an inner product>, Con-

straint_Error may be raised if an intermediate result has a component outside the range of Real'Base even though the final mathematical result would not.

57/2

```
function Re (X : Complex_Vector) return Real_Vector;  
function Im (X : Complex_Vector) return Real_Vector;
```

58/2

Each function returns a vector of the specified Cartesian components of X. The index range of the result is X'Range.

59/2

```
procedure Set_Re (X : in out Complex_Vector; Re : in Real_Vector);  
procedure Set_Im (X : in out Complex_Vector; Im : in Real_Vector);
```

60/2

Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length is not equal to Re'Length or Im'Length.

61/2

```
function Compose_From_Cartesian (Re      : Real_Vector)  
  return Complex_Vector;  
function Compose_From_Cartesian (Re, Im : Real_Vector)  
  return Complex_Vector;
```

62/2

Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index range of the result is Re'Range. Constraint_Error is raised if Re'Length is not equal to Im'Length.

63/2

```
function Modulus (X      : Complex_Vector) return Real_Vector;  
function "abs"   (Right : Complex_Vector) return Real_Vector
```

```

                                renames Modulus;
function Argument (X      : Complex_Vector) return Real_Vector;
function Argument (X      : Complex_Vector;
                  Cycle : Real'Base)      return Real_Vector;

```

64/2

Each function calculates and returns a vector of the specified polar components of X or Right using the corresponding function in `numerics.generic_complex_types`. The index range of the result is `X'Range` or `Right'Range`.

65/2

```

function Compose_From_Polar (Modulus, Argument : Real_Vector)
    return Complex_Vector;
function Compose_From_Polar (Modulus, Argument : Real_Vector;
                            Cycle              : Real'Base)
    return Complex_Vector;

```

66/2

Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of polar components using the corresponding function in `numerics.generic_complex_types` on matching components of Modulus and Argument. The index range of the result is `Modulus'Range`. `Constraint_Error` is raised if `Modulus'Length` is not equal to `Argument'Length`.

67/2

```

function "+" (Right : Complex_Vector) return Complex_Vector;
function "-" (Right : Complex_Vector) return Complex_Vector;

```

68/2

Each operation returns the result of applying the corresponding operation in `numerics.generic_complex_types` to each component of Right. The index range of the result is `Right'Range`.

69/2

```

function Conjugate (X : Complex_Vector) return Complex_Vector;

```

70/2

This function returns the result of applying the appropriate function `Conjugate` in `numerics.generic_complex_types` to each component of `X`. The index range of the result is `X'Range`.

71/2

```
function "+" (Left, Right : Complex_Vector) return Complex_Vector;  
function "-" (Left, Right : Complex_Vector) return Complex_Vector;
```

72/2

Each operation returns the result of applying the corresponding operation in `numerics.-generic_complex_types` to each component of `Left` and the matching component of `Right`. The index range of the result is `Left'Range`. `Constraint_Error` is raised if `Left'Length` is not equal to `Right'Length`.

73/2

```
function "*" (Left, Right : Complex_Vector) return Complex;
```

74/2

This operation returns the inner product of `Left` and `Right`. `Constraint_Error` is raised if `Left'Length` is not equal to `Right'Length`. This operation involves an inner product.

75/2

```
function "abs" (Right : Complex_Vector) return Complex;
```

76/2

This operation returns the Hermitian `L2-norm` of `Right` (the square root of the inner product of the vector with its conjugate).

77/2

```
function "+" (Left : Real_Vector;  
             Right : Complex_Vector) return Complex_Vector;  
function "+" (Left : Complex_Vector;
```

```

                                Right : Real_Vector)    return Complex_Vector;
function "-" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "-" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;

```

78/2

Each operation returns the result of applying the corresponding operation in `numerics.generic_complex_types` to each component of `Left` and the matching component of `Right`. The index range of the result is `Left'Range`. `Constraint_Error` is raised if `Left'Length` is not equal to `Right'Length`.

79/2

```

function "*" (Left : Real_Vector;    Right : Complex_Vector) return Complex;
function "*" (Left : Complex_Vector; Right : Real_Vector)    return Complex;

```

80/2

Each operation returns the inner product of `Left` and `Right`. `Constraint_Error` is raised if `Left'Length` is not equal to `Right'Length`. These operations involve an inner product.

81/2

```

function "*" (Left : Complex; Right : Complex_Vector) return Complex_Vector;

```

82/2

This operation returns the result of multiplying each component of `Right` by the complex number `Left` using the appropriate operation `"*"` in `numerics.generic_complex_types`. The index range of the result is `Right'Range`.

83/2

```

function "*" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
function "/" (Left : Complex_Vector; Right : Complex) return Complex_Vector;

```

84/2

Each operation returns the result of applying the corresponding operation in `numerics.generic_complex_types` to each component of the vector `Left` and the

complex number Right. The index range of the result is Left'Range.

85/2

```
function "*" (Left : Real'Base;  
             Right : Complex_Vector) return Complex_Vector;
```

86/2

This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in numerics.generic_complex_types. The index range of the result is Right'Range.

87/2

```
function "*" (Left : Complex_Vector;  
             Right : Real'Base) return Complex_Vector;  
function "/" (Left : Complex_Vector;  
             Right : Real'Base) return Complex_Vector;
```

88/2

Each operation returns the result of applying the corresponding operation in numerics.-generic_complex_types to each component of the vector Left and the real number Right. The index range of the result is Left'Range.

89/2

```
function Unit_Vector (Index : Integer;  
                     Order  : Positive;  
                     First  : Integer := 1) return Complex_Vector;
```

90/2

This function returns a <unit vector> with Order components and a lower bound of First. All components are set to (0.0, 0.0) except for the Index component which is set to (1.0, 0.0). Constraint_Error is raised if Index < First, Index > First + Order - 1, or if First + Order - 1 > Integer'Last.

91/2

```
function Re (X : Complex_Matrix) return Real_Matrix;  
function Im (X : Complex_Matrix) return Real_Matrix;
```

92/2

Each function returns a matrix of the specified Cartesian components of X. The index ranges of the result are those of X.

93/2

```
procedure Set_Re (X : in out Complex_Matrix; Re : in Real_Matrix);
procedure Set_Im (X : in out Complex_Matrix; Im : in Real_Matrix);
```

94/2

Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length(1) is not equal to Re'Length(1) or Im'Length(1) or if X'Length(2) is not equal to Re'Length(2) or Im'Length(2).

95/2

```
function Compose_From_Cartesian (Re      : Real_Matrix)
  return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix)
  return Complex_Matrix;
```

96/2

Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index ranges of the result are those of Re. Constraint_Error is raised if Re'Length(1) is not equal to Im'Length(1) or Re'Length(2) is not equal to Im'Length(2).

97/2

```
function Modulus (X      : Complex_Matrix) return Real_Matrix;
function "abs"   (Right : Complex_Matrix) return Real_Matrix
  renames Modulus;
function Argument (X      : Complex_Matrix) return Real_Matrix;
function Argument (X      : Complex_Matrix);
```

```
Cycle : Real'Base)      return Real_Matrix;
```

98/2

Each function calculates and returns a matrix of the specified polar components of X or Right using the corresponding function in `numerics.generic_complex_types`. The index ranges of the result are those of X or Right.

99/2

```
function Compose_From_Polar (Modulus, Argument : Real_Matrix)
  return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
  return Complex_Matrix;
```

100/2

Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of polar components using the corresponding function in `numerics.generic_complex_types` on matching components of Modulus and Argument. The index ranges of the result are those of Modulus. `Constraint_Error` is raised if `Modulus'Length(1)` is not equal to `Argument'Length(1)` or `Modulus'Length(2)` is not equal to `Argument'Length(2)`.

101/2

```
function "+" (Right : Complex_Matrix) return Complex_Matrix;
function "-" (Right : Complex_Matrix) return Complex_Matrix;
```

102/2

Each operation returns the result of applying the corresponding operation in `numerics.generic_complex_types` to each component of Right. The index ranges of the result are those of Right.

103/2

```
function Conjugate (X : Complex_Matrix) return Complex_Matrix;
```

104/2

This function returns the result of applying the appropriate function `Conjugate` in `numerics.generic_complex_types` to each component of `X`. The index ranges of the result are those of `X`.

105/2

```
function Transpose (X : Complex_Matrix) return Complex_Matrix;
```

106/2

This function returns the transpose of a matrix `X`. The first and second index ranges of the result are `X'Range(2)` and `X'Range(1)` respectively.

107/2

```
function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;  
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

108/2

Each operation returns the result of applying the corresponding operation in `numerics.generic_complex_types` to each component of `Left` and the matching component of `Right`. The index ranges of the result are those of `Left`. `Constraint_Error` is raised if `Left'Length(1)` is not equal to `Right'Length(1)` or `Left'Length(2)` is not equal to `Right'Length(2)`.

109/2

```
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;
```

110/2

This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are `Left'Range(1)` and `Right'Range(2)` respectively. `Constraint_Error` is raised if `Left'Length(2)` is not equal to `Right'Length(1)`. This operation involves inner products.

111/2


```
function "*" (Left, Right : Complex_Vector) return Complex_Matrix;
```

112/2

This operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in numerics.generic_complex_types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

113/2

```
function "*" (Left : Complex_Vector;
             Right : Complex_Matrix) return Complex_Vector;
```

114/2

This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.

115/2

```
function "*" (Left : Complex_Matrix;
             Right : Complex_Vector) return Complex_Vector;
```

116/2

This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.

117/2

```
function "+" (Left : Real_Matrix;
             Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left : Complex_Matrix;
             Right : Real_Matrix) return Complex_Matrix;
function "-" (Left : Real_Matrix;
             Right : Complex_Matrix) return Complex_Matrix;
```

```
function "-" (Left : Complex_Matrix;
             Right : Real_Matrix) return Complex_Matrix;
```

118/2

Each operation returns the result of applying the corresponding operation in `numerics.generic_complex_types` to each component of `Left` and the matching component of `Right`. The index ranges of the result are those of `Left`. `Constraint_Error` is raised if `Left'Length(1)` is not equal to `Right'Length(1)` or `Left'Length(2)` is not equal to `Right'Length(2)`.

119/2

```
function "*" (Left : Real_Matrix;
             Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left : Complex_Matrix;
             Right : Real_Matrix) return Complex_Matrix;
```

120/2

Each operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are `Left'Range(1)` and `Right'Range(2)` respectively. `Constraint_Error` is raised if `Left'Length(2)` is not equal to `Right'Length(1)`. These operations involve inner products.

121/2

```
function "*" (Left : Real_Vector;
             Right : Complex_Vector) return Complex_Matrix;
function "*" (Left : Complex_Vector;
             Right : Real_Vector) return Complex_Matrix;
```

122/2

Each operation returns the outer product of a (column) vector `Left` by a (row) vector `Right` using the appropriate operation `"*"` in `numerics.generic_complex_types` for computing the individual components. The first and second index ranges of the result are `Left'Range` and `Right'Range` respectively.

123/2

```
function "*" (Left : Real_Vector;  
             Right : Complex_Matrix) return Complex_Vector;  
function "*" (Left : Complex_Vector;  
             Right : Real_Matrix)   return Complex_Vector;
```

124/2

Each operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). These operations involve inner products.

125/2

```
function "*" (Left : Real_Matrix;  
             Right : Complex_Vector) return Complex_Vector;  
function "*" (Left : Complex_Matrix;  
             Right : Real_Vector)   return Complex_Vector;
```

126/2

Each operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. These operations involve inner products.

127/2

```
function "*" (Left : Complex; Right : Complex_Matrix) return Complex_Matrix;■
```

128/2

This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in numerics.generic_complex_types. The index ranges of the result are those of Right.

129/2

```
function "*" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;■  
function "/" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;■
```

130/2

Each operation returns the result of applying the corresponding operation in `numerics.generic_complex_types` to each component of the matrix `Left` and the complex number `Right`. The index ranges of the result are those of `Left`.

131/2

```
function "*" (Left : Real'Base;  
             Right : Complex_Matrix) return Complex_Matrix;
```

132/2

This operation returns the result of multiplying each component of `Right` by the real number `Left` using the appropriate operation `"*"` in `numerics.generic_complex_types`. The index ranges of the result are those of `Right`.

133/2

```
function "*" (Left : Complex_Matrix;  
             Right : Real'Base) return Complex_Matrix;  
function "/" (Left : Complex_Matrix;  
             Right : Real'Base) return Complex_Matrix;
```

134/2

Each operation returns the result of applying the corresponding operation in `numerics.generic_complex_types` to each component of the matrix `Left` and the real number `Right`. The index ranges of the result are those of `Left`.

135/2

```
function Solve (A : Complex_Matrix; X : Complex_Vector) return Complex_Vector;■
```

136/2

This function returns a vector `Y` such that `X` is (nearly) equal to `A * Y`. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is `A'Range(2)`. `Constraint_Error` is raised if `A'Length(1)`,

A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

137/2

```
function Solve (A, X : Complex_Matrix) return Complex_Matrix;
```

138/2

This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

139/2

```
function Inverse (A : Complex_Matrix) return Complex_Matrix;
```

140/2

This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.

141/2

```
function Determinant (A : Complex_Matrix) return Complex;
```

142/2

This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2).

143/2

```
function Eigenvalues(A : Complex_Matrix) return Real_Vector;
```

144/2

This function returns the eigenvalues of the Hermitian matrix A as a vector sorted into

order with the largest first. `Constraint_Error` is raised if `A'Length(1)` is not equal to `A'Length(2)`. The index range of the result is `A'Range(1)`. `Argument_Error` is raised if the matrix `A` is not Hermitian.

145/2

```
procedure Eigensystem(A          : in Complex_Matrix;
                     Values     : out Real_Vector;
                     Vectors    : out Complex_Matrix);
```

146/2

This procedure computes both the eigenvalues and eigenvectors of the Hermitian matrix `A`. The out parameter `Values` is the same as that obtained by calling the function `Eigenvalues`. The out parameter `Vectors` is a matrix whose columns are the eigenvectors of the matrix `A`. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are mutually orthonormal, including when there are repeated eigenvalues. `Constraint_Error` is raised if `A'Length(1)` is not equal to `A'Length(2)`. The index ranges of the parameter `Vectors` are those of `A`. `Argument_Error` is raised if the matrix `A` is not Hermitian.

147/2

```
function Unit_Matrix (Order          : Positive;
                     First_1, First_2 : Integer := 1)
                    return Complex_Matrix;
```

148/2

This function returns a square <unit matrix> with `Order**2` components and lower bounds of `First_1` and `First_2` (for the first and second index ranges respectively). All components are set to (0.0, 0.0) except for the main diagonal, whose components are set to (1.0, 0.0). `Constraint_Error` is raised if `First_1 + Order - 1 > Integer'Last` or `First_2 + Order - 1 > Integer'Last`.

Implementation Requirements

149/2

Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.

150/2

For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real_Base and Complex in both the strict mode and the relaxed mode (see Section 21.2 [G.2], page 1103).

151/2

For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product $\langle X \rangle * \langle Y \rangle$ shall not exceed $\langle g \rangle * \text{abs}(\langle X \rangle) * \text{abs}(\langle Y \rangle)$ where $\langle g \rangle$ is defined as

152/2

$$\langle g \rangle = \langle X \rangle' \text{Length} * \text{Real}' \text{Machine_Radix}^{**}(1 - \text{Real}' \text{Model_Mantissa})$$

for mixed complex and real operands

153/2

$$\langle g \rangle = \text{sqrt}(2.0) * \langle X \rangle' \text{Length} * \text{Real}' \text{Machine_Radix}^{**}(1 - \text{Real}' \text{Model_Mantissa})$$

for two complex operands

154/2

For the L2–norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed $\langle g \rangle / 2.0 + 3.0 * \text{Real}' \text{Model_Epsilon}$ where $\langle g \rangle$ has the definition appropriate for two complex operands.

Documentation Requirements

155/2

Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

Implementation Permissions

156/2

The nongeneric equivalent packages may, but need not, be actual instantiations of the generic package for the appropriate predefined type.

157/2

Although many operations are defined in terms of operations from numerics.-generic_complex_types, they need not be implemented by calling those operations provided that the effect is the same.

Implementation Advice

158/2

Implementations should implement the Solve and Inverse functions using established techniques. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done then it should be documented.

159/2

It is not the intention that any special provision should be made to determine whether a

matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise `Constraint_Error` is sufficient.

160/2

The test that a matrix is Hermitian should use the equality operator to compare the real components and negation followed by equality to compare the imaginary components (see Section 21.2.1 [G.2.1], page 1104).

161/2

Implementations should not perform operations on mixed complex and real operands by first converting the real operand to complex. See Section 21.1.1 [G.1.1], page 1084.

22 Annex H High Integrity Systems

1/2

This Annex addresses requirements for high integrity systems (including safety–critical systems and security–critical systems). It provides facilities and specifies documentation requirements that relate to several needs:

2

- Understanding program execution;

3

- Reviewing object code;

4

- Restricting language constructs whose usage might complicate the demonstration of program correctness

4.1

Execution understandability is supported by pragma `Normalize_Scalars`, and also by requirements for the implementation to document the effect of a program in the presence of a bounded error or where the language rules leave the effect unspecified.

5

The pragmas `Reviewable` and `Restrictions` relate to the other requirements addressed by this Annex.

NOTES

6

- 1 The `Valid` attribute (see Section 14.9.2 [13.9.2], page 524) is also useful in addressing these needs, to avoid problems that could otherwise arise from scalars that have values outside their declared range constraints.

22.1 H.1 Pragma `Normalize_Scalars`

1

This pragma ensures that an otherwise uninitialized scalar object is set to a predictable value, but out of range if possible.

Syntax

2

The form of a pragma `Normalize_Scalars` is as follows:

3

```
pragma Normalize_Scalars;
```

Post-Compilation Rules

4

Pragma `NormalizeScalars` is a configuration pragma. It applies to all `compilation_units` included in a partition.

Documentation Requirements

5/2

If a pragma `NormalizeScalars` applies, the implementation shall document the implicit initial values for scalar subtypes, and shall identify each case in which such a value is used and is not an invalid representation.

Implementation Advice

6/2

Whenever possible, the implicit initial values for a scalar subtype should be an invalid representation (see Section 14.9.1 [13.9.1], page 522).

NOTES

7

2 The initialization requirement applies to uninitialized scalar objects that are subcomponents of composite objects, to allocated objects, and to stand-alone objects. It also applies to scalar out parameters. Scalar subcomponents of composite out parameters are initialized to the corresponding part of the actual, by virtue of Section 7.4.1 [6.4.1], page 270.

8

3 The initialization requirement does not apply to a scalar for which pragma `Import` has been specified, since initialization of an imported object is performed solely by the foreign language environment (see Section 16.1 [B.1], page 894).

9

4 The use of pragma `NormalizeScalars` in conjunction with `Pragma Restrictions(No_Exceptions)` may result in erroneous execution (see Section 22.4 [H.4], page 1158).

22.2 H.2 Documentation of Implementation Decisions

Documentation Requirements

1

The implementation shall document the range of effects for each situation that the language rules identify as either a bounded error or as having an unspecified effect. If the implementation can constrain the effects of erroneous execution for a given construct, then it shall document such constraints. The documentation might be provided either independently of any compilation unit or partition, or as part of an annotated listing for a given unit or partition. See also Section 2.1.3 [1.1.3], page 23, and Section 2.1.2 [1.1.2], page 20.

NOTES

2

5 Among the situations to be documented are the conventions chosen for parameter passing, the methods used for the management of run-time storage, and the method used to evaluate numeric expressions if this involves extended range or extra precision.

22.3 H.3 Reviewable Object Code

1

Object code review and validation are supported by pragmas `Reviewable` and `Inspection_Point`.

22.3.1 H.3.1 Pragma `Reviewable`

1

This pragma directs the implementation to provide information to facilitate analysis and review of a program's object code, in particular to allow determination of execution time and storage usage and to identify the correspondence between the source and object programs.

Syntax

2

The form of a pragma `Reviewable` is as follows:

3

```
pragma Reviewable;  
Post-Compilation Rules
```

4

Pragma `Reviewable` is a configuration pragma. It applies to all `compilation_units` included in a partition.

Implementation Requirements

5

The implementation shall provide the following information for any compilation unit to which such a pragma applies:

6

- Where compiler-generated run-time checks remain;

7

- An identification of any construct with a language-defined check that is recognized prior to run time as certain to fail if executed (even if the generation of run-time checks has been suppressed);

8/2

- For each read of a scalar object, an identification of the read as either "known to be initialized," or "possibly uninitialized," independent of whether pragma NormalizeScalars applies;

9

- Where run-time support routines are implicitly invoked;

10

- An object code listing, including:

11

- Machine instructions, with relative offsets;

12

- Where each data object is stored during its lifetime;

13

- Correspondence with the source program, including an identification of the code produced per declaration and per statement.

14

- An identification of each construct for which the implementation detects the possibility of erroneous execution;

15

- For each subprogram, block, task, or other construct implemented by reserving and subsequently freeing an area on a run-time stack, an identification of the length of the fixed-size portion of the area and an indication of whether the non-fixed size portion is reserved on the stack or in a dynamically-managed storage region.

16

The implementation shall provide the following information for any partition to which the pragma applies:

17

- An object code listing of the entire partition, including initialization and finalization code as well as run-time system components, and with an identification of those instructions and data that will be relocated at load time;

18

- A description of the run–time model relevant to the partition.

18.1

The implementation shall provide control– and data–flow information, both within each compilation unit and across the compilation units of the partition.

Implementation Advice

19

The implementation should provide the above information in both a human–readable and machine–readable form, and should document the latter so as to ease further processing by automated tools.

20

Object code listings should be provided both in a symbolic format and also in an appropriate numeric format (such as hexadecimal or octal).

NOTES

21

6 The order of elaboration of library units will be documented even in the absence of pragma Reviewable (see Section 11.2 [10.2], page 409).

22.3.2 H.3.2 Pragma Inspection_Point

1

An occurrence of a pragma `Inspection_Point` identifies a set of objects each of whose values is to be available at the point(s) during program execution corresponding to the position of the pragma in the compilation unit. The purpose of such a pragma is to facilitate code validation.

Syntax

2

The form of a pragma `Inspection_Point` is as follows:

3

```
pragma Inspection_Point[(<object_>name {, <object_>name})];
```

Legality Rules

4

A pragma `Inspection_Point` is allowed wherever a `declarative_item` or `statement` is allowed. Each `<object_>name` shall statically denote the declaration of an object.

Static Semantics

5/2

An `<inspection point>` is a point in the object code corresponding to the occurrence of a pragma `Inspection_Point` in the compilation unit. An object is `<inspectable>` at an inspection point if the corresponding pragma `Inspection_Point` either has an argument denoting

that object, or has no arguments and the declaration of the object is visible at the inspection point.

Dynamic Semantics

6

Execution of a pragma `Inspection_Point` has no effect.

Implementation Requirements

7

Reaching an inspection point is an external interaction with respect to the values of the inspectable objects at that point (see Section 2.1.3 [1.1.3], page 23).

Documentation Requirements

8

For each inspection point, the implementation shall identify a mapping between each inspectable object and the machine resources (such as memory locations or registers) from which the object's value can be obtained.

NOTES

9/2

7 The implementation is not allowed to perform "dead store elimination" on the last assignment to a variable prior to a point where the variable is inspectable. Thus an inspection point has the effect of an implicit read of each of its inspectable objects.

10

8 Inspection points are useful in maintaining a correspondence between the state of the program in source code terms, and the machine state during the program's execution. Assertions about the values of program objects can be tested in machine terms at inspection points. Object code between inspection points can be processed by automated tools to verify programs mechanically.

11

9 The identification of the mapping from source program objects to machine resources is allowed to be in the form of an annotated object listing, in human-readable or tool-processable form.

22.4 H.4 High Integrity Restrictions

1

This clause defines restrictions that can be used with pragma Restrictions (see Section 14.12 [13.12], page 535); these facilitate the demonstration of program correctness by allowing tailored versions of the run-time system.

Static Semantics

2/2

<This paragraph was deleted.>

3/2

The following <restriction_>identifiers are language defined:

4

Tasking-related restriction:

5

No_Protected_Types

There are no declarations of protected types or protected objects.

6

Memory-management related restrictions:

7

No_Allocators

There are no occurrences of an allocator.

8/1

No_Local_Allocators

Allocators are prohibited in subprograms, generic subprograms, tasks, and entry bodies.

9/2

<This paragraph was deleted.>

10

Immediate_Reclamation

Except for storage occupied by objects created by allocators and not deallocated via unchecked deallocation, any storage reserved at run time for an object is immediately reclaimed when the object no longer exists.

11

Exception-related restriction:

12

No_Exceptions

Raise-statements and exception_handlers are not allowed. No language-defined run-time checks are generated; however, a run-time check performed automatically by the hardware is permitted.

13

Other restrictions:

14

No_Floating_Point

Uses of predefined floating point types and operations, and declarations of new floating point types, are not allowed.

15

No_Fixed_Point

Uses of predefined fixed point types and operations, and declarations of new fixed point types, are not allowed.

16/2

<This paragraph was deleted.>

17

No_Access_Subprograms

The declaration of access-to-subprogram types is not allowed. ■

18

No_Unchecked_Access

The Unchecked_Access attribute is not allowed. ■

19

No_Dispatch

Occurrences of T'Class are not allowed, for any (tagged) subtype T.

20/2

No_IO

Semantic dependence on any of the library units Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, or Stream_IO is not allowed.

21

No_Delay

Delay_Statements and semantic dependence on package Calendar are not allowed.

22

No_Recursion

As part of the execution of a subprogram, the same subprogram is not invoked.

23

No_Reentrancy

During the execution of a subprogram by a task, no other task invokes the same subprogram.

Implementation Requirements

23.1/2

An implementation of this Annex shall support:

23.2/2

- the restrictions defined in this subclause; and

23.3/2

- the following restrictions defined in Section 18.7 [D.7], page 1001: `No_Task_Hierarchy`, `No_Abort_Statement`, `No_Implicit_Heap_Allocation`; and

23.4/2

- the pragma `Profile(Ravenscar)`; and

23.5/2

- the following uses of `<restriction_parameter_>identifiers` defined in Section 18.7 [D.7], page 1001, which are checked prior to program execution:

23.6/2

- `Max_Task_Entries` => 0,

23.7/2

- `Max_Asynchronous_Select_Nesting` => 0, and

23.8/2

- `Max_Tasks` => 0.

24

If an implementation supports pragma `Restrictions` for a particular argument, then except for the restrictions `No_Unchecked_Deallocation`, `No_Unchecked_Conversion`, `No_Access_Subprograms`, and `No_Unchecked_Access`, the associated restriction applies to the run–time system.

Documentation Requirements

25

If a pragma `Restrictions(No_Exceptions)` is specified, the implementation shall document the effects of all constructs where language–defined checks are still performed automatically (for example, an overflow check performed by the processor).

Erroneous Execution

26

Program execution is erroneous if pragma `Restrictions(No_Exceptions)` has been specified and the conditions arise under which a generated language–defined run–time check would fail.

27

Program execution is erroneous if pragma `Restrictions(No_Recursion)` has been specified and a subprogram is invoked as part of its own execution, or if pragma `Restrictions(No_Reentrancy)` has been specified and during the execution of a subprogram by a task, another task invokes the same subprogram.

NOTES

28/2

10 Uses of `<restriction_parameter>identifier No_Dependence` defined in Section 14.12.1 [13.12.1], page 536: `No_Dependence => Ada.Unchecked_Deallocation` and `No_Dependence => Ada.Unchecked_Conversion` may be appropriate for high-integrity systems. Other uses of `No_Dependence` can also be appropriate for high-integrity systems.

22.5 H.5 Pragma Detect_Blocking

1/2

The following pragma forces an implementation to detect potentially blocking operations within a protected operation.

Syntax

2/2

The form of a pragma `Detect_Blocking` is as follows:

3/2

```
pragma Detect_Blocking;  
Post-Compilation Rules
```

4/2

A pragma `Detect_Blocking` is a configuration pragma.

Dynamic Semantics

5/2

An implementation is required to detect a potentially blocking operation within a protected operation, and to raise `Program_Error` (see Section 10.5.1 [9.5.1], page 344).

Implementation Permissions

6/2

An implementation is allowed to reject a compilation_unit if a potentially blocking operation is present directly within an entry_body or the body of a protected subprogram.

NOTES

7/2

11 An operation that causes a task to be blocked within a foreign language domain is not defined to be potentially blocking, and need not be detected.

22.6 H.6 Pragma Partition_Elaboration_Policy

1/2

This clause defines a pragma for user control over elaboration policy.

Syntax

2/2

The form of a pragma `Partition_Elaboration_Policy` is as follows:

3/2

pragma Partition_Elaboration_Policy (<policy_>identifier);

4/2

The <policy_>identifier shall be either Sequential, Concurrent or an implementation–defined identifier.

Post-Compilation Rules

5/2

A pragma Partition_Elaboration_Policy is a configuration pragma. It specifies the elaboration policy for a partition. At most one elaboration policy shall be specified for a partition.

6/2

If the Sequential policy is specified for a partition then pragma Restrictions (No_Task_Hierarchy) shall also be specified for the partition.

Dynamic Semantics

7/2

Notwithstanding what this International Standard says elsewhere, this pragma allows partition elaboration rules concerning task activation and interrupt attachment to be changed. If the <policy_>identifier is Concurrent, or if there is no pragma Partition_Elaboration_Policy defined for the partition, then the rules defined elsewhere in this Standard apply.

8/2

If the partition elaboration policy is Sequential, then task activation and interrupt attachment are performed in the following sequence of steps:

9/2

- The activation of all library–level tasks and the attachment of interrupt handlers are deferred until all library units are elaborated.

10/2

- The interrupt handlers are attached by the environment task.

11/2

- The environment task is suspended while the library–level tasks are activated.

12/2

- The environment task executes the main subprogram (if any) concurrently with these executing tasks.

13/2

If several dynamic interrupt handler attachments for the same interrupt are deferred, then the most recent call of Attach_Handler or Exchange_Handler determines which handler is attached.

14/2

If any deferred task activation fails, `Tasking_Error` is raised at the beginning of the sequence of statements of the body of the environment task prior to calling the main subprogram.

Implementation Advice

15/2

If the partition elaboration policy is `Sequential` and the `Environment` task becomes permanently blocked during elaboration then the partition is deadlocked and it is recommended that the partition be immediately terminated.

Implementation Permissions

16/2

If the partition elaboration policy is `Sequential` and any task activation fails then an implementation may immediately terminate the active partition to mitigate the hazard posed by continuing to execute with a subset of the tasks being active.

NOTES

17/2

12 If any deferred task activation fails, the environment task is unable to handle the `Tasking_Error` exception and completes immediately. By contrast, if the partition elaboration policy is `Concurrent`, then this exception could be handled within a library unit.

23 Annex J Obsolescent Features

1/2

This Annex contains descriptions of features of the language whose functionality is largely redundant with other features defined by this International Standard. Use of these features is not recommended in newly written programs. Use of these features can be prevented by using pragma Restrictions (No_Obsolescent_Features), see Section 14.12.1 [13.12.1], page 536.

23.1 J.1 Renamings of Ada 83 Library Units

Static Semantics

1

The following library_unit_renaming_declarations exist:

2

```
with Ada.Unchecked_Conversion;
generic function Unchecked_Conversion renames Ada.Unchecked_Conversion;■
```

3

```
with Ada.Unchecked_Deallocation;
generic procedure Unchecked_Deallocation renames Ada.Unchecked_Deallocation;■
```

4

```
with Ada.Sequential_IO;
generic package Sequential_IO renames Ada.Sequential_IO;
```

5

```
with Ada.Direct_IO;
generic package Direct_IO renames Ada.Direct_IO;
```

6

```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```

7

```
with Ada.IO_Exceptions;
package IO_Exceptions renames Ada.IO_Exceptions;
```

8

```
with Ada.Calendar;
package Calendar renames Ada.Calendar;
```

9

```
with System.Machine_Code;  
package Machine_Code renames System.Machine_Code; --< If supported.>■
```

Implementation Requirements

10

The implementation shall allow the user to replace these renamings.

23.2 J.2 Allowed Replacements of Characters

Syntax

1

The following replacements are allowed for the vertical line, number sign, and quotation mark characters:

2

- A vertical line character (|) can be replaced by an exclamation mark (!) where used as a delimiter.

3

- The number sign characters (#) of a based_literal can be replaced by colons (:) provided that the replacement is done for both occurrences.

4

- The quotation marks (") used as string brackets at both ends of a string literal can be replaced by percent signs (%) provided that the enclosed sequence of characters contains no quotation mark, and provided that both string brackets are replaced. Any percent sign within the sequence of characters shall then be doubled and each such doubled percent sign is interpreted as a single percent sign character value.

5

These replacements do not change the meaning of the program.

23.3 J.3 Reduced Accuracy Subtypes

1

A digits_constraint may be used to define a floating point subtype with a new value for its requested decimal precision, as reflected by its Digits attribute. Similarly, a delta_constraint may be used to define an ordinary fixed point subtype with a new value for its <delta>, as reflected by its Delta attribute.

Syntax

2

$\text{delta_constraint} ::= \text{delta } \langle \text{static_} \rangle \text{expression } [\text{range_constraint}]$
Name Resolution Rules

3

The expression of a `delta_constraint` is expected to be of any real type.

Legality Rules

4

The expression of a `delta_constraint` shall be static.

5

For a `subtype_indication` with a `delta_constraint`, the `subtype_mark` shall denote an ordinary fixed point subtype.

6

For a `subtype_indication` with a `digits_constraint`, the `subtype_mark` shall denote either a decimal fixed point subtype or a floating point subtype (notwithstanding the rule given in Section 4.5.9 [3.5.9], page 106, that only allows a decimal fixed point subtype).

Static Semantics

7

A `subtype_indication` with a `subtype_mark` that denotes an ordinary fixed point subtype and a `delta_constraint` defines an ordinary fixed point subtype with a `<delta>` given by the value of the expression of the `delta_constraint`. If the `delta_constraint` includes a `range_constraint` (see [S0036], page 76), then the ordinary fixed point subtype is constrained by the `range_constraint` (see [S0036], page 76).

8

A `subtype_indication` with a `subtype_mark` that denotes a floating point subtype and a `digits_constraint` defines a floating point subtype with a requested decimal precision (as reflected by its `Digits` attribute) given by the value of the expression of the `digits_constraint`. If the `digits_constraint` includes a `range_constraint` (see [S0036], page 76), then the floating point subtype is constrained by the `range_constraint` (see [S0036], page 76).

Dynamic Semantics

9

A `delta_constraint` is `<compatible>` with an ordinary fixed point subtype if the value of the expression is no less than the `<delta>` of the subtype, and the `range_constraint`, if any, is compatible with the subtype.

10

A `digits_constraint` is `<compatible>` with a floating point subtype if the value of the expression is no greater than the requested decimal precision of the subtype, and the `range_constraint`, if any, is compatible with the subtype.

11

The elaboration of a `delta_constraint` consists of the elaboration of the `range_constraint`, if any.

23.4 J.4 The Constrained Attribute

Static Semantics

1

For every private subtype S, the following attribute is defined:

2

S'Constrained

Yields the value
False if S denotes
an unconstrained
nonformal private
subtype with
discriminants; also
yields the value
False if S denotes a
generic formal private
subtype, and the
associated actual
subtype is either
an unconstrained
subtype with
discriminants or an
unconstrained array
subtype; yields the
value True otherwise.
The value of this
attribute is of the
predefined subtype
Boolean.

23.5 J.5 ASCII

Static Semantics

1

The following declaration exists in the declaration of package Standard:

2

```
package ASCII is
```

3

```
--< Control characters:>
```

4

```
NUL   : constant Character := <nul>;      SOH   : constant Character := <soh>;■  
STX   : constant Character := <stx>;      ETX   : constant Character := <etx>;■
```

```

EOT   : constant Character := <eot>;      ENQ   : constant Character := <enq>;█
ACK   : constant Character := <ack>;      BEL   : constant Character := <bel>;█
BS    : constant Character := <bs>;       HT    : constant Character := <ht>;█
LF    : constant Character := <lf>;       VT    : constant Character := <vt>;█
FF    : constant Character := <ff>;       CR    : constant Character := <cr>;█
SO    : constant Character := <so>;       SI    : constant Character := <si>;█
DLE   : constant Character := <dle>;      DC1   : constant Character := <dc1>;█
DC2   : constant Character := <dc2>;      DC3   : constant Character := <dc3>;█
DC4   : constant Character := <dc4>;      NAK   : constant Character := <nak>;█
SYN   : constant Character := <syn>;      ETB   : constant Character := <etb>;█
CAN   : constant Character := <can>;      EM    : constant Character := <em>;█
SUB   : constant Character := <sub>;      ESC   : constant Character := <esc>;█
FS    : constant Character := <fs>;       GS    : constant Character := <gs>;█
RS    : constant Character := <rs>;       US    : constant Character := <us>;█
DEL   : constant Character := <del>;

```

5

```
--< Other characters:>
```

6

```

Exclam  : constant Character:= '!' ;      Quotation : constant Character:= '"';█
Sharp   : constant Character:= '#';      Dollar    : constant Character:= '$';█
Percent : constant Character:= '%';      Ampersand : constant Character:= '&';█
Colon   : constant Character:= ':';      Semicolon : constant Character:= ';';█
Query   : constant Character:= '?';      At_Sign   : constant Character:= '@';█
L_Bracket: constant Character:= '[';      Back_Slash: constant Character:= '\';█
R_Bracket: constant Character:= ']';      Circumflex: constant Character:= '^';█
Underline: constant Character:= '_';      Grave     : constant Character:= '`';█
L_Brace  : constant Character:= '{';      Bar       : constant Character:= '|';█
R_Brace  : constant Character:= '}';      Tilde     : constant Character:= '~';█

```

7

```
--< Lower case letters:>
```

8

```

LC_A: constant Character:= 'a';
...
LC_Z: constant Character:= 'z';

```

9

```
end ASCII;
```

23.6 J.6 Numeric_Error

Static Semantics

1

The following declaration exists in the declaration of package Standard:

2

```
Numeric_Error : exception renames Constraint_Error;
```

23.7 J.7 At Clauses

Syntax

1

```
at_clause ::= for direct_name use at expression;
```

Static Semantics

2

An `at_clause` of the form "for <x> use at <y>;" is equivalent to an `attribute_definition_clause` of the form "for <x>'Address use <y>";".

23.7.1 J.7.1 Interrupt Entries

1

Implementations are permitted to allow the attachment of task entries to interrupts via the address clause. Such an entry is referred to as an <interrupt entry>.

2

The address of the task entry corresponds to a hardware interrupt in an implementation-defined manner. (See `Ada.Interrupts.Reference` in Section 17.3.2 [C.3.2], page 957.)

Static Semantics

3

The following attribute is defined:

4

For any task entry X:

5

X'Address

For a task entry whose address is specified (an <interrupt entry>), the value refers to the corresponding hardware interrupt. For such an entry, as for any other task entry, the meaning of this value is implementation defined. The value of

this attribute is of the type of the subtype System.Address.

6

Address may be specified for single entries via an attribute_definition_clause.

Dynamic Semantics

7

As part of the initialization of a task object, the address clause for an interrupt entry is elaborated, which evaluates the expression of the address clause. A check is made that the address specified is associated with some interrupt to which a task entry may be attached. If this check fails, Program_Error is raised. Otherwise, the interrupt entry is attached to the interrupt associated with the specified address.

8

Upon finalization of the task object, the interrupt entry, if any, is detached from the corresponding interrupt and the default treatment is restored.

9

While an interrupt entry is attached to an interrupt, the interrupt is reserved (see Section 17.3 [C.3], page 951).

10

An interrupt delivered to a task entry acts as a call to the entry issued by a hardware task whose priority is in the System.Interrupt_Priority range. It is implementation defined whether the call is performed as an ordinary entry call, a timed entry call, or a conditional entry call; which kind of call is performed can depend on the specific interrupt.

Bounded (Run-Time) Errors

11

It is a bounded error to evaluate E'Caller (see Section 17.7.1 [C.7.1], page 965) in an accept_statement for an interrupt entry. The possible effects are the same as for calling Current_Task from an entry body.

Documentation Requirements

12

The implementation shall document to which interrupts a task entry may be attached.

13

The implementation shall document whether the invocation of an interrupt entry has the effect of an ordinary entry call, conditional call, or a timed call, and whether the effect varies in the presence of pending interrupts.

Implementation Permissions

14

The support for this subclause is optional.

15

Interrupts to which the implementation allows a task entry to be attached may be designated

as reserved for the entire duration of program execution; that is, not just when they have an interrupt entry attached to them.

16/1

Interrupt entry calls may be implemented by having the hardware execute directly the appropriate `accept_statement`. Alternatively, the implementation is allowed to provide an internal interrupt handler to simulate the effect of a normal task calling the entry.

17

The implementation is allowed to impose restrictions on the specifications and bodies of tasks that have interrupt entries.

18

It is implementation defined whether direct calls (from the program) to interrupt entries are allowed.

19

If a `select_statement` contains both a `terminate_alternative` and an `accept_alternative` for an interrupt entry, then an implementation is allowed to impose further requirements for the selection of the `terminate_alternative` in addition to those given in Section 10.3 [9.3], page 335.

NOTES

20/1

1 Queued interrupts correspond to ordinary entry calls. Interrupts that are lost if not immediately processed correspond to conditional entry calls. It is a consequence of the priority rules that an `accept_statement` executed in response to an interrupt can be executed with the active priority at which the hardware generates the interrupt, taking precedence over lower priority tasks, without a scheduling action.

21

2 Control information that is supplied upon an interrupt can be passed to an associated interrupt entry as one or more parameters of mode in.

Examples

22

<Example of an interrupt entry:>

23

```
task Interrupt_Handler is
  entry Done;
  for Done'Address use Ada.Interrupts.Reference(Ada.Interrupts.Names.Device_Done);
end Interrupt_Handler;
```

23.8 J.8 Mod Clauses

Syntax

1

```
mod_clause ::= at mod <static_>expression;  
            Static Semantics
```

2

A record_representation_clause of the form:

3

```
for <r> use  
  record at mod <a>  
    ...  
  end record;
```

4

is equivalent to:

5

```
for <r>'Alignment use <a>;  
for <r> use  
  record  
    ...  
  end record;
```

23.9 J.9 The Storage_Size Attribute

Static Semantics

1

For any task subtype T, the following attribute is defined:

2

T'Storage_Size

Denotes an implementation-defined value of type <universal_integer> representing the number of storage elements reserved for a task of the subtype T.

3/2

Storage_Size may be specified for a

task first subtype
that is not an
interface via an at-
tribute_definition_clause. ■

23.10 J.10 Specific Suppression of Checks

1/2

Pragma Suppress can be used to suppress checks on specific entities.

Syntax

2/2

The form of a specific Suppress pragma is as follows:

3/2

```
pragma Suppress(identifier, [On =>] name);
```

Legality Rules

4/2

The identifier shall be the name of a check (see Section 12.5 [11.5], page 431). The name shall statically denote some entity.

5/2

For a specific Suppress pragma that is immediately within a package_specification, the name shall denote an entity (or several overloaded subprograms) declared immediately within the package_specification (see [S0174], page 279).

Static Semantics

6/2

A specific Suppress pragma applies to the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a package_specification, to the end of the scope of the named entity. The pragma applies only to the named entity, or, for a subtype, on objects and values of its type. A specific Suppress pragma suppresses the named check for any entities to which it applies (see Section 12.5 [11.5], page 431). Which checks are associated with a specific entity is not defined by this International Standard.

Implementation Permissions

7/2

An implementation is allowed to place restrictions on specific Suppress pragmas.

NOTES

8/2

3 An implementation may support a similar On parameter on pragma Unsuppress (see Section 12.5 [11.5], page 431).

23.11 J.11 The Class Attribute of Untagged Incomplete Types

Static Semantics

1/2

For the first subtype S of a type <T> declared by an `incomplete_type_declaration` that is not tagged, the following attribute is defined:

2/2

S'Class

Denotes the first subtype of the incomplete class-wide type rooted at <T>. The completion of <T> shall declare a tagged type. Such an attribute reference shall occur in the same library unit as the `incomplete_type_declaration`.

23.12 J.12 Pragma Interface

Syntax

1/2

In addition to an identifier, the reserved word `interface` is allowed as a pragma name, to provide compatibility with a prior edition of this International Standard.

23.13 J.13 Dependence Restriction Identifiers

1/2

The following restrictions involve dependence on specific language-defined units. The more general restriction `No_Dependence` (see Section 14.12.1 [13.12.1], page 536) should be used for this purpose.

Static Semantics

2/2

The following <restriction_>identifiers exist:

3/2

`No_Asynchronous_Control`

Semantic dependence on the predefined package `Asynchronous_Task_Control` is not allowed.

4/2

No_Unchecked_Conversion

Semantic dependence
on the predefined
generic function
Unchecked_Conversion
is not allowed. ■

5/2

No_Unchecked_Deallocation

Semantic dependence
on the predefined
generic procedure
Unchecked_Deallocation
is not allowed. ■

23.14 J.14 Character and Wide_Character Conversion Functions

Static Semantics

1/2

The following declarations exist in the declaration of package Ada.Characters.Handling:

2/2

```
function Is_Character (Item : in Wide_Character) return Boolean
renames Conversions.Is_Character;
function Is_String (Item : in Wide_String) return Boolean
renames Conversions.Is_String;
```

3/2

```
function To_Character (Item : in Wide_Character;
Substitute : in Character := ' ')
return Character
renames Conversions.To_Character;
```

4/2

```
function To_String (Item : in Wide_String;
Substitute : in Character := ' ')
return String
renames Conversions.To_String;
```

5/2

```
function To_Wide_Character (Item : in Character) return Wide_Character
renames Conversions.To_Wide_Character; ■
```

6/2

```
function To_Wide_String (Item : in String) return Wide_String
renames Conversions.To_Wide_String;
```

24 Annex K Language-Defined Attributes

1

This annex summarizes the definitions given elsewhere of the language-defined attributes.

2

P'Access

For a prefix P that denotes a subprogram:

3

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (<S>), as determined by the expected type. See Section 4.10.2 [3.10.2], page 164.

4

X'Access

For a prefix X that denotes an aliased view of an object:

5

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. See Section 4.10.2 [3.10.2], page 164.

6/1

X'Address

For a prefix X that denotes an object, program unit, or label:

7

Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address. See Section 14.3 [13.3], page 486.

8

S'Adjacent

For every subtype S of a floating point type <T>:

9

S'Adjacent denotes a function with the following specification:

10

```
function S'Adjacent (<X>, <Towards> : <T>)■  
    return <T>
```

11

If <Towards> = <X>, the function yields <X>; otherwise, it yields the machine number of the type <T> adjacent to <X> in the direction of <Towards>, if that machine number

exists. If the result would be outside the base range of S, Constraint_Error is raised. When <T>'Signed_Zeros is True, a zero result has the sign of <X>. When <Towards> is zero, its sign has no bearing on the result. See Section 15.5.3 [A.5.3], page 663.

12
S'Aft

For every fixed point subtype S:

13

S'Aft yields the number of decimal digits needed after the decimal point to accommodate the <delta> of the subtype S, unless the <delta> of the subtype S is greater than 0.1, in which case the attribute yields the value one. (S'Aft is the smallest positive integer N for which $(10^{**N}) * S'Delta$ is greater than or equal to one.) The value of this attribute is of the type <universal_integer>. See Section 4.5.10 [3.5.10], page 109.

13.1/2
S'Alignment

For every subtype S:

13.2/2

The value of this attribute is of type `<universal_integer>`, and nonnegative.

13.3/2

For an object `X` of subtype `S`, if `S'Alignment` is not zero, then `X'Alignment` is a nonzero integral multiple of `S'Alignment` unless specified otherwise by a representation item. See Section 14.3 [13.3], page 486.

14/1

`X'Alignment`

For a prefix `X` that denotes an object:

15

The value of this attribute is of type `<universal_integer>`, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If `X'Alignment` is not zero, then `X` is aligned on a storage unit boundary and `X'Address` is an integral multiple of `X'Alignment` (that is, the `Address` modulo the `Alignment` is zero).

16/2

<This paragraph was deleted.> See Section 14.3 [13.3], page 486.

17

S'Base

For every scalar subtype S:

18

S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the <base subtype> of the type. See Section 4.5 [3.5], page 76.

19

S'Bit_Order

For every specific record subtype S:

20

Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. See Section 14.5.3 [13.5.3], page 508.

21/1

P'Body_Version

For a prefix P that statically denotes a program unit:

22

Yields a value of the predefined type String that identifies the version of the

compilation unit that contains the body (but not any subunits) of the program unit. See Section 19.3 [E.3], page 1043.

23
T'Callable

For a prefix T that is of a task type (after any implicit dereference):

24

Yields the value True when the task denoted by T is <callable>, and False otherwise; See Section 10.9 [9.9], page 388.

25
E'Caller

For a prefix E that denotes an entry_declaration:

26

Yields a value of the type Task_Id that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an entry_body or accept_statement corresponding to the entry_declaration denoted by E. See Section 17.7.1 [C.7.1], page 965.

27
S'Ceiling

For every subtype S of
a floating point type
<T>:

28

S'Ceiling denotes
a function with
the following
specification:

29

```
function S'Ceiling (<X> : <T>)
    return <T>
```

30

The function
yields the value
ceiling(<X>), i.e.,
the smallest (most
negative) integral
value greater than or
equal to <X>. When
<X> is zero, the result
has the sign of <X>; a
zero result otherwise
has a negative sign
when S'Signed_Zeros
is True. See
Section 15.5.3 [A.5.3],
page 663.

31

S'Class

For every subtype
S of a tagged type
<T> (specific or
class-wide):

32

S'Class denotes
a subtype of the
class-wide type
(called <T>'Class in
this International
Standard) for the
class rooted at <T>

(or if S already denotes a class-wide subtype, then S'Class is the same as S).

33

S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type <T> belong to S. See Section 4.9 [3.9], page 136.

34

S'Class

For every subtype S of an untagged private type whose full view is tagged:

35

Denotes the class-wide subtype corresponding to the full view of S. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. After the full view, the Class attribute of the full view can be used. See Section 8.3.1 [7.3.1], page 287.

36/1

X'Component_Size

For a prefix X that denotes an array subtype or array

object (after any implicit dereference):

37

Denotes the size in bits of components of the type of X. The value of this attribute is of type `<universal_integer>`. See Section 14.3 [13.3], page 486.

38

S'Compose

For every subtype S of a floating point type `<T>`:

39

S'Compose denotes a function with the following specification:

40

```
function S'Compose (<Fraction> : <T>;  
                  <Exponent> : <universal_integer>)  
    return <T>
```

41

Let `<v>` be the value `<Fraction> · <T>'Machine_Radix<Exponent>-<k>`, where `<k>` is the normalized exponent of `<Fraction>`. If `<v>` is a machine number of the type `<T>`, or if `|<v>| >= <T>'Model_Small`, the function yields `<v>`; otherwise, it yields either one of the machine numbers of the type `<T>` adjacent to `<v>`.

Constraint_Error is optionally raised if <v> is outside the base range of S. A zero result has the sign of <Fraction> when S'Signed_Zeros is True. See Section 15.5.3 [A.5.3], page 663.

42
A'Constrained

For a prefix A that is of a discriminated type (after any implicit dereference):

43

Yields the value True if A denotes a constant, a value, or a constrained variable, and False otherwise. See Section 4.7.2 [3.7.2], page 129.

44
S'Copy_Sign

For every subtype S of a floating point type <T>:

45

S'Copy_Sign denotes a function with the following specification:

46

```
function S'Copy_Sign (<Value>, <Sign> : <T>)
  return <T>
```

47

If the value of <Value> is nonzero,

the function yields a result whose magnitude is that of `<Value>` and whose sign is that of `<Sign>`; otherwise, it yields the value zero. `Constraint_Error` is optionally raised if the result is outside the base range of `S`. A zero result has the sign of `<Sign>` when `S'Signed_Zeros` is `True`. See Section 15.5.3 [A.5.3], page 663.

48
E'Count

For a prefix `E` that denotes an entry of a task or protected unit:

49

Yields the number of calls presently queued on the entry `E` of the current instance of the unit. The value of this attribute is of the type `<universal_integer>`. See Section 10.9 [9.9], page 388.

50/1
S'Definite

For a prefix `S` that denotes a formal indefinite subtype:

51

`S'Definite` yields `True` if the actual subtype corresponding to `S`

is definite; otherwise it yields False. The value of this attribute is of the predefined type Boolean. See Section 13.5.1 [12.5.1], page 462.

52
S'Delta

For every fixed point subtype S:

53

S'Delta denotes the <delta> of the fixed point subtype S. The value of this attribute is of the type <universal_real>. See Section 4.5.10 [3.5.10], page 109.

54
S'Denorm

For every subtype S of a floating point type <T>:

55

Yields the value True if every value expressible in the form
 $\pm \langle \text{mantissa} \rangle \cdot \langle T \rangle' \text{Machine_Radix}^{\langle T \rangle' \text{Machine_Emin}}$

where <mantissa> is a nonzero <T>'Machine_Mantissa-digit fraction in the number base <T>'Machine_Radix, the first digit of which is zero, is a machine number (see Section 4.5.7 [3.5.7],

page 103) of the type `<T>`; yields the value `False` otherwise. The value of this attribute is of the predefined type `Boolean`. See Section 15.5.3 [A.5.3], page 663.

56
S'Digits

For every floating point subtype S:

57

S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type `<universal_integer>`. See Section 4.5.8 [3.5.8], page 105.

58
S'Digits

For every decimal fixed point subtype S:

59

S'Digits denotes the `<digits>` of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type `<universal_integer>`. See Section 4.5.10 [3.5.10], page 109.

60

S'Exponent

For every subtype S of a floating point type <T>:

61

S'Exponent denotes a function with the following specification:

62

```
function S'Exponent (<X> : <T>)
  return <universal_integer>
```

63

The function yields the normalized exponent of <X>. See Section 15.5.3 [A.5.3], page 663.

64

S'External_Tag

For every subtype S of a tagged type <T> (specific or class-wide):

65

S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined.

See Section 4.9.2 [3.9.2], page 145, and Section 14.13.2 [13.13.2], page 540. See Section 14.3 [13.3], page 486.

66/1
A'First

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

67

A'First denotes the lower bound of the first index range; its type is the corresponding index type. See Section 4.6.2 [3.6.2], page 119.

68
S'First

For every scalar subtype S:

69

S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S. See Section 4.5 [3.5], page 76.

70/1
A'First(N)

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

`A'First(N)` denotes the lower bound of the N -th index range; its type is the corresponding index type. See Section 4.6.2 [3.6.2], page 119.

71.1/2
`R.C'First_Bit`

For a component C of a composite, non-array object R :

71.2/2

If the nondefault bit ordering applies to the composite type, and if a `component_clause` specifies the placement of C , denotes the value given for the `first_bit` of the `component_clause`; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C , of the first bit occupied by C . This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type `<universal_integer>`. See Section 14.5.2 [13.5.2], page 506.

72
`S'Floor`

For every subtype S of
a floating point type
<T>:

73

S'Floor denotes
a function with
the following
specification:

74

```
function S'Floor (<X> : <T>)
    return <T>
```

75

The function
yields the value
floor(<X>), i.e.,
the largest (most
positive) integral
value less than or
equal to <X>. When
<X> is zero, the result
has the sign of <X>; a
zero result otherwise
has a positive sign.
See Section 15.5.3
[A.5.3], page 663.

76

S'Fore

For every fixed point
subtype S:

77

S'Fore yields the
minimum number of
characters needed
before the decimal
point for the decimal
representation of any
value of the subtype
S, assuming that
the representation
does not include
an exponent,

but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute is of the type `<universal_integer>`. See Section 4.5.10 [3.5.10], page 109.

78
S'Fraction

For every subtype S of a floating point type `<T>`:

79

S'Fraction denotes a function with the following specification:

80

```
function S'Fraction (<X> : <T>)
  return <T>
```

81

The function yields the value $\langle X \rangle \cdot \langle T \rangle \text{'Machine_Radix}^{-\langle k \rangle}$, where `<k>` is the normalized exponent of `<X>`. A zero result, which can only occur when `<X>` is zero, has the sign of `<X>`. See Section 15.5.3 [A.5.3], page 663. ■

82/1
E'Identity

For a prefix E that denotes an exception:

83

`E'Identity` returns the unique identity of the exception. The type of this attribute is `Exception_Id`. See Section 12.4.1 [11.4.1], page 423.

84

`T'Identity`

For a prefix T that is of a task type (after any implicit dereference):

85

Yields a value of the type `Task_Id` that identifies the task denoted by T. See Section 17.7.1 [C.7.1], page 965.

86

`S'Image`

For every scalar subtype S:

87

`S'Image` denotes a function with the following specification:

88

```
function S'Image(<Arg> : S'Base)
  return String
```

89/2

The function returns an image of the value of <Arg> as a String.

See Section 4.5 [3.5],
page 76.

90
S'Class'Input

For every subtype
S'Class of a
class-wide type
<T>'Class:

91

S'Class'Input
denotes a function
with the following
specification:

92/2

```
function S'Class'Input(  
    <Stream> : not null access Ada.Streams.Root_Stream  
    return <T>'Class
```

93/2

First reads the
external tag from
<Stream> and
determines the
corresponding inter-
nal tag (by calling
Tags.Descendant_Tag(String'Input(<Stream>),
S'Tag) which might
raise Tag_Error --
see Section 4.9 [3.9],
page 136) and then
dispatches to the
subprogram denoted
by the Input attribute
of the specific type
identified by the
internal tag; returns
that result. If
the specific type
identified by the
internal tag is not
covered by <T>'Class
or is abstract,
Constraint_Error

is raised. See
Section 14.13.2
[13.13.2], page 540.

94
S'Input

For every subtype S of
a specific type <T>:

95

S'Input denotes
a function with
the following
specification:

96/2

```
function S'Input(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  return <T>
```

97

S'Input reads and
returns one value
from <Stream>,
using any bounds or
discriminants written
by a corresponding
S'Output to
determine how
much to read. See
Section 14.13.2
[13.13.2], page 540.

98/1
A'Last

For a prefix A that is
of an array type (af-
ter any implicit deref-
erence), or denotes a
constrained array sub-
type:

99

A'Last denotes
the upper bound
of the first index

range; its type is the corresponding index type. See Section 4.6.2 [3.6.2], page 119.

100
S'Last

For every scalar subtype S:

101

S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S. See Section 4.5 [3.5], page 76.

102/1
A'Last(N)

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

103

A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type. See Section 4.6.2 [3.6.2], page 119.

103.1/2
R.C'Last_Bit

For a component C of a composite, non-array object R:

103.2/2

If the nondefault bit ordering applies

to the composite type, and if a `component_clause` specifies the placement of `C`, denotes the value given for the `last_bit` of the `component_clause`; otherwise, denotes the offset, from the start of the first of the storage elements occupied by `C`, of the last bit occupied by `C`. This offset is measured in bits. The value of this attribute is of the type `<universal_integer>`. See Section 14.5.2 [13.5.2], page 506.

104
S'Leading_Part

For every subtype `S` of a floating point type `<T>`:

105

S'Leading_Part denotes a function with the following specification:

106

```
function S'Leading_Part (<X> : <T>;  
                        <Radix_Digits> : <universal_  
return <T>
```

107

Let `<v>` be the value `<T>'Machine_Radix<k>-<Radix_Digits>`, where `<k>` is the normalized exponent of `<X>`. The function yields the value

108

- `floor(<X>/<v>)`
· `<v>`, when
`<X>` is non-
negative and
`<Radix_Digits>`
is positive;

109

- `ceiling(<X>/<v>)`
· `<v>`, when `<X>`
is negative and
`<Radix_Digits>`
is positive.

110

`Constraint_Error`
is raised when
`<Radix_Digits>` is
zero or negative. A
zero result, which
can only occur when
`<X>` is zero, has the
sign of `<X>`. See
Section 15.5.3 [A.5.3],
page 663.

111/1
`A'Length`

For a prefix `A` that is
of an array type (af-
ter any implicit deref-
erence), or denotes a
constrained array sub-
type:

112

`A'Length` denotes
the number of values
of the first index
range (zero for a null
range); its type is
`<universal_integer>`.
See Section 4.6.2
[3.6.2], page 119.

113/1
A'Length(N)

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

114

A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is `<universal_integer>`. See Section 4.6.2 [3.6.2], page 119.

115
S'Machine

For every subtype S of a floating point type `<T>`:

116

S'Machine denotes a function with the following specification:

117

```
function S'Machine (<X> : <T>)
  return <T>
```

118

If `<X>` is a machine number of the type `<T>`, the function yields `<X>`; otherwise, it yields the value obtained by rounding or truncating `<X>` to either one of the adjacent

machine numbers of the type `<T>`. `Constraint_Error` is raised if rounding or truncating `<X>` to the precision of the machine numbers results in a value outside the base range of `S`. A zero result has the sign of `<X>` when `S'Signed_Zeros` is `True`. See Section 15.5.3 [A.5.3], page 663.

119
`S'Machine_Emax`

For every subtype `S` of a floating point type `<T>`:

120

Yields the largest (most positive) value of `<exponent>` such that every value expressible in the canonical form (for the type `<T>`), having a `<mantissa>` of `<T>'Machine_Mantissa` digits, is a machine number (see Section 4.5.7 [3.5.7], page 103) of the type `<T>`. This attribute yields a value of the type `<universal_integer>`. See Section 15.5.3 [A.5.3], page 663.

121
`S'Machine_Emin`

For every subtype `S` of a floating point type `<T>`:

122

Yields the smallest (most negative) value of `<exponent>` such that every value expressible in the canonical form (for the type `<T>`), having a `<mantissa>` of `<T>'Machine_Mantissa` digits, is a machine number (see Section 4.5.7 [3.5.7], page 103) of the type `<T>`. This attribute yields a value of the type `<universal_integer>`. See Section 15.5.3 [A.5.3], page 663.

123

`S'Machine_Mantissa`

For every subtype `S` of a floating point type `<T>`:

124

Yields the largest value of `<p>` such that every value expressible in the canonical form (for the type `<T>`), having a `<p>-digit` `<mantissa>` and an `<exponent>` between `<T>'Machine_Emin` and `<T>'Machine_Emax`, is a machine number (see Section 4.5.7 [3.5.7], page 103) of the type `<T>`. This attribute yields a value of the type `<universal_integer>`.

See Section 15.5.3 [A.5.3], page 663.

125
S'Machine_Overflows

For every subtype S of a floating point type <T>:

126

Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type <T>; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See Section 15.5.3 [A.5.3], page 663.

127
S'Machine_Overflows

For every subtype S of a fixed point type <T>:

128

Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type <T>; yields the value False otherwise. The value of this attribute is of the predefined

type Boolean. See Section 15.5.4 [A.5.4], page 679.

129

S'Machine_Radix

For every subtype S of a floating point type <T>:

130

Yields the radix of the hardware representation of the type <T>. The value of this attribute is of the type <universal_integer>. See Section 15.5.3 [A.5.3], page 663.

131

S'Machine_Radix

For every subtype S of a fixed point type <T>:

132

Yields the radix of the hardware representation of the type <T>. The value of this attribute is of the type <universal_integer>. See Section 15.5.4 [A.5.4], page 679.

132.1/2

S'Machine_Rounding

For every subtype S of a floating point type <T>:

132.2/2

S'Machine_Rounding denotes a function

with the following specification:

132.3/2

```
function S'Machine_Rounding (<X> : <T>)
    return <T>
```

132.4/2

The function yields the integral value nearest to <X>. If <X> lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of <X> when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor. See Section 15.5.3 [A.5.3], page 663.

133

S'Machine_Rounds

For every subtype S of a floating point type <T>:

134

Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type <T>; yields the value False otherwise. The value

of this attribute is of the predefined type Boolean. See Section 15.5.3 [A.5.3], page 663.

135
S'Machine_Rounds

For every subtype S of a fixed point type <T>:

136

Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type <T>; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See Section 15.5.4 [A.5.4], page 679.

137
S'Max

For every scalar subtype S:

138

S'Max denotes a function with the following specification:

139

```
function S'Max(<Left>, <Right> : S'Base)
return S'Base
```

140

The function returns the greater of the

values of the two parameters. See Section 4.5 [3.5], page 76.

140.1/2

S'Max_Size_In_Storage_Elements

For every subtype S:

140.2/2

Denotes the maximum value for Size_In_Storage_Elements that could be requested by the implementation via Allocate for an access type whose designated subtype is S. For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then this accounts for any calls on Allocate that could be performed to provide space for such coextensions. The value of this attribute is of type <universal_integer>. See Section 14.11.1 [13.11.1], page 531.

141

S'Min

For every scalar subtype S:

142

S'Min denotes a function with

the following
specification:

143

```
function S'Min(<Left>, <Right> : S'Base)■  
    return S'Base
```

144

The function returns
the lesser of the
values of the two
parameters. See
Section 4.5 [3.5],
page 76.

144.1/2
S'Mod

For every modular
subtype S:

144.2/2

S'Mod denotes
a function with
the following
specification:

144.3/2

```
function S'Mod (<Arg> : <universal_integer>)■  
    return S'Base
```

144.4/2

This function
returns <Arg> mod
S'Modulus, as a value
of the type of S. See
Section 4.5.4 [3.5.4],
page 95.

145
S'Model

For every subtype S of
a floating point type
<T>:

146

S'Model denotes a function with the following specification:

147

```
function S'Model (<X> : <T>)
  return <T>
```

148

If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see Section 21.2.2 [G.2.2], page 1105, for the definition that applies to implementations supporting the Numerics Annex. See Section 15.5.3 [A.5.3], page 663.

149

S'Model_Emin

For every subtype S of a floating point type <T>:

150

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to the value of <T>'Machine_Emin. See Section 21.2.2 [G.2.2], page 1105, for further requirements that apply to

151
S'Model_Epsilon

implementations supporting the Numerics Annex. The value of this attribute is of the type `<universal_integer>`. See Section 15.5.3 [A.5.3], page 663.

152

For every subtype S of a floating point type `<T>`:

153
S'Model_Mantissa

Yields the value `<T>'Machine_Radix1 - <T>'Model_Mantissa`. The value of this attribute is of the type `<universal_real>`. See Section 15.5.3 [A.5.3], page 663.

154

For every subtype S of a floating point type `<T>`:

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to `ceiling(<d> · log(10) / log(<T>'Machine_Radix)) + 1`, where `<d>` is the requested decimal precision of `<T>`, and less than or equal to the value of

`<T>'Machine_Mantissa`.
See Section 21.2.2 [G.2.2], page 1105, for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type `<universal_integer>`. See Section 15.5.3 [A.5.3], page 663.

155
`S'Model_Small`

For every subtype `S` of a floating point type `<T>`:

156

Yields the value `<T>'Machine_Radix<T>'Model_Emin` – 1. The value of this attribute is of the type `<universal_real>`. See Section 15.5.3 [A.5.3], page 663.

157
`S'Modulus`

For every modular subtype `S`:

158

`S'Modulus` yields the modulus of the type of `S`, as a value of the type `<universal_integer>`. See Section 4.5.4 [3.5.4], page 95.

159
`S'Class'Output`

For every subtype `S'Class` of a

160 class-wide type
<T>'Class:

S'Class'Output
denotes a procedure
with the following
specification:

161/2

```
procedure S'Class'Output(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item>    : in <T>'Class)
```

162/2

First writes the external tag of <Item> to <Stream> (by calling String'Output(<Stream>, Tags.- External_Tag(<Item>'Tag)) -- see Section 4.9 [3.9], page 136) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag. Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S. See Section 14.13.2 [13.13.2], page 540.

163 S'Output

For every subtype S of a specific type <T>:

164

S'Output denotes a procedure with

the following
specification:

165/2

```
procedure S'Output(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item> : in <T>)
```

166

S'Output writes the value of <Item> to <Stream>, including any bounds or discriminants. See Section 14.13.2 [13.13.2], page 540.

167/1

D'Partition_Id

For a prefix D that denotes a library-level declaration, excepting a declaration of or within a declared-pure library unit:

168

Denotes a value of the type <universal_integer> that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see Section 19.2.3 [E.2.3], page 1041) the given partition is the one where the body of D was elaborated. See Section 19.1 [E.1], page 1034.

169
S'Pos

For every discrete
subtype S:

170

S'Pos denotes
a function with
the following
specification:

171

```
function S'Pos(<Arg> : S'Base)
  return <universal_integer>
```

172

This function returns
the position number
of the value of <Arg>,
as a value of type
<universal_integer>.
See Section 4.5.5
[3.5.5], page 99.

172.1/2
R.C'Position

For a component
C of a composite,
non-array object R:

172.2/2

If the nondefault
bit ordering applies
to the composite
type, and if a
component_clause
specifies the place-
ment of C, denotes
the value given for
the position of the
component_clause;
otherwise, denotes
the same value
as R.C'Address –
R'Address. The value

of this attribute
is of the type
<universal_integer>.
See Section 14.5.2
[13.5.2], page 506.

173
S'Pred

For every scalar sub-
type S:

174

S'Pred denotes
a function with
the following
specification:

175

```
function S'Pred(<Arg> : S'Base)
  return S'Base
```

176

For an enumeration
type, the function
returns the value
whose position
number is one less
than that of the
value of <Arg>;
Constraint_Error
is raised if there is
no such value of
the type. For an
integer type, the
function returns the
result of subtracting
one from the value
of <Arg>. For a
fixed point type, the
function returns the
result of subtracting
<small> from the
value of <Arg>. For
a floating point
type, the function
returns the machine

number (as defined in Section 4.5.7 [3.5.7], page 103) immediately below the value of <Arg>; `Constraint_Error` is raised if there is no such machine number. See Section 4.5 [3.5], page 76.

176.1/2
P'Priority

For a prefix P that denotes a protected object:

176.2/2

Denotes a non-aliased component of the protected object P. This component is of type `System.Any_Priority` and its value is the priority of P. `P'Priority` denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P. See Section 18.5.2 [D.5.2], page 998.

177/1
A'Range

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

178

179
S'Range

A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once. See Section 4.6.2 [3.6.2], page 119.

180

For every scalar subtype S:

181/1
A'Range(N)

S'Range is equivalent to the range S'First .. S'Last. See Section 4.5 [3.5], page 76.

182

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

183
S'Class'Read

A'Range(N) is equivalent to the range A'First(N) .. A'Last(N), except that the prefix A is only evaluated once. See Section 4.6.2 [3.6.2], page 119.

184

For every subtype S'Class of a class-wide type <T>'Class:

S'Class'Read denotes a procedure with the following specification:

185/2

```
procedure S'Class'Read(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item> : out <T>'Class)
```

186

Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item. See Section 14.13.2 [13.13.2], page 540.

187

S'Read

For every subtype S of a specific type <T>:

188

S'Read denotes a procedure with the following specification:

189/2

```
procedure S'Read(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item> : out <T>)
```

190

S'Read reads the value of <Item> from <Stream>. See Section 14.13.2 [13.13.2], page 540.

191

S'Remainder

For every subtype S of
a floating point type
<T>:

192

S'Remainder denotes
a function with
the following
specification:

193

```
function S'Remainder (<X>, <Y> : <T>)
    return <T>
```

194

For nonzero <Y>, let
<v> be the value <X>
- <n> · <Y>, where
<n> is the integer
nearest to the exact
value of <X>/<Y>; if
 $|\langle n \rangle - \langle X \rangle / \langle Y \rangle|$
= 1/2, then <n> is
chosen to be even.
If <v> is a machine
number of the type
<T>, the function
yields <v>; otherwise,
it yields zero.
Constraint_Error
is raised if <Y> is
zero. A zero result
has the sign of <X>
when S'Signed_Zeros
is True. See
Section 15.5.3 [A.5.3],
page 663.

195

S'Round

For every decimal
fixed point subtype S:

196

S'Round denotes
a function with

the following
specification:

197

```
function S'Round(<X> : <universal_real>)■  
    return S'Base
```

198

The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S). See Section 4.5.10 [3.5.10], page 109.

199

S'Rounding

For every subtype S of a floating point type <T>:

200

S'Rounding denotes a function with the following specification:

201

```
function S'Rounding (<X> : <T>)  
    return <T>
```

202

The function yields the integral value nearest to <X>, rounding away from zero if <X> lies exactly halfway between two integers. A zero result has the sign of <X> when S'Signed_Zeros is True. See

Section 15.5.3 [A.5.3],
page 663.

203
S'Safe.First

For every subtype S of
a floating point type
<T>:

204

Yields the lower
bound of the
safe range (see
Section 4.5.7 [3.5.7],
page 103) of the
type <T>. If the
Numerics Annex is
not supported, the
value of this attribute
is implementation
defined; see
Section 21.2.2 [G.2.2],
page 1105, for the
definition that applies
to implementations
supporting the
Numerics Annex.
The value of this
attribute is of the
type <universal_real>.
See Section 15.5.3
[A.5.3], page 663.

205
S'Safe.Last

For every subtype S of
a floating point type
<T>:

206

Yields the upper
bound of the
safe range (see
Section 4.5.7 [3.5.7],
page 103) of the
type <T>. If the
Numerics Annex is

not supported, the value of this attribute is implementation defined; see Section 21.2.2 [G.2.2], page 1105, for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type `<universal_real>`. See Section 15.5.3 [A.5.3], page 663.

207
S'Scale

For every decimal fixed point subtype S:

208

S'Scale denotes the `<scale>` of the subtype S, defined as the value N such that $S'Delta = 10.0^{*(-N)}$. The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S. The value of this attribute is of the type `<universal_integer>`. See Section 4.5.10 [3.5.10], page 109.

209
S'Scaling

For every subtype S of a floating point type `<T>`:

210

S'Scaling denotes a function with the following specification:

211

```
function S'Scaling (<X> : <T>;  
                  <Adjustment> : <universal_integer>  
                  )  
    return <T>
```

212

Let $\langle v \rangle$ be the value $\langle X \rangle \cdot \langle T \rangle \text{'Machine_Radix} \langle \text{Adjustment} \rangle$. If $\langle v \rangle$ is a machine number of the type $\langle T \rangle$, or if $|\langle v \rangle| \geq \langle T \rangle \text{'Model_Small}$, the function yields $\langle v \rangle$; otherwise, it yields either one of the machine numbers of the type $\langle T \rangle$ adjacent to $\langle v \rangle$. `Constraint_Error` is optionally raised if $\langle v \rangle$ is outside the base range of S. A zero result has the sign of $\langle X \rangle$ when `S'Signed_Zeros` is `True`. See Section 15.5.3 [A.5.3], page 663.

213

S'Signed_Zeros

For every subtype S of a floating point type $\langle T \rangle$:

214

Yields the value `True` if the hardware representation for the type $\langle T \rangle$ has

the capability of representing both positively and negatively signed zeros, these being generated and used by the predefined operations of the type <T> as specified in IEC 559:1989; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See Section 15.5.3 [A.5.3], page 663.

215
S'Size

For every subtype S:


216

If S is definite, denotes the size (in bits) that the implementation would choose for the following objects of subtype S:

217

- A record component of subtype S when the record type is packed.

218

- The formal parameter of an instance of Unchecked_Conversion that converts from subtype S
- 

to some other
subtype.

219

If S is indefinite,
the meaning is
implementation
defined. The value
of this attribute
is of the type
<universal_integer>.
See Section 14.3
[13.3], page 486.

220/1
X'Size

For a prefix X that de-
notes an object:

221

Denotes the size
in bits of the
representation of the
object. The value
of this attribute
is of the type
<universal_integer>.
See Section 14.3
[13.3], page 486.

222
S'Small

For every fixed point
subtype S:

223

S'Small denotes the
<small> of the type of
S. The value of this
attribute is of the
type <universal_real>.
See Section 4.5.10
[3.5.10], page 109.

224
S'Storage_Pool

225 For every
access-to-object
subtype S:

Denotes the storage
pool of the type of
S. The type of this
attribute is `Root_-
Storage_Pool'Class`.
See Section 14.11
[13.11], page 526.

226
`S'Storage_Size`

For every
access-to-object
subtype S:

227

Yields the result
of calling `Stor-
age_Size(S'Storage_Pool)`,
which is intended to
be a measure of the
number of storage
elements reserved for
the pool. The type
of this attribute is
<universal_integer>.
See Section 14.11
[13.11], page 526.

228/1
`T'Storage_Size`

For a prefix T that
denotes a task object
(after any implicit
dereference):

229

Denotes the number
of storage elements
reserved for the
task. The value
of this attribute
is of the type

<universal_integer>.
The `Storage_Size` includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the "task control block" used by some implementations.)
See Section 14.3 [13.3], page 486.

229.1/2
`S'Stream_Size`

For every subtype `S` of an elementary type `<T>`:

229.2/2

Denotes the number of bits occupied in a stream by items of subtype `S`. Hence, the number of stream elements required per item of elementary type `<T>` is:

229.3/2

`<T>'Stream_Size / Ada.Streams.Stream_Element'Size`■

229.4/2

The value of this attribute is of type `<universal_integer>` and is a multiple of `Stream_Element'Size`.
See Section 14.13.2 [13.13.2], page 540.

230
`S'Succ`

For every scalar sub-
type S:

231

S'Succ denotes
a function with
the following
specification:

232

```
function S'Succ(<Arg> : S'Base)
  return S'Base
```

233

For an enumeration
type, the function
returns the value
whose position
number is one more
than that of the
value of <Arg>;
Constraint_Error is
raised if there is no
such value of the
type. For an integer
type, the function
returns the result
of adding one to
the value of <Arg>.
For a fixed point
type, the function
returns the result of
adding <small> to
the value of <Arg>.
For a floating point
type, the function
returns the machine
number (as defined
in Section 4.5.7
[3.5.7], page 103)
immediately above
the value of <Arg>;
Constraint_Error
is raised if there is
no such machine

number. See
Section 4.5 [3.5],
page 76.

234
S'Tag

For every subtype
S of a tagged type
<T> (specific or
class-wide):

235

S'Tag denotes the
tag of the type
<T> (or if <T> is
class-wide, the tag of
the root type of the
corresponding class).
The value of this
attribute is of type
Tag. See Section 4.9
[3.9], page 136.

236
X'Tag

For a prefix X that
is of a class-wide
tagged type (after any
implicit dereference):

237

X'Tag denotes the
tag of X. The value
of this attribute is
of type Tag. See
Section 4.9 [3.9],
page 136.

238
T'Terminated

For a prefix T that
is of a task type
(after any implicit
dereference):

239

Yields the value True
if the task denoted by

T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean. See Section 10.9 [9.9], page 388.

240
S'Truncation

For every subtype S of a floating point type <T>:

241

S'Truncation denotes a function with the following specification:

242

```
function S'Truncation (<X> : <T>)
  return <T>
```

243

The function yields the value ceiling(<X>) when <X> is negative, and floor(<X>) otherwise. A zero result has the sign of <X> when S'Signed_Zeros is True. See Section 15.5.3 [A.5.3], page 663.

244
S'Unbiased_Rounding

For every subtype S of a floating point type <T>:

245

S'Unbiased_Rounding denotes a function

with the following
specification:

246

```
function S'Unbiased_Rounding (<X> : <T>)■  
    return <T>
```

247

The function yields the integral value nearest to <X>, rounding toward the even integer if <X> lies exactly halfway between two integers. A zero result has the sign of <X> when S'Signed_Zeros is True. See Section 15.5.3 [A.5.3], page 663.

248

X'Unchecked_Access

For a prefix X that denotes an aliased view of an object:

249

All rules and semantics that apply to X'Access (see Section 4.10.2 [3.10.2], page 164) apply also to X'Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package. See Section 14.10 [13.10], page 525.

250
S'Val

For every discrete
subtype S:

251

S'Val denotes
a function with
the following
specification:

252

```
function S'Val(<Arg> : <universal_integer>)■  
    return S'Base
```

253

This function returns
a value of the type
of S whose position
number equals the
value of <Arg>. See
Section 4.5.5 [3.5.5],
page 99.

254
X'Valid

For a prefix X that
denotes a scalar
object (after any
implicit dereference):

255

Yields True if and
only if the object
denoted by X is
normal and has a
valid representation.
The value of this
attribute is of the
predefined type
Boolean. See
Section 14.9.2
[13.9.2], page 524.

256
S'Value

For every scalar sub-
type S:

257

S'Value denotes
a function with
the following
specification:

258

```
function S'Value(<Arg> : String)
  return S'Base
```

259

This function returns
a value given an
image of the value
as a String, ignoring
any leading or
trailing spaces. See
Section 4.5 [3.5],
page 76.

260/1

P'Version

For a prefix P that
statically denotes a
program unit:

261

Yields a value of
the predefined type
String that identifies
the version of the
compilation unit
that contains the
declaration of the
program unit. See
Section 19.3 [E.3],
page 1043.

262

S'Wide_Image

For every scalar sub-
type S:

263

S'Wide_Image
denotes a function
with the following
specification:

264

```
function S'Wide_Image(<Arg> : S'Base)
  return Wide_String
```

265/2

The function returns
an image of the
value of <Arg> as a
Wide_String. See
Section 4.5 [3.5],
page 76.

266

S'Wide_Value

For every scalar sub-
type S:

267

S'Wide_Value
denotes a function
with the following
specification:

268

```
function S'Wide_Value(<Arg> : Wide_String)
  return S'Base
```

269

This function returns
a value given an
image of the value
as a Wide_String,
ignoring any leading
or trailing spaces.
See Section 4.5 [3.5],
page 76.

269.1/2

S'Wide_Wide_Image

For every scalar sub-
type S:

269.2/2

S'Wide_Wide_Image
denotes a function
with the following
specification:

269.3/2

```
function S'Wide_Wide_Image(<Arg> : S'Base)
  return Wide_Wide_String
```

269.4/2

The function returns
an <image> of the
value of <Arg>,
that is, a sequence
of characters
representing the value
in display form. See
Section 4.5 [3.5],
page 76.

269.5/2

S'Wide_Wide_Value

For every scalar sub-
type S:

269.6/2

S'Wide_Wide_Value
denotes a function
with the following
specification:

269.7/2

```
function S'Wide_Wide_Value(<Arg> : Wide_Wide_String)
  return S'Base
```

269.8/2

This function returns
a value given an
image of the value as
a Wide_Wide_String,
ignoring any leading

or trailing spaces.
See Section 4.5 [3.5],
page 76.

269.9/2
S'Wide_Wide_Width

For every scalar sub-
type S:

269.10/2

S'Wide_Wide_Width
denotes the
maximum length of
a Wide_Wide_String
returned by
S'Wide_Wide_Image
over all values of
the subtype S. It
denotes zero for a
subtype that has a
null range. Its type is
<universal_integer>.
See Section 4.5 [3.5],
page 76.

270
S'Wide_Width

For every scalar sub-
type S:

271

S'Wide_Width
denotes the
maximum length of a
Wide_String returned
by S'Wide_Image
over all values of
the subtype S. It
denotes zero for a
subtype that has a
null range. Its type is
<universal_integer>.
See Section 4.5 [3.5],
page 76.

272
S'Width

For every scalar subtype S:

273

S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S. It denotes zero for a subtype that has a null range. Its type is <universal_integer>. See Section 4.5 [3.5], page 76.

274

S'Class'Write

For every subtype S'Class of a class-wide type <T>'Class:

275

S'Class'Write denotes a procedure with the following specification:

276/2

```
procedure S'Class'Write(  
    <Stream> : not null access Ada.Streams.Root_Stream  
    <Item>   : in <T>'Class)
```

277

Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item. See Section 14.13.2 [13.13.2], page 540.

278

S'Write

For every subtype S of
a specific type <T>:

279

S'Write denotes
a procedure with
the following
specification:

280/2

```
procedure S'Write(  
  <Stream> : not null access Ada.Streams.Root_Stream  
  <Item> : in <T>)
```

281

S'Write writes the
value of <Item>
to <Stream>. See
Section 14.13.2
[13.13.2], page 540.

25 Annex L Language-Defined Pragmas

1

This Annex summarizes the definitions given elsewhere of the language-defined pragmas.

2

pragma All_Calls_Remote[(<library_unit_>name)]; -- See Section 19.2.3 [E.2.3], page 1041.

2.1/2

pragma Assert([Check ==>] <boolean_>expression[, [Message ==>] <string_>expression]); -- See Section 12.4.2 [11.4.2], page 427.

2.2/2

pragma Assertion_Policy(<policy_>identifier); -- See Section 12.4.2 [11.4.2], page 427.

3

pragma Asynchronous(local_name); -- See Section 19.4.1 [E.4.1], page 1047.

4

pragma Atomic(local_name); -- See Section 17.6 [C.6], page 962.

5

pragma Atomic_Components(<array_>local_name); -- See Section 17.6 [C.6], page 962.

6

pragma Attach_Handler(<handler_>name, expression); -- See Section 17.3.1 [C.3.1], page 954.

7

pragma Controlled(<first_subtype_>local_name); -- See Section 14.11.3 [13.11.3], page 534.

8

pragma Convention([Convention ==>] <convention_>identifier,[Entity ==>] local_name); -- See Section 16.1 [B.1], page 894.

8.1/2

pragma Detect_Blocking; -- See Section 22.5 [H.5], page 1163.

9

pragma Discard_Names([([On ==>] local_name)]; -- See Section 17.5 [C.5], page 961.

10

pragma Elaborate(<library_unit_>name{, <library_unit_>name}); -- See Section 11.2.1 [10.2.1], page 413.

11

pragma Elaborate_All(<library_unit_>name{, <library_unit_>name}); -- See Section 11.2.1 [10.2.1], page 413.

12

pragma Elaborate_Body[(<library_unit_>name)]; -- See Section 11.2.1 [10.2.1], page 413.

13

pragma Export(
 [Convention ==>] <convention_>identifier, [Entity ==>] local_name
 [, [External_Name ==>] <string_>expression] [, [Link_Name ==>] <string_>expression]); -- See Section 16.1 [B.1], page 894.

14

pragma Import(
 [Convention ==>] <convention_>identifier, [Entity ==>] local_name
 [, [External_Name ==>] <string_>expression] [, [Link_Name ==>] <string_>expression]); -- See Section 16.1 [B.1], page 894.

[Convention =>] <convention_>identifier, [Entity =>] local_name
[, [External_Name =>] <string_>expression] [, [Link_Name =>] <string_>expression]); --
See Section 16.1 [B.1], page 894.

15

pragma Inline(name {, name}); -- See Section 7.3.2 [6.3.2], page 266.

16

pragma Inspection_Point[(<object_>name {, <object_>name})]; -- See Section 22.3.2
[H.3.2], page 1157.

17

pragma Interrupt_Handler(<handler_>name); -- See Section 17.3.1 [C.3.1], page 954.

18

pragma Interrupt_Priority[(expression)]; -- See Section 18.1 [D.1], page 975.

19

pragma Linker_Options(<string_>expression); -- See Section 16.1 [B.1], page 894.

20

pragma List(identifier); -- See Section 3.8 [2.8], page 44.

21

pragma Locking_Policy(<policy_>identifier); -- See Section 18.3 [D.3], page 991.

21.1/2

pragma No_Return(<procedure_>local_name{, <procedure_>local_name}); -- See
Section 7.5.1 [6.5.1], page 275.

22

pragma Normalize Scalars; -- See Section 22.1 [H.1], page 1153.

23

pragma Optimize(identifier); -- See Section 3.8 [2.8], page 44.

24

pragma Pack(<first_subtype_>local_name); -- See Section 14.2 [13.2], page 485.

25

pragma Page; -- See Section 3.8 [2.8], page 44.

25.1/2

pragma Partition_Elaboration_Policy (<policy_>identifier); -- See Section 22.6 [H.6],
page 1163.

25.2/2

pragma Prelaborable_Initialization(direct_name); -- See Section 11.2.1 [10.2.1], page 413.

26

pragma Prelaborate[(<library_unit_>name)]; -- See Section 11.2.1 [10.2.1], page 413.

27

pragma Priority(expression); -- See Section 18.1 [D.1], page 975.

27.1/2

pragma Priority_Specific_Dispatching (
 <policy_>identifier, <first_priority_>expression, <last_priority_>expression); -- See
Section 18.2.2 [D.2.2], page 980.

27.2/2

pragma Profile (<profile_>identifier {, <profile_>pragma_argument_association}); -- See Section 18.13 [D.13], page 1020.

28

pragma Pure[(<library_unit_>name)]; -- See Section 11.2.1 [10.2.1], page 413.

29

pragma Queuing_Policy(<policy_>identifier); -- See Section 18.4 [D.4], page 994.

29.1/2

pragma Relative_Deadline (<relative_deadline_>expression); -- See Section 18.2.6 [D.2.6], page 987.

30

pragma Remote_Call_Interface[(<library_unit_>name)]; -- See Section 19.2.3 [E.2.3], page 1041.

31

pragma Remote_Types[(<library_unit_>name)]; -- See Section 19.2.2 [E.2.2], page 1039.

32

pragma Restrictions(restriction{, restriction}); -- See Section 14.12 [13.12], page 535.

33

pragma Reviewable; -- See Section 22.3.1 [H.3.1], page 1155.

34

pragma Shared_Passive[(<library_unit_>name)]; -- See Section 19.2.1 [E.2.1], page 1038.

35

pragma Storage_Size(expression); -- See Section 14.3 [13.3], page 486.

36

pragma Suppress(identifier); -- See Section 12.5 [11.5], page 431.

37

pragma Task_Dispatching_Policy(<policy_>identifier); -- See Section 18.2.2 [D.2.2], page 980.

37.1/2

pragma Unchecked_Union (<first_subtype_>local_name); -- See Section 16.3.3 [B.3.3], page 928.

37.2/2

pragma Unsuppress(identifier); -- See Section 12.5 [11.5], page 431.

38

pragma Volatile(local_name); -- See Section 17.6 [C.6], page 962.

39

pragma Volatile_Components(<array_>local_name); -- See Section 17.6 [C.6], page 962.

26 Annex M Summary of Documentation Requirements

1/2

The Ada language allows for certain target machine dependences in a controlled manner. Each Ada implementation must document many characteristics and properties of the target system. This International Standard contains specific documentation requirements. In addition, many characteristics that require documentation are identified throughout this International Standard as being implementation defined. Finally, this International Standard requires documentation of whether implementation advice is followed. The following clauses provide summaries of these documentation requirements.

26.1 M.1 Specific Documentation Requirements

1/2

In addition to implementation–defined characteristics, each Ada implementation must document various properties of the implementation:

2/2

- The behavior of implementations in implementation–defined situations shall be documented — see Section 26.2 [M.2], page 1250, "Section 26.2 [M.2], page 1250, Implementation-Defined Characteristics" for a listing. See Section 2.1.3 [1.1.3], page 23(19).

3/2

- The set of values that a user–defined Allocate procedure needs to accept for the Alignment parameter. How the standard storage pool is chosen, and how storage is allocated by standard storage pools. See Section 14.11 [13.11], page 526(22).

4/2

- The algorithm used for random number generation, including a description of its period. See Section 15.5.2 [A.5.2], page 654(44).

5/2

- The minimum time interval between calls to the time–dependent Reset procedure that is guaranteed to initiate different random number sequences. See Section 15.5.2 [A.5.2], page 654(45).

6/2

- The conditions under which `Io_Exceptions.Name_Error`, `Io_Exceptions.Use_Error`, and `Io_Exceptions.Device_Error` are propagated. See Section 15.13 [A.13], page 752(15).

7/2

- The behavior of package `Environment_Variables` when environment variables are changed by external mechanisms. See Section 15.17 [A.17], page 775(30/2).

8/2

- The overhead of calling machine—code or intrinsic subprograms. See Section 17.1 [C.1], page 949(6).

9/2

- The types and attributes used in machine code insertions. See Section 17.1 [C.1], page 949(7).

10/2

- The subprogram calling conventions for all supported convention identifiers. See Section 17.1 [C.1], page 949(8).

11/2

- The mapping between the `Link_Name` or Ada designator and the external link name. See Section 17.1 [C.1], page 949(9).

12/2

- The treatment of interrupts. See Section 17.3 [C.3], page 951(22).

13/2

- The metrics for interrupt handlers. See Section 17.3.1 [C.3.1], page 954(16).

14/2

- If the `Ceiling-Locking` policy is in effect, the default ceiling priority for a protected object that contains an interrupt handler pragma. See Section 17.3.2 [C.3.2], page 957(24/2).

15/2

- Any circumstances when the elaboration of a preelaborated package causes code to be executed. See Section 17.4 [C.4], page 960(12).

16/2

- Whether a partition can be restarted without reloading. See Section 17.4 [C.4], page 960(13).

17/2

- The effect of calling `Current_Task` from an entry body or interrupt handler. See Section 17.7.1 [C.7.1], page 965(19).

18/2

- For package Task_Attributes, limits on the number and size of task attributes, and how to configure any limits. See Section 17.7.2 [C.7.2], page 967(19).

19/2

- The metrics for the Task_Attributes package. See Section 17.7.2 [C.7.2], page 967(27).

20/2

- The details of the configuration used to generate the values of all metrics. See Section 30.5 [D], page 1424(2).

21/2

- The maximum priority inversion a user task can experience from the implementation. See Section 18.2.3 [D.2.3], page 982(12/2).

22/2

- The amount of time that a task can be preempted for processing on behalf of lower-priority tasks. See Section 18.2.3 [D.2.3], page 982(13/2).

23/2

- The quantum values supported for round robin dispatching. See Section 18.2.5 [D.2.5], page 985(16/2).

24/2

- The accuracy of the detection of the exhaustion of the budget of a task for round robin dispatching. See Section 18.2.5 [D.2.5], page 985(17/2).

25/2

- Any conditions that cause the completion of the setting of the deadline of a task to be delayed for a multiprocessor. See Section 18.2.6 [D.2.6], page 987(32/2).

26/2

- Any conditions that cause the completion of the setting of the priority of a task to be delayed for a multiprocessor. See Section 18.5.1 [D.5.1], page 996(12.1/2).

27/2

- The metrics for Set_Priority. See Section 18.5.1 [D.5.1], page 996(14).

28/2

- The metrics for setting the priority of a protected object. See Section 18.5.2 [D.5.2], page 998(10).

29/2

- On a multiprocessor, any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. See Section 18.6 [D.6], page 1000(3).

30/2

- The metrics for aborts. See Section 18.6 [D.6], page 1000(8).

31/2

- The values of `Time_First`, `Time_Last`, `Time_Span_First`, `Time_Span_Last`, `Time_Span_Unit`, and `Tick` for package `Real_Time`. See Section 18.8 [D.8], page 1008(33).

32/2

- The properties of the underlying time base used in package `Real_Time`. See Section 18.8 [D.8], page 1008(34).

33/2

- Any synchronization of package `Real_Time` with external time references. See Section 18.8 [D.8], page 1008(35).

34/2

- Any aspects of the external environment that could interfere with package `Real_Time`. See Section 18.8 [D.8], page 1008(36/1).

35/2

- The metrics for package `Real_Time`. See Section 18.8 [D.8], page 1008(45).

36/2

- The minimum value of the delay expression of a `delay_relative_statement` that causes a task to actually be blocked. See Section 18.9 [D.9], page 1013(7).

37/2

- The minimum difference between the value of the delay expression of a `delay_until_statement` and the value of `Real_Time.Clock`, that causes the task to actually be blocked. See Section 18.9 [D.9], page 1013(8).

38/2

- The metrics for delay statements. See Section 18.9 [D.9], page 1013(13).

39/2

- The upper bound on the duration of interrupt blocking caused by the implementation. See Section 18.12 [D.12], page 1018(5).

40/2

- The metrics for entry-less protected objects. See Section 18.12 [D.12], page 1018(12).

41/2

- The values of `CPU_Time_First`, `CPU_Time_Last`, `CPU_Time_Unit`, and `CPU_Tick` of package `Execution_Time`. See Section 18.14 [D.14], page 1021(21/2).

42/2

- The properties of the mechanism used to implement package `Execution_Time`. See Section 18.14 [D.14], page 1021(22/2).

43/2

- The metrics for execution time. See Section 18.14 [D.14], page 1021(27).

44/2

- The metrics for timing events. See Section 18.15 [D.15], page 1031(24).

45/2

- Whether the RPC-receiver is invoked from concurrent tasks, and if so, the number of such tasks. See Section 19.5 [E.5], page 1050(25).

46/2

- Any techniques used to reduce cancellation errors in `Numerics.Generic_Real_Arrays` shall be documented. See Section 21.3.1 [G.3.1], page 1119(86/2).

47/2

- Any techniques used to reduce cancellation errors in `Numerics.Generic_Complex_Arrays` shall be documented. See Section 21.3.2 [G.3.2], page 1130(155/2).

48/2

- If a pragma `Normalize Scalars` applies, the implicit initial values of scalar subtypes shall be documented. Such a value should be an invalid representation when possible; any cases when it is not shall be documented. See Section 22.1 [H.1], page 1153(5/2).

49/2

- The range of effects for each bounded error and each unspecified effect. If the effects of a given erroneous construct are constrained, the constraints shall be documented. See Section 22.2 [H.2], page 1154(1).

50/2

- For each inspection point, a mapping between each inspectable object and the machine resources where the object's value can be obtained shall be provided. See Section 22.3.2 [H.3.2], page 1157(8).

51/2

- If a pragma Restrictions(No_Exceptions) is specified, the effects of all constructs where language-defined checks are still performed. See Section 22.4 [H.4], page 1158(25).

52/2

- The interrupts to which a task entry may be attached. See Section 23.7.1 [J.7.1], page 1171(12).

53/2

- The type of entry call invoked for an interrupt entry. See Section 23.7.1 [J.7.1], page 1171(13).

26.2 M.2 Implementation-Defined Characteristics

1/2

The Ada language allows for certain machine dependences in a controlled manner. Each Ada implementation must document all implementation-defined characteristics:

1.1/2

- Whether or not each recommendation given in Implementation Advice is followed -- see Section 26.3 [M.3], page 1267, "Section 26.3 [M.3], page 1267, Implementation Advice" for a listing. See Section 2.1.2 [1.1.2], page 20(37).

2

- Capacity limitations of the implementation. See Section 2.1.3 [1.1.3], page 23(3).

3

- Variations from the standard that are impractical to avoid given the implementation's execution environment. See Section 2.1.3 [1.1.3], page 23(6).

4

- Which code statements cause external interactions. See Section 2.1.3 [1.1.3], page 23(10).

5

- The coded representation for the text of an Ada program. See Section 3.1 [2.1], page 32(4/2).

5.1/2

- The semantics of an Ada program whose text is not in Normalization Form KC. See Section 3.1 [2.1], page 32(4.1/2).

5.2/2

- <This paragraph was deleted.>

6

- The representation for an end of line. See Section 3.2 [2.2], page 36(2/2).

7

- Maximum supported line length and lexical element length. See Section 3.2 [2.2], page 36(14).

8

- Implementation-defined pragmas. See Section 3.8 [2.8], page 44(14).

9

- Effect of pragma Optimize. See Section 3.8 [2.8], page 44(27).

9.1/2

- The sequence of characters of the value returned by S'Wide_Image when some of the graphic characters of S'Wide_Wide_Image are not defined in Wide_Character. See Section 4.5 [3.5], page 76(30/2).

9.2/2

- The sequence of characters of the value returned by S'Image when some of the graphic characters of S'Wide_Wide_Image are not defined in Character. See Section 4.5 [3.5], page 76(37/2).

10

- The predefined integer types declared in Standard. See Section 4.5.4 [3.5.4], page 95(25).

11

- Any nonstandard integer types and the operators defined for them. See Section 4.5.4 [3.5.4], page 95(26).

12

- Any nonstandard real types and the operators defined for them. See Section 4.5.6 [3.5.6], page 102(8).

13

- What combinations of requested decimal precision and range are supported for floating point types. See Section 4.5.7 [3.5.7], page 103(7).

14

- The predefined floating point types declared in Standard. See Section 4.5.7 [3.5.7], page 103(16).

15

- The <small> of an ordinary fixed point type. See Section 4.5.9 [3.5.9], page 106(8/2).

16

- What combinations of <small>, range, and <digits> are supported for fixed point types. See Section 4.5.9 [3.5.9], page 106(10).

16.1/2

- The result of `Tags.Wide_Wide_Expanded_Name` for types declared within an unnamed `block_statement`. See Section 4.9 [3.9], page 136(10).

16.2/2

- The sequence of characters of the value returned by `Tags.Expanded_Name` (respectively, `Tags.Wide_Expanded_Name`) when some of the graphic characters of `Tags.Wide_Wide_Expanded_Name` are not defined in `Character` (respectively, `Wide_Character`). See Section 4.9 [3.9], page 136(10.1/2).

17

- Implementation–defined attributes. See Section 5.1.4 [4.1.4], page 187(12/1).

17.1/2

- Rounding of real static expressions which are exactly half-way between two machine numbers. See Section 5.9 [4.9], page 234(38/2).

18

- Any implementation-defined time types. See Section 10.6 [9.6], page 358(6).

19

- The time base associated with relative delays. See Section 10.6 [9.6], page 358(20).

20

- The time base of the type `Calendar.Time`. See Section 10.6 [9.6], page 358(23).

20.1/2

- The time zone used for package `Calendar` operations. See Section 10.6 [9.6], page 358(24/2).

21

- Any limit on `delay_until` statements of `select` statements. See Section 10.6 [9.6], page 358(29).

21.1/2

- The result of `Calendar.Formatting.Image` if its argument represents more than 100 hours. See Section 10.6.1 [9.6.1], page 363(86/2).

22

- Whether or not two nonoverlapping parts of a composite object are independently addressable, in the case where packing, record layout, or `Component_Size` is specified for the object. See Section 10.10 [9.10], page 389(1).

23

- The representation for a compilation. See Section 11.1 [10.1], page 394(2).

24

- Any restrictions on compilations that contain multiple `compilation_units`. See Section 11.1 [10.1], page 394(4).

25

- The mechanisms for creating an environment and for adding and replacing compilation units. See Section 11.1.4 [10.1.4], page 406(3/2).

25.1/2

- The mechanisms for adding a compilation unit mentioned in a `limited_with_clause` to an environment. See Section 11.1.4 [10.1.4], page 406(3/2).

26

- The manner of explicitly assigning library units to a partition. See Section 11.2 [10.2], page 409(2).

27

- The implementation–defined means, if any, of specifying which compilation units are needed by a given compilation unit. See Section 11.2 [10.2], page 409(2).

28

- The manner of designating the main subprogram of a partition. See Section 11.2 [10.2], page 409(7).

29

- The order of elaboration of `library_items`. See Section 11.2 [10.2], page 409(18).

30

- Parameter passing and function return for the main subprogram. See Section 11.2 [10.2], page 409(21).

31

- The mechanisms for building and running partitions. See Section 11.2 [10.2], page 409(24).

32

- The details of program execution, including program termination. See Section 11.2 [10.2], page 409(25).

33

- The semantics of any nonactive partitions supported by the implementation. See Section 11.2 [10.2], page 409(28).

34

- The information returned by `Exception_Message`. See Section 12.4.1 [11.4.1], page 423(10.1/2).

34.1/2

- The result of `Exceptions.Wide_Wide_Exception_Name` for exceptions declared within an unnamed block_statement. See Section 12.4.1 [11.4.1], page 423(12).

34.2/2

- The sequence of characters of the value returned by `Exceptions.Exception_Name` (respectively, `Exceptions.Wide_Exception_Name`) when some of the graphic characters of `Exceptions.Wide_Wide_Exception_Name` are not defined in `Character` (respectively, `Wide_Character`). See Section 12.4.1 [11.4.1], page 423(12.1/2).

35

- The information returned by `Exception_Information`. See Section 12.4.1 [11.4.1], page 423(13/2).

35.1/2

- Implementation—defined <policy_>identifiers allowed in a pragma `Assertion_Policy`. See Section 12.4.2 [11.4.2], page 427(9/2).

35.2/2

- The default assertion policy. See Section 12.4.2 [11.4.2], page 427(10/2).

36

- Implementation—defined check names. See Section 12.5 [11.5], page 431(27).

36.1/2

- Existence and meaning of second parameter of pragma `Unsuppress`. See Section 12.5 [11.5], page 431(27.1/2).

36.2/2

- The cases that cause conflicts between the representation of the ancestors of a type_declaration. See Section 14.1 [13.1], page 481(13.1/2).

37

- The interpretation of each aspect of representation. See Section 14.1 [13.1], page 481(20).

38

- Any restrictions placed upon representation items. See Section 14.1 [13.1], page 481(20).

38.1/2

- The set of machine scalars. See Section 14.3 [13.3], page 486(8.1/2).

39

- The meaning of Size for indefinite subtypes. See Section 14.3 [13.3], page 486(48).

40

- The default external representation for a type tag. See Section 14.3 [13.3], page 486(75/1).

41

- What determines whether a compilation unit is the same in two different partitions. See Section 14.3 [13.3], page 486(76).

42

- Implementation–defined components. See Section 14.5.1 [13.5.1], page 501(15).

43

- If Word_Size = Storage_Unit, the default bit ordering. See Section 14.5.3 [13.5.3], page 508(5).

43.1/2

- The contents of the visible part of package System. See Section 14.7 [13.7], page 510(2).

43.2/2

- The range of Storage_Elements.Storage_Offset, the modulus of Storage_Elements.Storage_Element, and the declaration of Storage_Elements.Integer_Address. See Section 14.7.1 [13.7.1], page 516(11).

44

- The contents of the visible part of package System.Machine_Code, and the meaning of code_statements. See Section 14.8 [13.8], page 518(7).

44.1/2

- The result of unchecked conversion for instances with scalar result types whose result is not defined by the language. See Section 14.9 [13.9], page 520(11).

44.2/2

- The effect of unchecked conversion for instances with nonscalar result types whose effect is not defined by the language. See Section 14.9 [13.9], page 520(11).

44.3/2

- <This paragraph was deleted.>

45

- Whether or not the implementation provides user-accessible names for the standard pool type(s). See Section 14.11 [13.11], page 526(17).

45.1/2

- The meaning of `Storage_Size` when neither the `Storage_Size` nor the `Storage_Pool` is specified for an access type. See Section 14.11 [13.11], page 526(18).

45.2/2

- <This paragraph was deleted.>

45.3/2

- The set of restrictions allowed in a pragma `Restrictions`. See Section 14.12 [13.12], page 535(7/2).

46

- The consequences of violating limitations on `Restrictions` pragmas. See Section 14.12 [13.12], page 535(9).

46.1/2

- The contents of the stream elements read and written by the `Read` and `Write` attributes of elementary types. See Section 14.13.2 [13.13.2], page 540(9).

47

- The names and characteristics of the numeric subtypes declared in the visible part of package `Standard`. See Section 15.1 [A.1], page 556(3).

47.1/2

- The values returned by `Strings.Hash`. See Section 15.4.9 [A.4.9], page 646(3/2).

48

- The accuracy actually achieved by the elementary functions. See Section 15.5.1 [A.5.1], page 648(1).

49

- The sign of a zero result from some of the operators or functions in `Numerics.Generic_Elementary_Functions`, when `Float_Type.Signed_Zeros` is `True`. See Section 15.5.1 [A.5.1], page 648(46).

50

- The value of `Numerics.Float_Random.Max_Image_Width`. See Section 15.5.2 [A.5.2], page 654(27).

51

- The value of `Numerics.Discrete_Random.Max_Image_Width`. See Section 15.5.2 [A.5.2], page 654(27).

51.1/2

- <This paragraph was deleted.>

52

- The string representation of a random number generator's state. See Section 15.5.2 [A.5.2], page 654(38).

52.1/2

- <This paragraph was deleted.>

53

- The values of the `Model_Mantissa`, `Model_Emin`, `Model_Epsilon`, `Model`, `Safe_First`, and `Safe_Last` attributes, if the Numerics Annex is not supported. See Section 15.5.3 [A.5.3], page 663(72).

53.1/2

- <This paragraph was deleted.>

54

- The value of `Buffer_Size` in `Storage_IO`. See Section 15.9 [A.9], page 695(10).

54.1/2

- The external files associated with the standard input, standard output, and standard error files. See Section 15.10 [A.10], page 696(5).

55

- The accuracy of the value produced by `Put`. See Section 15.10.9 [A.10.9], page 731(36).

55.1/1

- Current size for a stream file for which positioning is not supported. See Section 15.12.1 [A.12.1], page 746(1.1/1).

55.2/2

- The meaning of `Argument_Count`, `Argument`, and `Command_Name` for package `Command_Line`. The bounds of type `Command_Line.Exit_Status`. See Section 15.15 [A.15], page 754(1).

55.3/2

- The interpretation of file names and directory names. See Section 15.16 [A.16], page 757(46/2).

55.4/2

- The maximum value for a file size in `Directories`. See Section 15.16 [A.16], page 757(87/2).

55.5/2

- The result for `Directories.Size` for a directory or special file. See Section 15.16 [A.16], page 757(93/2).

55.6/2

- The result for `Directories.Modification_Time` for a directory or special file. See Section 15.16 [A.16], page 757(95/2).

55.7/2

- The interpretation of a non-null search pattern in `Directories`. See Section 15.16 [A.16], page 757(104/2).

55.8/2

- The results of a `Directories` search if the contents of the directory are altered while a search is in progress. See Section 15.16 [A.16], page 757(110/2).

55.9/2

- The definition and meaning of an environment variable. See Section 15.17 [A.17], page 775(1/2).

55.10/2

- The circumstances where an environment variable cannot be defined. See Section 15.17 [A.17], page 775(16/2).

55.11/2

- Environment names for which `Set` has the effect of `Clear`. See Section 15.17 [A.17], page 775(17/2).

55.12/2

- The value of `Containers.Hash_Type'Modulus`. The value of `Containers.Count_Type'Last`. See Section 15.18.1 [A.18.1], page 779(7/2).

56

- Implementation—defined convention names. See Section 16.1 [B.1], page 894(11).

57

- The meaning of link names. See Section 16.1 [B.1], page 894(36).

58

- The manner of choosing link names when neither the link name nor the address of an imported or exported entity is specified. See Section 16.1 [B.1], page 894(36).

59

- The effect of pragma `Linker_Options`. See Section 16.1 [B.1], page 894(37).

60

- The contents of the visible part of package `Interfaces` and its language—defined descendants. See Section 16.2 [B.2], page 900(1).

60.1/2

- Implementation—defined children of package `Interfaces`. See Section 16.2 [B.2], page 900(11).

60.2/2

- The definitions of certain types and constants in `Interfaces.C`. See Section 16.3 [B.3], page 901(41).

60.3/1

- The types `Floating`, `Long_Floating`, `Binary`, `Long_Binary`, `Decimal_Element`, and `COBOL_Character`; and the initializations of the variables `Ada_To_COBOL` and `COBOL_To_Ada`, in `Interfaces.COBOLE`. See Section 16.4 [B.4], page 931(50).

60.4/1

- The types `Fortran_Integer`, `Real`, `Double_Precision`, and `Character_Set` in `Interfaces.Fortran`. See Section 16.5 [B.5], page 945(17).

60.5/2

- Implementation—defined intrinsic subprograms. See Section 17.1 [C.1], page 949(1).

60.6/2

- <This paragraph was deleted.>

60.7/2

- <This paragraph was deleted.>

60.8/2

- Any restrictions on a protected procedure or its containing type when a pragma `Attach_Handler` or `Interrupt_Handler` applies. See Section 17.3.1 [C.3.1], page 954(17).

60.9/2

- Any other forms of interrupt handler supported by the `Attach_Handler` and `Interrupt_Handler` pragmas. See Section 17.3.1 [C.3.1], page 954(19).

60.10/2

- <This paragraph was deleted.>

61

- The semantics of pragma `Discard_Names`. See Section 17.5 [C.5], page 961(7).

62

- The result of the `Task_Identification.Image` attribute. See Section 17.7.1 [C.7.1], page 965(7).

62.1/2

- The value of `Current_Task` when in a protected entry, interrupt handler, or finalization of a task attribute. See Section 17.7.1 [C.7.1], page 965(17/2).

62.2/2

- <This paragraph was deleted.>

62.3/1

- Granularity of locking for `Task_Attributes`. See Section 17.7.2 [C.7.2], page 967(16/1).

62.4/2

- <This paragraph was deleted.>

62.5/2

- <This paragraph was deleted.>

63

- The declarations of `Any_Priority` and `Priority`. See Section 18.1 [D.1], page 975(11).

64

- Implementation–defined execution resources. See Section 18.1 [D.1], page 975(15).

65

- Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy. See Section 18.2.1 [D.2.1], page 978(3).

65.1/2

- The effect of implementation–defined execution resources on task dispatching. See Section 18.2.1 [D.2.1], page 978(9/2).

65.2/2

- <This paragraph was deleted.>

65.3/2

- <This paragraph was deleted.>

65.4/2

- Implementation defined task dispatching policies. See Section 18.2.2 [D.2.2], page 980(18).

65.5/2

- The value of `Default_Quantum` in `Dispatching.Round_Robin`. See Section 18.2.5 [D.2.5], page 985(4).

66

- Implementation–defined <policy_>identifiers allowed in a pragma `Locking_Policy`. See Section 18.3 [D.3], page 991(4).

66.1/2

- The locking policy if no `Locking_Policy` pragma applies to any unit of a partition. See Section 18.3 [D.3], page 991(6).

67

- Default ceiling priorities. See Section 18.3 [D.3], page 991(10/2).

68

- The ceiling of any protected object used internally by the implementation. See Section 18.3 [D.3], page 991(16).

69

- Implementation–defined queuing policies. See Section 18.4 [D.4], page 994(1/1).

69.1/2

- <This paragraph was deleted.>

70

- Any operations that implicitly require heap storage allocation. See Section 18.7 [D.7], page 1001(8).

70.1/2

- When restriction `No_Task_Termination` applies to a partition, what happens when a task terminates. See Section 18.7 [D.7], page 1001(15.1/2).

70.2/2

- The behavior when restriction `Max_Storage_At_Blocking` is violated. See Section 18.7 [D.7], page 1001(17/1).

70.3/2

- The behavior when restriction `Max_Asynchronous_Select_Nesting` is violated. See Section 18.7 [D.7], page 1001(18/1).

70.4/2

- The behavior when restriction `Max_Tasks` is violated. See Section 18.7 [D.7], page 1001(19).

70.5/2

- Whether the use of `pragma Restrictions` results in a reduction in program code or data size or execution time. See Section 18.7 [D.7], page 1001(20).

70.6/2

- <This paragraph was deleted.>

70.7/2

- <This paragraph was deleted.>

70.8/2

- <This paragraph was deleted.>

71

- The means for creating and executing distributed programs. See Section 30.6 [E], page 1435(5).

72

- Any events that can result in a partition becoming inaccessible. See Section 19.1 [E.1], page 1034(7).

73

- The scheduling policies, treatment of priorities, and management of shared resources between partitions in certain cases. See Section 19.1 [E.1], page 1034(11).

73.1/1

- <This paragraph was deleted.>

74

- Whether the execution of the remote subprogram is immediately aborted as a result of cancellation. See Section 19.4 [E.4], page 1044(13).

74.1/2

- The range of type `System.RPC.Partition_Id`. See Section 19.5 [E.5], page 1050(14).

74.2/2

- <This paragraph was deleted.>

75

- Implementation–defined interfaces in the PCS. See Section 19.5 [E.5], page 1050(26).

76

- The values of named numbers in the package `Decimal`. See Section 20.2 [F.2], page 1055(7).

77

- The value of `Max_Picture_Length` in the package `Text_IO.Editing`. See Section 20.3.3 [F.3.3], page 1073(16).

78

- The value of `Max_Picture_Length` in the package `Wide_Text_IO.Editing`. See Section 20.3.4 [F.3.4], page 1081(5).

78.1/2

- The value of `Max_Picture_Length` in the package `Wide_Wide_Text_IO.Editing`. See Section 20.3.5 [F.3.5], page 1081(5).

79

- The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations. See Section 21.1 [G.1], page 1084(1).

80

- The sign of a zero result (or a component thereof) from any operator or function in `Numerics.Generic.Complex.Types`, when `Real_Signed_Zeros` is `True`. See Section 21.1.1 [G.1.1], page 1084(53).

81

- The sign of a zero result (or a component thereof) from any operator or function in `Numerics.Generic.Complex.Elementary.Functions`, when `Complex.Types.Real_Signed_Zeros` is `True`. See Section 21.1.2 [G.1.2], page 1091(45).

82

- Whether the strict mode or the relaxed mode is the default. See Section 21.2 [G.2], page 1103(2).

83

- The result interval in certain cases of fixed-to-float conversion. See Section 21.2.1 [G.2.1], page 1104(10).

84

- The result of a floating point arithmetic operation in overflow situations, when the `Machine_Overflows` attribute of the result type is `False`. See Section 21.2.1 [G.2.1], page 1104(13).

85

- The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal. See Section 21.2.1 [G.2.1], page 1104(16).

86

- The definition of <close result set>, which determines the accuracy of certain fixed point multiplications and divisions. See Section 21.2.3 [G.2.3], page 1109(5).

87

- Conditions on a <universal_real> operand of a fixed point multiplication or division for which the result shall be in the <perfect result set>. See Section 21.2.3 [G.2.3], page 1109(22).

88

- The result of a fixed point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. See Section 21.2.3 [G.2.3], page 1109(27).

89

- The result of an elementary function reference in overflow situations, when the Machine_Overflows attribute of the result type is False. See Section 21.2.4 [G.2.4], page 1113(4).

90

- The value of the <angle threshold>, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound. See Section 21.2.4 [G.2.4], page 1113(10).

91

- The accuracy of certain elementary functions for parameters beyond the angle threshold. See Section 21.2.4 [G.2.4], page 1113(10).

92

- The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the Machine_Overflows attribute of the corresponding real type is False. See Section 21.2.6 [G.2.6], page 1116(5).

93

- The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold. See Section 21.2.6 [G.2.6], page 1116(8).

93.1/2

- The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Real_Matrix. See Section 21.3.1 [G.3.1], page 1119(81/2).

93.2/2

- The accuracy requirements for the subprograms `Solve`, `Inverse`, `Determinant`, `Eigenvalues` and `Eigensystem` for type `Complex_Matrix`. See Section 21.3.2 [G.3.2], page 1130(149/2).

93.3/2

- <This paragraph was deleted.>

93.4/2

- <This paragraph was deleted.>

93.5/2

- <This paragraph was deleted.>

93.6/2

- <This paragraph was deleted.>

93.7/2

- Implementation–defined <policy_>identifiers allowed in a pragma `Partition_Elaboration_Policy`. See Section 22.6 [H.6], page 1163(4/2).

26.3 M.3 Implementation Advice

1/2

This International Standard sometimes gives advice about handling certain target machine dependences. Each Ada implementation must document whether that advice is followed:

2/2

- `Program_Error` should be raised when an unsupported Specialized Needs Annex feature is used at run time. See Section 2.1.3 [1.1.3], page 23(20).

3/2

- Implementation–defined extensions to the functionality of a language–defined library unit should be provided by adding children to the library unit. See Section 2.1.3 [1.1.3], page 23(21).

4/2

- If a bounded error or erroneous execution is detected, `Program_Error` should be raised. See Section 2.1.5 [1.1.5], page 28(12).

5/2

- Implementation–defined pragmas should have no semantic effect for error–free programs. See Section 3.8 [2.8], page 44(16).

6/2

- Implementation–defined pragmas should not make an illegal program legal, unless they complete a declaration or configure the `library_items` in an environment. See Section 3.8 [2.8], page 44(19).

7/2

- `Long_Integer` should be declared in Standard if the target supports 32–bit arithmetic. No other named integer subtypes should be declared in Standard. See Section 4.5.4 [3.5.4], page 95(28).

8/2

- For a two’s complement target, modular types with a binary modulus up to `System.Max_Int*2+2` should be supported. A nonbinary modulus up to `Integer’Last` should be supported. See Section 4.5.4 [3.5.4], page 95(29).

9/2

- `Program_Error` should be raised for the evaluation of S’Pos for an enumeration type, if the value of the operand does not correspond to the internal code for any enumeration literal of the type. See Section 4.5.5 [3.5.5], page 99(8).

10/2

- `Long_Float` should be declared in Standard if the target supports 11 or more digits of precision. No other named float subtypes should be declared in Standard. See Section 4.5.7 [3.5.7], page 103(17).

11/2

- Multidimensional arrays should be represented in row–major order, unless the array has convention Fortran. See Section 4.6.2 [3.6.2], page 119(11).

12/2

- `Tags.Internal_Tag` should return the tag of a type whose innermost master is the master of the point of the function call.. See Section 4.9 [3.9], page 136(26.1/2).

13/2

- For a real static expression with a non–formal type that is not part of a larger static expression should be rounded the same as the target system. See Section 5.9 [4.9], page 234(38.1/2).

14/2

- The value of `Duration'Small` should be no greater than 100 microseconds. See Section 10.6 [9.6], page 358(30).

15/2

- The time base for `delay_relative_statements` should be monotonic. See Section 10.6 [9.6], page 358(31).

16/2

- Leap seconds should be supported if the target system supports them. Otherwise, operations in `Calendar.Formatting` should return results consistent with no leap seconds. See Section 10.6.1 [9.6.1], page 363(89/2).

17/2

- When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance. See Section 11.1.5 [10.1.5], page 407(10/1).

18/2

- A type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package. See Section 11.2.1 [10.2.1], page 413(12).

19/2

- `Exception_Information` should provide information useful for debugging, and should include the `Exception_Name` and `Exception_Message`. See Section 12.4.1 [11.4.1], page 423(19).

20/2

- `Exception_Message` by default should be short, provide information useful for debugging, and should not include the `Exception_Name`. See Section 12.4.1 [11.4.1], page 423(19).

21/2

- Code executed for checks that have been suppressed should be minimized. See Section 12.5 [11.5], page 431(28).

22/2

- The recommended level of support for all representation items should be followed. See Section 14.1 [13.1], page 481(28/2).

23/2

- Storage allocated to objects of a packed type should be minimized. See Section 14.2 [13.2], page 485(6).

24/2

- The recommended level of support for pragma Pack should be followed. See Section 14.2 [13.2], page 485(9).

25/2

- For an array X, X'Address should point at the first component of the array rather than the array bounds. See Section 14.3 [13.3], page 486(14).

26/2

- The recommended level of support for the Address attribute should be followed. See Section 14.3 [13.3], page 486(19).

27/2

- The recommended level of support for the Alignment attribute should be followed. See Section 14.3 [13.3], page 486(35).

28/2

- The Size of an array object should not include its bounds. See Section 14.3 [13.3], page 486(41.1/2).

29/2

- If the Size of a subtype allows for efficient independent addressability, then the Size of most objects of the subtype should equal the Size of the subtype. See Section 14.3 [13.3], page 486(52).

30/2

- A Size clause on a composite subtype should not affect the internal layout of components. See Section 14.3 [13.3], page 486(53).

31/2

- The recommended level of support for the Size attribute should be followed. See Section 14.3 [13.3], page 486(56).

32/2

- The recommended level of support for the Component_Size attribute should be followed. See Section 14.3 [13.3], page 486(73).

33/2

- The recommended level of support for `enumeration_representation_clauses` should be followed. See Section 14.4 [13.4], page 500(10).

34/2

- The recommended level of support for `record_representation_clauses` should be followed. See Section 14.5.1 [13.5.1], page 501(22).

35/2

- If a component is represented using a pointer to the actual data of the component which is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data. If a component is allocated discontinuously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes. See Section 14.5.2 [13.5.2], page 506(5).

36/2

- The recommended level of support for the nondefault bit ordering should be followed. See Section 14.5.3 [13.5.3], page 508(8).

37/2

- `Type System.Address` should be a private type. See Section 14.7 [13.7], page 510(37).

38/2

- Operations in `System` and its children should reflect the target environment; operations that do not make sense should raise `Program_Error`. See Section 14.7.1 [13.7.1], page 516(16).

39/2

- Since the `Size` of an array object generally does not include its bounds, the bounds should not be part of the converted data in an instance of `Unchecked_Conversion`. See Section 14.9 [13.9], page 520(14/2).

40/2

- There should not be unnecessary run-time checks on the result of an `Unchecked_Conversion`; the result should be returned by reference when possible. Restrictions on `Unchecked_Conversions` should be avoided. See Section 14.9 [13.9], page 520(15).

41/2

- The recommended level of support for `Unchecked_Conversion` should be followed. See Section 14.9 [13.9], page 520(17).

42/2

- Any cases in which heap storage is dynamically allocated other than as part of the evaluation of an allocator should be documented. See Section 14.11 [13.11], page 526(23).

43/2

- A default storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects. See Section 14.11 [13.11], page 526(24).

44/2

- Usually, a storage pool for an access discriminant or access parameter should be created at the point of an allocator, and be reclaimed when the designated object becomes inaccessible. For other anonymous access types, the pool should be created at the point where the type is elaborated and need not support deallocation of individual objects. See Section 14.11 [13.11], page 526(25).

45/2

- For a standard storage pool, an instance of `Unchecked_Deallocation` should actually reclaim the storage. See Section 14.11.2 [13.11.2], page 532(17).

46/2

- If not specified, the value of `Stream_Size` for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size. See Section 14.13.2 [13.13.2], page 540(1.6/2).

47/2

- The recommended level of support for the `Stream_Size` attribute should be followed. See Section 14.13.2 [13.13.2], page 540(1.8/2).

48/2

- If an implementation provides additional named predefined integer types, then the names should end with "Integer". If an implementation provides additional named predefined floating point types, then the names should end with "Float". See Section 15.1 [A.1], page 556(52).

49/2

- Implementation-defined operations on `Wide_Character`, `Wide_String`, `Wide_Wide_Character`, and `Wide_Wide_String` should be child units of `Wide_Characters` or `Wide_Wide_Characters`. See Section 15.3.1 [A.3.1], page 565(7/2).

50/2

- Bounded string objects should not be implemented by implicit pointers and dynamic allocation. See Section 15.4.4 [A.4.4], page 610(106).

51/2

- `Strings.Hash` should be good a hash function, returning a wide spread of values for different string values, and similar strings should rarely return the same value. See Section 15.4.9 [A.4.9], page 646(12/2).

52/2

- Any storage associated with an object of type `Generator` of the random number packages should be reclaimed on exit from the scope of the object. See Section 15.5.2 [A.5.2], page 654(46).

53/2

- Each value of `Initiator` passed to `Reset` for the random number packages should initiate a distinct sequence of random numbers, or, if that is not possible, be at least a rapidly varying function of the initiator value. See Section 15.5.2 [A.5.2], page 654(47).

54/2

- `Get_Immediate` should be implemented with unbuffered input; input should be available immediately; line-editing should be disabled. See Section 15.10.7 [A.10.7], page 723(23).

55/2

- `Package Directories.Information` should be provided to retrieve other information about a file. See Section 15.16 [A.16], page 757(124/2).

56/2

- `Directories.Start_Search` and `Directories.Search` should raise `Use_Error` for malformed patterns. See Section 15.16 [A.16], page 757(125/2).

57/2

- `Directories.Rename` should be supported at least when both `New_Name` and `Old_Name` are simple names and `New_Name` does not identify an existing external file. See Section 15.16 [A.16], page 757(126/2).

58/2

- If the execution environment supports subprocesses, the current environment variables should be used to initialize the environment variables of a subprocess. See Section 15.17 [A.17], page 775(32/2).

59/2

- Changes to the environment variables made outside the control of `Environment_Variables` should be reflected immediately. See Section 15.17 [A.17], page 775(33/2).

60/2

- `Containers.Hash_Type'Modulus` should be at least 2^{32} . `Containers.Count_Type'Last` should be at least $2^{31}-1$. See Section 15.18.1 [A.18.1], page 779(8/2).

61/2

- The worst-case time complexity of `Element` for `Containers.Vector` should be $O(\log N)$. See Section 15.18.2 [A.18.2], page 779(256/2).

62/2

- The worst-case time complexity of `Append` with `Count = 1` when N is less than the capacity for `Containers.Vector` should be $O(\log N)$. See Section 15.18.2 [A.18.2], page 779(257).

63/2

- The worst-case time complexity of `Prepend` with `Count = 1` and `Delete_First` with `Count=1` for `Containers.Vectors` should be $O(N \log N)$. See Section 15.18.2 [A.18.2], page 779(258/2).

64/2

- The worst-case time complexity of a call on procedure `Sort` of an instance of `Containers.Vectors.Generic_Sorting` should be $O(N^2)$, and the average time complexity should be better than $O(N^2)$. See Section 15.18.2 [A.18.2], page 779(259/2).

65/2

- `Containers.Vectors.Generic_Sorting.Sort` and `Containers.Vectors.Generic_Sorting.Merge` should minimize copying of elements. See Section 15.18.2 [A.18.2], page 779(260/2).

66/2

- `Containers.Vectors.Move` should not copy elements, and should minimize copying of internal data structures. See Section 15.18.2 [A.18.2], page 779(261/2).

67/2

- If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation. See Section 15.18.2 [A.18.2], page 779(262/2).

68/2

- The worst–case time complexity of `Element`, `Insert` with `Count=1`, and `Delete` with `Count=1` for `Containers.Doubly_Linked_Lists` should be $O(\log N)$. See Section 15.18.3 [A.18.3], page 810(160/2).

69/2

- a call on procedure `Sort` of an instance of `Containers.Doubly_Linked_Lists.Generic_Sorting` should have an average time complexity better than $O(N^2)$ and worst case no worse than $O(N^2)$. See Section 15.18.3 [A.18.3], page 810(161/2).

70/2

- `Containers.Doubly_Link_Lists.Move` should not copy elements, and should minimize copying of internal data structures. See Section 15.18.3 [A.18.3], page 810(162/2).

71/2

- If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation. See Section 15.18.3 [A.18.3], page 810(163/2).

72/2

- `Move` for a map should not copy elements, and should minimize copying of internal data structures. See Section 15.18.4 [A.18.4], page 830(83/2).

73/2

- If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation. See Section 15.18.4 [A.18.4], page 830(84/2).

74/2

- The average time complexity of `Element`, `Insert`, `Include`, `Replace`, `Delete`, `Exclude` and `Find` operations that take a key parameter for `Containers.Hashed_Maps` should be $O(\log N)$. The average time complexity of the subprograms of `Containers.Hashed_Maps` that take a cursor parameter should be $O(1)$. See Section 15.18.5 [A.18.5], page 839(62/2).

75/2

- The worst–case time complexity of `Element`, `Insert`, `Include`, `Replace`, `Delete`, `Exclude` and `Find` operations that take a key parameter for `Containers.Ordered_Maps` should be $O(N^2)$ or better. The worst–case time complexity of the subprograms of `Containers.Ordered_Maps` that take a cursor parameter should be $O(1)$. See Section 15.18.6 [A.18.6], page 846(95/2).

76/2

- Move for sets should not copy elements, and should minimize copying of internal data structures. See Section 15.18.7 [A.18.7], page 855(104/2).

77/2

- If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation. See Section 15.18.7 [A.18.7], page 855(105/2).

78/2

- The average time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations of Containers.Hashing_Sets that take an element parameter should be $O(\log N)$. The average time complexity of the subprograms of Containers.Hashing_Sets that take a cursor parameter should be $O(1)$. The average time complexity of Containers.Hashing_Sets.Reserve_Capacity should be $O(N)$. See Section 15.18.8 [A.18.8], page 867(88/2).

79/2

- The worst-case time complexity of the Insert, Include, Replace, Delete, Exclude and Find operations of Containers.Ordered_Sets that take an element parameter should be $O((\log N)^2)$. The worst-case time complexity of the subprograms of Containers.Ordered_Sets that take a cursor parameter should be $O(1)$. See Section 15.18.9 [A.18.9], page 876(116/2).

80/2

- Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should have an average time complexity better than $O(N^2)$ and worst case no worse than $O(N^2)$. See Section 15.18.16 [A.18.16], page 891(10/2).

81/2

- Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements. See Section 15.18.16 [A.18.16], page 891(11/2).

82/2

- If pragma Export is supported for a language, the main program should be able to be written in that language. Subprograms named "adainit" and "adafinal" should be provided for elaboration and finalization of the environment task. See Section 16.1 [B.1], page 894(39).

83/2

- Automatic elaboration of preelaborated packages should be provided when pragma Export is supported. See Section 16.1 [B.1], page 894(40).

84/2

- For each supported convention <L> other than Intrinsic, pragmas Import and Export should be supported for objects of <L>-compatible types and for subprograms, and pragma Convention should be supported for <L>-eligible types and for subprograms. See Section 16.1 [B.1], page 894(41).

85/2

- If an interface to C, COBOL, or Fortran is provided, the corresponding package or packages described in Chapter 16 [Annex B], page 894, "Chapter 16 [Annex B], page 894, Interface to Other Languages" should also be provided. See Section 16.2 [B.2], page 900(13).

86/2

- The constants nul, wide_nul, char16_nul, and char32_nul in package Interfaces.C should have a representation of zero. See Section 16.3 [B.3], page 901(62.1/2).

87/2

- If C interfacing is supported, the interface correspondences between Ada and C should be supported. See Section 16.3 [B.3], page 901(71).

88/2

- If COBOL interfacing is supported, the interface correspondences between Ada and COBOL should be supported. See Section 16.4 [B.4], page 931(98).

89/2

- If Fortran interfacing is supported, the interface correspondences between Ada and Fortran should be supported. See Section 16.5 [B.5], page 945(26).

90/2

- The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment. See Section 17.1 [C.1], page 949(3).

91/2

- Interface to assembler should be supported; the default assembler should be associated with the convention identifier Assembler. See Section 17.1 [C.1], page 949(4).

92/2

- If an entity is exported to assembly language, then the implementation should allocate it at an addressable location even if not otherwise referenced from the Ada code. A

call to a machine code or assembler subprogram should be treated as if it could read or update every object that is specified as exported. See Section 17.1 [C.1], page 949(5).

93/2

- Little or no overhead should be associated with calling intrinsic and machine-code subprograms. See Section 17.1 [C.1], page 949(10).

94/2

- Intrinsic subprograms should be provided to access any machine operations that provide special capabilities or efficiency not normally available. See Section 17.1 [C.1], page 949(16).

95/2

- If the Ceiling-Locking policy is not in effect and the target system allows for finer-grained control of interrupt blocking, a means for the application to specify which interrupts are to be blocked during protected actions should be provided. See Section 17.3 [C.3], page 951(28/2).

96/2

- Interrupt handlers should be called directly by the hardware. See Section 17.3.1 [C.3.1], page 954(20).

97/2

- Violations of any implementation-defined restrictions on interrupt handlers should be detected before run time. See Section 17.3.1 [C.3.1], page 954(21).

98/2

- If implementation-defined forms of interrupt handler procedures are supported, then for each such form of a handler, a type analogous to `Parameterless_Handler` should be specified in a child package of `Interrupts`, with the same operations as in the predefined package `Interrupts`. See Section 17.3.2 [C.3.2], page 957(25).

99/2

- Preelaborated packages should be implemented such that little or no code is executed at run time for the elaboration of entities. See Section 17.4 [C.4], page 960(14).

100/2

- If `pragma Discard_Names` applies to an entity, then the amount of storage used for storing names associated with that entity should be reduced. See Section 17.5 [C.5], page 961(8).

101/2

- A load or store of a volatile object whose size is a multiple of `System.Storage_Unit` and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others. See Section 17.6 [C.6], page 962(22/2).

102/2

- A load or store of an atomic object should be implemented by a single load or store instruction. See Section 17.6 [C.6], page 962(23/2).

103/2

- If the target domain requires deterministic memory use at run time, storage for task attributes should be pre-allocated statically and the number of attributes pre-allocated should be documented. See Section 17.7.2 [C.7.2], page 967(30).

104/2

- Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination. See Section 17.7.2 [C.7.2], page 967(30.1/2).

105/2

- Names that end with "_Locking" should be used for implementation-defined locking policies. See Section 18.3 [D.3], page 991(17).

106/2

- Names that end with "_Queuing" should be used for implementation-defined queuing policies. See Section 18.4 [D.4], page 994(16).

107/2

- The `abort_statement` should not require the task executing the statement to block. See Section 18.6 [D.6], page 1000(9).

108/2

- On a multi-processor, the delay associated with aborting a task on another processor should be bounded. See Section 18.6 [D.6], page 1000(10).

109/2

- When feasible, specified restrictions should be used to produce a more efficient implementation. See Section 18.7 [D.7], page 1001(21).

110/2

- When appropriate, mechanisms to change the value of Tick should be provided. See Section 18.8 [D.8], page 1008(47).

111/2

- Calendar.Clock and Real_Time.Clock should be transformations of the same time base. See Section 18.8 [D.8], page 1008(48).

112/2

- The "best" time base which exists in the underlying system should be available to the application through Real_Time.Clock. See Section 18.8 [D.8], page 1008(49).

113/2

- When appropriate, implementations should provide configuration mechanisms to change the value of Execution_Time.CPU_Tick. See Section 18.14 [D.14], page 1021(29/2).

114/2

- For a timing event, the handler should be executed directly by the real-time clock interrupt mechanism. See Section 18.15 [D.15], page 1031(25).

115/2

- The PCS should allow for multiple tasks to call the RPC-receiver. See Section 19.5 [E.5], page 1050(28).

116/2

- The System.RPC.Write operation should raise Storage_Error if it runs out of space when writing an item. See Section 19.5 [E.5], page 1050(29).

117/2

- If COBOL (respectively, C) is supported in the target environment, then interfacing to COBOL (respectively, C) should be supported as specified in Chapter 16 [Annex B], page 894. See Section 30.7 [F], page 1451(7).

118/2

- Packed decimal should be used as the internal representation for objects of subtype <S> when <S>'Machine_Radix = 10. See Section 20.1 [F.1], page 1054(2).

119/2

- If Fortran (respectively, C) is supported in the target environment, then interfacing to Fortran (respectively, C) should be supported as specified in Chapter 16 [Annex B], page 894. See Section 30.8 [G], page 1458(7).

120/2

- Mixed real and complex operations (as well as pure–imaginary and complex operations) should not be performed by converting the real (resp. pure–imaginary) operand to complex. See Section 21.1.1 [G.1.1], page 1084(56).

121/2

- If `Real'Signed_Zeros` is true for `Numerics.Generic_Complex_Types`, a rational treatment of the signs of zero results and result components should be provided. See Section 21.1.1 [G.1.1], page 1084(58).

122/2

- If `Complex_Types.Real'Signed_Zeros` is true for `Numerics.Generic_Complex-Elementary_Functions`, a rational treatment of the signs of zero results and result components should be provided. See Section 21.1.2 [G.1.2], page 1091(49).

123/2

- For elementary functions, the forward trigonometric functions without a `Cycle` parameter should not be implemented by calling the corresponding version with a `Cycle` parameter. `Log` without a `Base` parameter should not be implemented by calling `Log` with a `Base` parameter. See Section 21.2.4 [G.2.4], page 1113(19).

124/2

- For complex arithmetic, the `Compose_From_Polar` function without a `Cycle` parameter should not be implemented by calling `Compose_From_Polar` with a `Cycle` parameter. See Section 21.2.6 [G.2.6], page 1116(15).

125/2

- `Solve` and `Inverse` for `Numerics.Generic_Real_Arrays` should be implemented using established techniques such as LU decomposition and the result should be refined by an iteration on the residuals. See Section 21.3.1 [G.3.1], page 1119(88/2).

126/2

- The equality operator should be used to test that a matrix in `Numerics.Generic_Real_Matrix` is symmetric. See Section 21.3.1 [G.3.1], page 1119(90/2).

127/2

- Solve and Inverse for Numerics.Generic_Complex_Arrays should be implemented using established techniques and the result should be refined by an iteration on the residuals. See Section 21.3.2 [G.3.2], page 1130(158/2).

128/2

- The equality and negation operators should be used to test that a matrix is Hermitian. See Section 21.3.2 [G.3.2], page 1130(160/2).

129/2

- Mixed real and complex operations should not be performed by converting the real operand to complex. See Section 21.3.2 [G.3.2], page 1130(161/2).

130/2

- The information produced by pragma Reviewable should be provided in both a human-readable and machine-readable form, and the latter form should be documented. See Section 22.3.1 [H.3.1], page 1155(19).

131/2

- Object code listings should be provided both in a symbolic format and in a numeric format. See Section 22.3.1 [H.3.1], page 1155(20).

132/2

- If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration then the partition should be immediately terminated. See Section 22.6 [H.6], page 1163(15/2).

27 Annex N Glossary

1/2

This Annex contains informal descriptions of some of the terms used in this International Standard. The index provides references to more formal definitions of all of the terms used in this International Standard.

1.1/2

Abstract type. An abstract type is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own.

2

Access type. An access type has values that designate aliased objects. Access types correspond to "pointer types" or "reference types" in some other languages.

3

Aliased. An aliased view of an object is one that can be designated by an access value. Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word aliased. The Access attribute can be used to create an access value designating an aliased object.

3.1/2

Ancestor. An ancestor of a type is the type itself or, in the case of a type derived from other types, its parent type or one of its progenitor types or one of their ancestors. Note that ancestor and descendant are inverse relationships.

4

Array type. An array type is a composite type whose components are all of the same type. Components are selected by indexing.

4.1/2

Category (of types). A category of types is a set of types with one or more common properties, such as primitive operations. A category of types that is closed under derivation is also known as a <class>.

5

Character type. A character type is an enumeration type whose values include characters.

6/2

Class (of types). A class is a set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class. The set of types of a class share common properties, such as their primitive operations.

7

Compilation unit. The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation_units. A compilation_unit contains either the declaration, the body, or a renaming of a program unit.

8/2

Composite type. A composite type may have components.

9

Construct. A <construct> is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under "Syntax".

10

Controlled type. A controlled type supports user-defined assignment and finalization. Objects are always finalized before being destroyed.

11

Declaration. A <declaration> is a language construct that associates a name with (a view of) an entity. A declaration may appear explicitly in the program text (an <explicit> declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an <implicit> declaration).

12/2

<This paragraph was deleted.>

13/2

Derived type. A derived type is a type defined in terms of one or more other types given in a derived type definition. The first of those types is the parent type of the derived type and any others are progenitor types. Each class containing the parent type or a progenitor type also contains the derived type. The derived type inherits properties such as components and primitive operations from the parent and progenitors. A type together with the types derived from it (directly or indirectly) form a derivation class.

13.1/2

Descendant. A type is a descendant of itself, its parent and progenitor types, and their ancestors. Note that descendant and ancestor are inverse relationships.

14

Discrete type. A discrete type is either an integer type or an enumeration type. Discrete types may be used, for example, in case statements and as array indices.

15/2

Discriminant. A discriminant is a parameter for a composite type. It can control, for example, the bounds of a component of the type if the component is an array. A discriminant for a task type can be used to pass data to a task of the type upon creation.

15.1/2

Elaboration. The process by which a declaration achieves its run-time effect is called elaboration. Elaboration is one of the forms of execution.

16

Elementary type. An elementary type does not have components.

17

Enumeration type. An enumeration type is defined by an enumeration of its values, which may be named by identifiers or character literals.

17.1/2

Evaluation. The process by which an expression achieves its run-time effect is called evaluation. Evaluation is one of the forms of execution.

18

Exception. An <exception> represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an <exception occurrence>. To <raise> an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called <handling> the exception.

19

Execution. The process by which a construct achieves its run-time effect is called <execution>. Execution of a declaration is also called <elaboration>. Execution of an expression is also called <evaluation>.

19.1/2

Function. A function is a form of subprogram that returns a result and can be called as part of an expression.

20

Generic unit. A generic unit is a template for a (nongeneric) program unit; the template can be parameterized by objects, types, subprograms, and packages. An instance of a generic unit is created by a `generic_instantiation`. The rules of the language are enforced when a generic unit is compiled, using a generic contract model; additional checks are performed upon instantiation to verify the contract is met. That is, the declaration of a generic unit represents a contract between the body of the generic and instances of the generic. Generic units can be used to perform the role that macros sometimes play in other languages.

20.1/2

Incomplete type. An incomplete type gives a view of a type that reveals only some of its properties. The remaining properties are provided by the full view given elsewhere. Incomplete types can be used for defining recursive data structures.

21

Integer type. Integer types comprise the signed integer types and the modular types. A signed integer type has a base range that includes both positive and negative numbers, and has operations that may raise an exception when the result is outside the base range. A modular type has a base range whose lower bound is zero, and has operations with "wraparound" semantics. Modular types subsume what are called "unsigned types" in some other languages.

21.1/2

Interface type. An interface type is a form of abstract tagged type which has no components or concrete operations except possibly null procedures. Interface types are used for composing other interfaces and tagged types and thereby provide multiple inheritance. Only an interface type can be used as a progenitor of another type.

22

Library unit. A library unit is a separately compiled program unit, and is always a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a <subsystem>.

23/2

Limited type. A limited type is a type for which copying (such as in an `assignment_statement`) is not allowed. A nonlimited type is a type for which copying is allowed.

24

Object. An object is either a constant or a variable. An object contains a value. An object is created by an `object_declaration` or by an allocator. A formal parameter is (a view of) an object. A subcomponent of an object is an object.

24.1/2

Overriding operation. An overriding operation is one that replaces an inherited primitive operation. Operations may be marked explicitly as overriding or not overriding.

25

Package. Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.

25.1/2

Parent. The parent of a derived type is the first type given in the definition of the derived type. The parent can be almost any kind of type, including an interface type.

26

Partition. A <partition> is a part of a program. Each partition consists of a set of library units. Each partition may run in a separate address space, possibly on a separate computer. A program may contain just one partition. A distributed program typically contains multiple partitions, which can execute concurrently.

27

Pragma. A pragma is a compiler directive. There are language–defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation–defined) pragmas.

28

Primitive operations. The primitive operations of a type are the operations (such as subprograms) declared together with the type declaration. They are inherited by other types in the same class of types. For a tagged type, the primitive subprograms are dispatching subprograms, providing run–time polymorphism. A dispatching subprogram may be called with statically tagged operands, in which case the subprogram body invoked is determined at compile time. Alternatively, a dispatching subprogram may be called using a dispatching call, in which case the subprogram body invoked is determined at run time.

29/2

Private extension. A private extension is a type that extends another type, with the additional properties hidden from its clients.

30/2

Private type. A private type gives a view of a type that reveals only some of its properties. The remaining properties are provided by the full view given elsewhere. Private types can be used for defining abstractions that hide unnecessary details from their clients.

30.1/2

Procedure. A procedure is a form of subprogram that does not return a result and can only be called by a statement.

30.2/2

Progenitor. A progenitor of a derived type is one of the types given in the definition of the derived type other than the first. A progenitor is always an interface type. Interfaces, tasks, and protected types may also have progenitors.

31

Program. A <program> is a set of <partitions>, each of which may execute in a separate

address space, possibly on a separate computer. A partition consists of a set of library units.

32

Program unit. A <program unit> is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units.

33/2

Protected type. A protected type is a composite type whose components are accessible only through one of its protected operations which synchronize concurrent access by multiple tasks.

34

Real type. A real type has values that are approximations of the real numbers. Floating point and fixed point types are real types.

35

Record extension. A record extension is a type that extends another type by adding additional components.

36

Record type. A record type is a composite type consisting of zero or more named components, possibly of different types.

36.1/2

Renaming. A renaming_declaration is a declaration that does not define a new entity, but instead defines a view of an existing entity.

37

Scalar type. A scalar type is either a discrete type or a real type.

37.1/2

Subprogram. A subprogram is a section of a program that can be executed in various contexts. It is invoked by a subprogram call that may qualify the effect of the subprogram through the passing of parameters. There are two forms of subprograms: functions, which return values, and procedures, which do not.

38/2

Subtype. A subtype is a type together with a constraint or null exclusion, which constrains the values of the subtype to satisfy a certain condition. The values of a subtype are a subset of the values of its type.

38.1/2

Synchronized. A synchronized entity is one that will work safely with multiple tasks at one time. A synchronized interface can be an ancestor of a task or a protected type. Such a task or protected type is called a synchronized tagged type.

39

Tagged type. The objects of a tagged type have a run-time type tag, which indicates the specific type with which the object was originally created. An operand of a class-wide tagged type can be used in a dispatching call; the tag indicates which subprogram body to invoke. Nondispatching calls, in which the subprogram body to invoke is determined at compile time, are also allowed. Tagged types may be extended with additional components.

40/2

Task type. A task type is a composite type used to represent active entities which execute concurrently and which can communicate via queued task entries. The top-level task of a partition is called the environment task.

41/2

Type. Each object has a type. A <type> has an associated set of values, and a set of <primitive operations> which implement the fundamental aspects of its semantics. Types are grouped into <categories>. Most language-defined categories of types are also <classes> of types.

42/2

View. A view of an entity reveals some or all of the properties of the entity. A single entity may have multiple views.

28 Annex P Syntax Summary

This Annex summarizes the complete syntax of the language. See Section 2.1.4 [1.1.4], page 25, for a description of the notation used.

Section 3.3 [2.3], page 38:

```
identifier ::=
    identifier_start {identifier_start | identifier_extend}
```

Section 3.3 [2.3], page 38:

```
identifier_start ::=
    letter_uppercase
    | letter_lowercase
    | letter_titlecase
    | letter_modifier
    | letter_other
    | number_letter
```

Section 3.3 [2.3], page 38:

```
identifier_extend ::=
    mark_non_spacing
    | mark_spacing_combining
    | number_decimal
    | punctuation_connector
    | other_format
```

Section 3.4 [2.4], page 39:

```
numeric_literal ::= decimal_literal | based_literal
```

Section 3.4.1 [2.4.1], page 40:

```
decimal_literal ::= numeral [.numeral] [exponent]
```

Section 3.4.1 [2.4.1], page 40:

```
numeral ::= digit {[underline] digit}
```

Section 3.4.1 [2.4.1], page 40:

```
exponent ::= E [+] numeral | E - numeral
```

Section 3.4.1 [2.4.1], page 40:

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Section 3.4.2 [2.4.2], page 41:

```
based_literal ::=
    base # based_numeral [.based_numeral] # [exponent]
```

Section 3.4.2 [2.4.2], page 41:

```
base ::= numeral
```

Section 3.4.2 [2.4.2], page 41:

```
based_numeral ::=
    extended_digit {[underline] extended_digit}
```

Section 3.4.2 [2.4.2], page 41:

```
extended_digit ::= digit | A | B | C | D | E | F
```

Section 3.5 [2.5], page 42:

```
character_literal ::= 'graphic_character'
```

Section 3.6 [2.6], page 42:

string_literal ::= "{string_element}"

Section 3.6 [2.6], page 42:

string_element ::= "" | <non_quotation_mark_>graphic_character

Section 3.7 [2.7], page 43:

comment ::= --{<non_end_of_line_>character}

Section 3.8 [2.8], page 44:

pragma ::=

pragma identifier [(pragma_argument_association {, pragma_argument_association})];

Section 3.8 [2.8], page 44:

pragma_argument_association ::=

[<pragma_argument_>identifier =>] name

| [<pragma_argument_>identifier =>] expression

Section 4.1 [3.1], page 49:

basic_declaration ::=

type_declaration | subtype_declaration

| object_declaration | number_declaration

| subprogram_declaration | abstract_subprogram_declaration

| null_procedure_declaration | package_declaration

| renaming_declaration | exception_declaration

| generic_declaration | generic_instantiation

Section 4.1 [3.1], page 49:

defining_identifier ::= identifier

Section 4.2.1 [3.2.1], page 53:

type_declaration ::= full_type_declaration

| incomplete_type_declaration

| private_type_declaration

| private_extension_declaration

Section 4.2.1 [3.2.1], page 53:

full_type_declaration ::=

type defining_identifier [known_discriminant_part] is type_definition; █

| task_type_declaration

| protected_type_declaration

Section 4.2.1 [3.2.1], page 53:

type_definition ::=

enumeration_type_definition | integer_type_definition

| real_type_definition | array_type_definition

| record_type_definition | access_type_definition

| derived_type_definition | interface_type_definition

Section 4.2.2 [3.2.2], page 55:

subtype_declaration ::=

subtype defining_identifier is subtype_indication;

Section 4.2.2 [3.2.2], page 55:

subtype_indication ::= [null_exclusion] subtype_mark [constraint]

Section 4.2.2 [3.2.2], page 55:

subtype_mark ::= <subtype_>name

Section 4.2.2 [3.2.2], page 55:

constraint ::= scalar_constraint | composite_constraint

Section 4.2.2 [3.2.2], page 55:

scalar_constraint ::=
 range_constraint | digits_constraint | delta_constraint

Section 4.2.2 [3.2.2], page 55:

composite_constraint ::=
 index_constraint | discriminant_constraint

Section 4.3.1 [3.3.1], page 61:

object_declaration ::=
 defining_identifier_list : [aliased] [constant] subtype_indication [:= expression];

 | defining_identifier_list : [aliased] [constant] access_definition [:= expression];

 | defining_identifier_list : [aliased] [constant] array_type_definition [:= expression];

 | single_task_declaration
 | single_protected_declaration

Section 4.3.1 [3.3.1], page 61:

defining_identifier_list ::=
 defining_identifier {, defining_identifier}

Section 4.3.2 [3.3.2], page 65:

number_declaration ::=
 defining_identifier_list : constant := <static_>expression;

Section 4.4 [3.4], page 66:

derived_type_definition ::=
 [abstract] [limited] new <parent_>subtype_indication [[and interface_list] record_extension_part]

Section 4.5 [3.5], page 76:

range_constraint ::= range range

Section 4.5 [3.5], page 76:

range ::= range_attribute_reference
 | simple_expression .. simple_expression

Section 4.5.1 [3.5.1], page 92:

enumeration_type_definition ::=
 (enumeration_literal_specification {, enumeration_literal_specification})

Section 4.5.1 [3.5.1], page 92:

enumeration_literal_specification ::= defining_identifier | defining_character_literal

Section 4.5.1 [3.5.1], page 92:

defining_character_literal ::= character_literal

Section 4.5.4 [3.5.4], page 95:

integer_type_definition ::= signed_integer_type_definition | modular_type_definition

Section 4.5.4 [3.5.4], page 95:
signed_integer_type_definition ::= range <static_>simple_expression .. <static_>simple_expression

Section 4.5.4 [3.5.4], page 95:
modular_type_definition ::= mod <static_>expression

Section 4.5.6 [3.5.6], page 102:
real_type_definition ::= floating_point_definition | fixed_point_definition

Section 4.5.7 [3.5.7], page 103:
floating_point_definition ::= digits <static_>expression [real_range_specification]

Section 4.5.7 [3.5.7], page 103:
real_range_specification ::= range <static_>simple_expression .. <static_>simple_expression

Section 4.5.9 [3.5.9], page 106:
fixed_point_definition ::= ordinary_fixed_point_definition | decimal_fixed_point_definition

Section 4.5.9 [3.5.9], page 106:
ordinary_fixed_point_definition ::= delta <static_>expression real_range_specification

Section 4.5.9 [3.5.9], page 106:
decimal_fixed_point_definition ::= delta <static_>expression digits <static_>expression [real_range_specification]

Section 4.5.9 [3.5.9], page 106:
digits_constraint ::= digits <static_>expression [range_constraint]

Section 4.6 [3.6], page 114:
array_type_definition ::= unconstrained_array_definition | constrained_array_definition

Section 4.6 [3.6], page 114:
unconstrained_array_definition ::= array(index_subtype_definition {, index_subtype_definition}) of component_definition

Section 4.6 [3.6], page 114:
index_subtype_definition ::= subtype_mark range <>

Section 4.6 [3.6], page 114:
constrained_array_definition ::= array (discrete_subtype_definition {, discrete_subtype_definition}) of component_definition

Section 4.6 [3.6], page 114:
discrete_subtype_definition ::= <discrete_>subtype_indication | range

Section 4.6 [3.6], page 114:
component_definition ::= [aliased] subtype_indication
| [aliased] access_definition

Section 4.6.1 [3.6.1], page 117:
index_constraint ::= (discrete_range {, discrete_range})

Section 4.6.1 [3.6.1], page 117:

discrete_range ::= <discrete_>subtype_indication | range

Section 4.7 [3.7], page 123:

discriminant_part ::= unknown_discriminant_part | known_discriminant_part

Section 4.7 [3.7], page 123:

unknown_discriminant_part ::= (<>)

Section 4.7 [3.7], page 123:

known_discriminant_part ::=

(discriminant_specification {; discriminant_specification})

Section 4.7 [3.7], page 123:

discriminant_specification ::=

defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]

| defining_identifier_list : access_definition [:= default_expression]

Section 4.7 [3.7], page 123:

default_expression ::= expression

Section 4.7.1 [3.7.1], page 127:

discriminant_constraint ::=

(discriminant_association {, discriminant_association})

Section 4.7.1 [3.7.1], page 127:

discriminant_association ::=

[<discriminant_>selector_name { | <discriminant_>selector_name } =>] expression

Section 4.8 [3.8], page 130:

record_type_definition ::= [[abstract] tagged] [limited] record_definition

Section 4.8 [3.8], page 130:

record_definition ::=

record

component_list

end record

| null record

Section 4.8 [3.8], page 130:

component_list ::=

component_item {component_item}

| {component_item} variant_part

| null;

Section 4.8 [3.8], page 130:

component_item ::= component_declaration | aspect_clause

Section 4.8 [3.8], page 130:

component_declaration ::=

defining_identifier_list : component_definition [:= default_expression];

Section 4.8.1 [3.8.1], page 134:

variant_part ::=

case <discriminant_>direct_name is

variant

```

    {variant}
  end case;

```

Section 4.8.1 [3.8.1], page 134:
variant ::=

```

  when discrete_choice_list =>
    component_list

```

Section 4.8.1 [3.8.1], page 134:
discrete_choice_list ::= discrete_choice { | discrete_choice }

Section 4.8.1 [3.8.1], page 134:
discrete_choice ::= expression | discrete_range | others

Section 4.9.1 [3.9.1], page 143:
record_extension_part ::= with record_definition

Section 4.9.3 [3.9.3], page 149:
abstract_subprogram_declaration ::=

```

  [overriding_indicator]
  subprogram_specification is abstract;

```

Section 4.9.4 [3.9.4], page 152:
interface_type_definition ::=

```

  [limited | task | protected | synchronized] interface [and interface_list]

```

Section 4.9.4 [3.9.4], page 152:
interface_list ::= <interface_>subtype_mark {and <interface_>subtype_mark}

Section 4.10 [3.10], page 156:
access_type_definition ::=

```

  [null_exclusion] access_to_object_definition
  | [null_exclusion] access_to_subprogram_definition

```

Section 4.10 [3.10], page 156:
access_to_object_definition ::=

```

  access [general_access_modifier] subtype_indication

```

Section 4.10 [3.10], page 156:
general_access_modifier ::= all | constant

Section 4.10 [3.10], page 156:
access_to_subprogram_definition ::=

```

  access [protected] procedure parameter_profile
  | access [protected] function parameter_and_result_profile

```

Section 4.10 [3.10], page 156:
null_exclusion ::= not null

Section 4.10 [3.10], page 156:
access_definition ::=

```

  [null_exclusion] access [constant] subtype_mark
  | [null_exclusion] access [protected] procedure parameter_profile
  | [null_exclusion] access [protected] function parameter_and_result_profile

```

Section 4.10.1 [3.10.1], page 160:
incomplete_type_declaration ::= type defining_identifier [discriminant_part] [is tagged];

Section 4.11 [3.11], page 175:

declarative_part ::= {declarative_item}

Section 4.11 [3.11], page 175:

declarative_item ::=

basic_declarative_item | body

Section 4.11 [3.11], page 175:

basic_declarative_item ::=

basic_declaration | aspect_clause | use_clause

Section 4.11 [3.11], page 175:

body ::= proper_body | body_stub

Section 4.11 [3.11], page 175:

proper_body ::=

subprogram_body | package_body | task_body | protected_body

Section 5.1 [4.1], page 179:

name ::=

direct_name | explicit_dereference

| indexed_component | slice

| selected_component | attribute_reference

| type_conversion | function_call

| character_literal

Section 5.1 [4.1], page 179:

direct_name ::= identifier | operator_symbol

Section 5.1 [4.1], page 179:

prefix ::= name | implicit_dereference

Section 5.1 [4.1], page 179:

explicit_dereference ::= name.all

Section 5.1 [4.1], page 179:

implicit_dereference ::= name

Section 5.1.1 [4.1.1], page 181:

indexed_component ::= prefix(expression {, expression})

Section 5.1.2 [4.1.2], page 182:

slice ::= prefix(discrete_range)

Section 5.1.3 [4.1.3], page 183:

selected_component ::= prefix . selector_name

Section 5.1.3 [4.1.3], page 183:

selector_name ::= identifier | character_literal | operator_symbol

Section 5.1.4 [4.1.4], page 187:

attribute_reference ::= prefix'attribute_designator

Section 5.1.4 [4.1.4], page 187:

attribute_designator ::=

identifier[(<static_>expression)]

| Access | Delta | Digits

Section 5.1.4 [4.1.4], page 187:

range_attribute_reference ::= prefix'range_attribute_designator

Section 5.1.4 [4.1.4], page 187:

range_attribute_designator ::= Range[(`<static_>`expression)]

Section 5.3 [4.3], page 190:

aggregate ::= record_aggregate | extension_aggregate | array_aggregate

Section 5.3.1 [4.3.1], page 191:

record_aggregate ::= (record_component_association_list)

Section 5.3.1 [4.3.1], page 191:

record_component_association_list ::=
 record_component_association {, record_component_association}
 | null record

Section 5.3.1 [4.3.1], page 191:

record_component_association ::=
 [component_choice_list =>] expression
 | component_choice_list => <>

Section 5.3.1 [4.3.1], page 191:

component_choice_list ::=
 <component_>selector_name { | <component_>selector_name }
 | others

Section 5.3.2 [4.3.2], page 194:

extension_aggregate ::=
 (ancestor_part with record_component_association_list)

Section 5.3.2 [4.3.2], page 194:

ancestor_part ::= expression | subtype_mark

Section 5.3.3 [4.3.3], page 196:

array_aggregate ::=
 positional_array_aggregate | named_array_aggregate

Section 5.3.3 [4.3.3], page 196:

positional_array_aggregate ::=
 (expression, expression {, expression})
 | (expression {, expression}, others => expression)
 | (expression {, expression}, others => <>)

Section 5.3.3 [4.3.3], page 196:

named_array_aggregate ::=
 (array_component_association {, array_component_association})

Section 5.3.3 [4.3.3], page 196:

array_component_association ::=
 discrete_choice_list => expression
 | discrete_choice_list => <>

Section 5.4 [4.4], page 201:

expression ::=
 relation {and relation} | relation {and then relation}
 | relation {or relation} | relation {or else relation}
 | relation {xor relation}

Section 5.4 [4.4], page 201:

```
relation ::=
    simple_expression [relational_operator simple_expression]
    | simple_expression [not] in range
    | simple_expression [not] in subtype_mark
```

Section 5.4 [4.4], page 201:

```
simple_expression ::= [unary_adding_operator] term {binary_adding_operator term}
```

Section 5.4 [4.4], page 201:

```
term ::= factor {multiplying_operator factor}
```

Section 5.4 [4.4], page 201:

```
factor ::= primary [** primary] | abs primary | not primary
```

Section 5.4 [4.4], page 201:

```
primary ::=
    numeric_literal | null | string_literal | aggregate
    | name | qualified_expression | allocator | (expression)
```

Section 5.5 [4.5], page 203:

```
logical_operator ::= and | or | xor
```

Section 5.5 [4.5], page 203:

```
relational_operator ::= = | /= | < | <= | > | >=
```

Section 5.5 [4.5], page 203:

```
binary_adding_operator ::= + | - | &
```

Section 5.5 [4.5], page 203:

```
unary_adding_operator ::= + | -
```

Section 5.5 [4.5], page 203:

```
multiplying_operator ::= * | / | mod | rem
```

Section 5.5 [4.5], page 203:

```
highest_precedence_operator ::= ** | abs | not
```

Section 5.6 [4.6], page 219:

```
type_conversion ::=
    subtype_mark(expression)
    | subtype_mark(name)
```

Section 5.7 [4.7], page 229:

```
qualified_expression ::=
    subtype_mark'(expression) | subtype_mark'aggregate
```

Section 5.8 [4.8], page 230:

```
allocator ::=
    new subtype_indication | new qualified_expression
```

Section 6.1 [5.1], page 240:

```
sequence_of_statements ::= statement {statement}
```

Section 6.1 [5.1], page 240:

```
statement ::=
    {label} simple_statement | {label} compound_statement
```

Section 6.1 [5.1], page 240:

```
simple_statement ::= null_statement
```

| assignment_statement | exit_statement
| goto_statement | procedure_call_statement
| simple_return_statement | entry_call_statement
| requeue_statement | delay_statement
| abort_statement | raise_statement
| code_statement

Section 6.1 [5.1], page 240:

compound_statement ::=
 if_statement | case_statement
 | loop_statement | block_statement
 | extended_return_statement
 | accept_statement | select_statement

Section 6.1 [5.1], page 240:

null_statement ::= null;

Section 6.1 [5.1], page 240:

label ::= <<<label>statement_identifier>>

Section 6.1 [5.1], page 240:

statement_identifier ::= direct_name

Section 6.2 [5.2], page 242:

assignment_statement ::=
 <variable>name := expression;

Section 6.3 [5.3], page 245:

if_statement ::=
 if condition then
 sequence_of_statements
 {elsif condition then
 sequence_of_statements}
 [else
 sequence_of_statements]
 end if;

Section 6.3 [5.3], page 245:

condition ::= <boolean>expression

Section 6.4 [5.4], page 246:

case_statement ::=
 case expression is
 case_statement_alternative
 {case_statement_alternative}
 end case;

Section 6.4 [5.4], page 246:

case_statement_alternative ::=
 when discrete_choice_list =>
 sequence_of_statements

Section 6.5 [5.5], page 248:

loop_statement ::=
 [<loop>statement_identifier:]

```
[iteration_scheme] loop
    sequence_of_statements
end loop [<loop->identifier];
```

Section 6.5 [5.5], page 248:

```
iteration_scheme ::= while condition
    | for loop_parameter_specification
```

Section 6.5 [5.5], page 248:

```
loop_parameter_specification ::=
    defining_identifier in [reverse] discrete_subtype_definition
```

Section 6.6 [5.6], page 251:

```
block_statement ::=
    [<block->statement_identifier:]
    [declare
        declarative_part]
    begin
        handled_sequence_of_statements
    end [<block->identifier];
```

Section 6.7 [5.7], page 252:

```
exit_statement ::=
    exit [<loop->name] [when condition];
```

Section 6.8 [5.8], page 253:

```
goto_statement ::= goto <label->name;
```

Section 7.1 [6.1], page 255:

```
subprogram_declaration ::=
    [overriding_indicator]
    subprogram_specification;
```

Section 7.1 [6.1], page 255:

```
subprogram_specification ::=
    procedure_specification
    | function_specification
```

Section 7.1 [6.1], page 255:

```
procedure_specification ::= procedure defining_program_unit_name parameter_profile
```

Section 7.1 [6.1], page 255:

```
function_specification ::= function defining_designator parameter_and_result_profile
```

Section 7.1 [6.1], page 255:

```
designator ::= [parent_unit_name . ]identifier | operator_symbol
```

Section 7.1 [6.1], page 255:

```
defining_designator ::= defining_program_unit_name | defining_operator_symbol
```

Section 7.1 [6.1], page 255:

```
defining_program_unit_name ::= [parent_unit_name . ]defining_identifier
```

Section 7.1 [6.1], page 255:

```
operator_symbol ::= string_literal
```

Section 7.1 [6.1], page 255:

```
defining_operator_symbol ::= operator_symbol
```

Section 7.1 [6.1], page 255:

parameter_profile ::= [formal_part]

Section 7.1 [6.1], page 255:

parameter_and_result_profile ::=
 [formal_part] return [null_exclusion] subtype_mark
 | [formal_part] return access_definition

Section 7.1 [6.1], page 255:

formal_part ::=
 (parameter_specification {; parameter_specification})

Section 7.1 [6.1], page 255:

parameter_specification ::=
 defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]

 | defining_identifier_list : access_definition [:= default_expression]

Section 7.1 [6.1], page 255:

mode ::= [in] | in out | out

Section 7.3 [6.3], page 261:

subprogram_body ::=
 [overriding_indicator]
 subprogram_specification is
 declarative_part
 begin
 handled_sequence_of_statements
 end [designator];

Section 7.4 [6.4], page 266:

procedure_call_statement ::=
 <procedure_>name;
 | <procedure_>prefix actual_parameter_part;

Section 7.4 [6.4], page 266:

function_call ::=
 <function_>name
 | <function_>prefix actual_parameter_part

Section 7.4 [6.4], page 266:

actual_parameter_part ::=
 (parameter_association {, parameter_association})

Section 7.4 [6.4], page 266:

parameter_association ::=
 [<formal_parameter_>selector_name =>] explicit_actual_parameter

Section 7.4 [6.4], page 266:

explicit_actual_parameter ::= expression | <variable_>name

Section 7.5 [6.5], page 272:

simple_return_statement ::= return [expression];

Section 7.5 [6.5], page 272:

extended_return_statement ::=

return defining_identifier : [aliased] return_subtype_indication [:= expression] [do

 handled_sequence_of_statements
end return];

Section 7.5 [6.5], page 272:

return_subtype_indication ::= subtype_indication | access_definition

Section 7.7 [6.7], page 277:

null_procedure_declaration ::=
 [overriding_indicator]
 procedure_specification is null;

Section 8.1 [7.1], page 279:

package_declaration ::= package_specification;

Section 8.1 [7.1], page 279:

package_specification ::=
 package_defining_program_unit_name is
 {basic_declarative_item}
 [private
 {basic_declarative_item}]
 end [[parent_unit_name.]identifier]

Section 8.2 [7.2], page 281:

package_body ::=
 package_body_defining_program_unit_name is
 declarative_part
 [begin
 handled_sequence_of_statements]
 end [[parent_unit_name.]identifier];

Section 8.3 [7.3], page 283:

private_type_declaration ::=
 type_defining_identifier [discriminant_part] is [[abstract] tagged] [limited] private;

Section 8.3 [7.3], page 283:

private_extension_declaration ::=
 type_defining_identifier [discriminant_part] is
 [abstract] [limited | synchronized] new <ancestor->subtype_indication

 [and interface_list] with private;

Section 9.3.1 [8.3.1], page 312:

overriding_indicator ::= [not] overriding

Section 9.4 [8.4], page 314:

use_clause ::= use_package_clause | use_type_clause

Section 9.4 [8.4], page 314:

use_package_clause ::= use <package->name {, <package->name};

Section 9.4 [8.4], page 314:

use_type_clause ::= use type subtype_mark {, subtype_mark};

Section 9.5 [8.5], page 316:

renaming_declaration ::=

```
    object_renaming_declaration
  | exception_renaming_declaration
  | package_renaming_declaration
  | subprogram_renaming_declaration
  | generic_renaming_declaration
```

Section 9.5.1 [8.5.1], page 317:

```
object_renaming_declaration ::=
    defining_identifier : [null_exclusion] subtype_mark renames <object_>name;
```

```
    | defining_identifier : access_definition renames <object_>name;
```

Section 9.5.2 [8.5.2], page 318:

```
exception_renaming_declaration ::= defining_identifier : exception renames <exception_>name;
```

Section 9.5.3 [8.5.3], page 319:

```
package_renaming_declaration ::= package defining_program_unit_name renames <package_>name;
```

Section 9.5.4 [8.5.4], page 319:

```
subprogram_renaming_declaration ::=
    [overriding_indicator]
    subprogram_specification renames <callable_entity_>name;
```

Section 9.5.5 [8.5.5], page 323:

```
generic_renaming_declaration ::=
    generic package defining_program_unit_name renames <generic_package_>name;
```

```
    | generic procedure defining_program_unit_name renames <generic_procedure_>name;
```

```
    | generic function defining_program_unit_name renames <generic_function_>name;
```

Section 10.1 [9.1], page 329:

```
task_type_declaration ::=
    task_type defining_identifier [known_discriminant_part] [is
        [new_interface_list_with]
        task_definition];
```

Section 10.1 [9.1], page 329:

```
single_task_declaration ::=
    task defining_identifier [is
        [new_interface_list_with]
        task_definition];
```

Section 10.1 [9.1], page 329:

```
task_definition ::=
    {task_item}
    [ private
        {task_item}]
    end [<task_>identifier]
```

Section 10.1 [9.1], page 329:

```
task_item ::= entry_declaration | aspect_clause
```

Section 10.1 [9.1], page 329:

```
task_body ::=
```

```
task body defining_identifier is
  declarative_part
begin
  handled_sequence_of_statements
end [<task_>identifier];
```

Section 10.4 [9.4], page 337:

```
protected_type_declaration ::=
  protected type defining_identifier [known_discriminant_part] is
    [new interface_list with]
    protected_definition;
```

Section 10.4 [9.4], page 337:

```
single_protected_declaration ::=
  protected defining_identifier is
    [new interface_list with]
    protected_definition;
```

Section 10.4 [9.4], page 337:

```
protected_definition ::=
  { protected_operation_declaration }
[ private
  { protected_element_declaration } ]
end [<protected_>identifier]
```

Section 10.4 [9.4], page 337:

```
protected_operation_declaration ::= subprogram_declaration
  | entry_declaration
  | aspect_clause
```

Section 10.4 [9.4], page 337:

```
protected_element_declaration ::= protected_operation_declaration
  | component_declaration
```

Section 10.4 [9.4], page 337:

```
protected_body ::=
  protected body defining_identifier is
    { protected_operation_item }
  end [<protected_>identifier];
```

Section 10.4 [9.4], page 337:

```
protected_operation_item ::= subprogram_declaration
  | subprogram_body
  | entry_body
  | aspect_clause
```

Section 10.5.2 [9.5.2], page 347:

```
entry_declaration ::=
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile;
```

Section 10.5.2 [9.5.2], page 347:

```
accept_statement ::=
  accept <entry_>direct_name [(entry_index)] parameter_profile [do
```



```

        handled_sequence_of_statements
    end [<entry_>identifier];
Section 10.5.2 [9.5.2], page 347:
entry_index ::= expression
Section 10.5.2 [9.5.2], page 347:
entry_body ::=
    entry_defining_identifier entry_body_formal_part entry_barrier is
        declarative_part
    begin
        handled_sequence_of_statements
    end [<entry_>identifier];
Section 10.5.2 [9.5.2], page 347:
entry_body_formal_part ::= [(entry_index_specification)] parameter_profile
Section 10.5.2 [9.5.2], page 347:
entry_barrier ::= when condition
Section 10.5.2 [9.5.2], page 347:
entry_index_specification ::= for defining_identifier in discrete_subtype_definition
Section 10.5.3 [9.5.3], page 352:
entry_call_statement ::= <entry_>name [actual_parameter_part];
Section 10.5.4 [9.5.4], page 356:
requeue_statement ::= requeue <entry_>name [with abort];
Section 10.6 [9.6], page 358:
delay_statement ::= delay_until_statement | delay_relative_statement
Section 10.6 [9.6], page 358:
delay_until_statement ::= delay until <delay_>expression;
Section 10.6 [9.6], page 358:
delay_relative_statement ::= delay <delay_>expression;
Section 10.7 [9.7], page 377:
select_statement ::=
    selective_accept
    | timed_entry_call
    | conditional_entry_call
    | asynchronous_select
Section 10.7.1 [9.7.1], page 378:
selective_accept ::=
    select
        [guard]
        select_alternative
    { or
        [guard]
        select_alternative }
    [ else
        sequence_of_statements ]
    end select;

```


Section 10.7.1 [9.7.1], page 378:

```
guard ::= when condition =>
```

Section 10.7.1 [9.7.1], page 378:

```
select_alternative ::=  
    accept_alternative  
    | delay_alternative  
    | terminate_alternative
```

Section 10.7.1 [9.7.1], page 378:

```
accept_alternative ::=  
    accept_statement [sequence_of_statements]
```

Section 10.7.1 [9.7.1], page 378:

```
delay_alternative ::=  
    delay_statement [sequence_of_statements]
```

Section 10.7.1 [9.7.1], page 378:

```
terminate_alternative ::= terminate;
```

Section 10.7.2 [9.7.2], page 381:

```
timed_entry_call ::=  
    select  
        entry_call_alternative  
    or  
        delay_alternative  
    end select;
```

Section 10.7.2 [9.7.2], page 381:

```
entry_call_alternative ::=  
    procedure_or_entry_call [sequence_of_statements]
```

Section 10.7.2 [9.7.2], page 381:

```
procedure_or_entry_call ::=  
    procedure_call_statement | entry_call_statement
```

Section 10.7.3 [9.7.3], page 382:

```
conditional_entry_call ::=  
    select  
        entry_call_alternative  
    else  
        sequence_of_statements  
    end select;
```

Section 10.7.4 [9.7.4], page 383:

```
asynchronous_select ::=  
    select  
        triggering_alternative  
    then abort  
        abortable_part  
    end select;
```

Section 10.7.4 [9.7.4], page 383:

```
triggering_alternative ::= triggering_statement [sequence_of_statements]
```



Section 10.7.4 [9.7.4], page 383:
 triggering_statement ::= procedure_or_entry_call | delay_statement

Section 10.7.4 [9.7.4], page 383:
 abortable_part ::= sequence_of_statements

Section 10.8 [9.8], page 385:
 abort_statement ::= abort <task_>name {, <task_>name};

Section 11.1.1 [10.1.1], page 394:
 compilation ::= {compilation_unit}

Section 11.1.1 [10.1.1], page 394:
 compilation_unit ::=
 context_clause library_item
 | context_clause subunit

Section 11.1.1 [10.1.1], page 394:
 library_item ::= [private] library_unit_declaration
 | library_unit_body
 | [private] library_unit_renaming_declaration

Section 11.1.1 [10.1.1], page 394:
 library_unit_declaration ::=
 subprogram_declaration | package_declaration
 | generic_declaration | generic_instantiation

Section 11.1.1 [10.1.1], page 394:
 library_unit_renaming_declaration ::=
 package_renaming_declaration
 | generic_renaming_declaration
 | subprogram_renaming_declaration

Section 11.1.1 [10.1.1], page 394:
 library_unit_body ::= subprogram_body | package_body

Section 11.1.1 [10.1.1], page 394:
 parent_unit_name ::= name

Section 11.1.2 [10.1.2], page 399:
 context_clause ::= {context_item}

Section 11.1.2 [10.1.2], page 399:
 context_item ::= with_clause | use_clause

Section 11.1.2 [10.1.2], page 399:
 with_clause ::= limited_with_clause | nonlimited_with_clause

Section 11.1.2 [10.1.2], page 399:
 limited_with_clause ::= limited [private] with <library_unit_>name {, <library_unit_>name};

Section 11.1.2 [10.1.2], page 399:
 nonlimited_with_clause ::= [private] with <library_unit_>name {, <library_unit_>name};

Section 11.1.3 [10.1.3], page 403:
 body_stub ::= subprogram_body_stub | package_body_stub | task_body_stub | protected_body_stub

Section 11.1.3 [10.1.3], page 403:
 subprogram_body_stub ::=

[overriding_indicator]
 subprogram_specification is separate;

Section 11.1.3 [10.1.3], page 403:
 package_body_stub ::= package body defining_identifier is separate;

Section 11.1.3 [10.1.3], page 403:
 task_body_stub ::= task body defining_identifier is separate;

Section 11.1.3 [10.1.3], page 403:
 protected_body_stub ::= protected body defining_identifier is separate;

Section 11.1.3 [10.1.3], page 403:
 subunit ::= separate (parent_unit_name) proper_body

Section 12.1 [11.1], page 419:
 exception_declaration ::= defining_identifier_list : exception;

Section 12.2 [11.2], page 420:
 handled_sequence_of_statements ::=

 sequence_of_statements

 [exception

 exception_handler

 {exception_handler}]

Section 12.2 [11.2], page 420:
 exception_handler ::=

 when [choice_parameter_specification:] exception_choice { | exception_choice } =>

 sequence_of_statements

Section 12.2 [11.2], page 420:
 choice_parameter_specification ::= defining_identifier

Section 12.2 [11.2], page 420:
 exception_choice ::= <exception->name | others

Section 12.3 [11.3], page 421:
 raise_statement ::= raise;

 | raise <exception->name [with <string->expression];

Section 13.1 [12.1], page 450:
 generic_declaration ::= generic_subprogram_declaration | generic_package_declaration

Section 13.1 [12.1], page 450:
 generic_subprogram_declaration ::=

 generic_formal_part subprogram_specification;

Section 13.1 [12.1], page 450:
 generic_package_declaration ::=

 generic_formal_part package_specification;

Section 13.1 [12.1], page 450:
 generic_formal_part ::= generic {generic_formal_parameter_declaration | use_clause}

Section 13.1 [12.1], page 450:
 generic_formal_parameter_declaration ::=

 formal_object_declaration

- | formal_type_declaration
- | formal_subprogram_declaration
- | formal_package_declaration

Section 13.3 [12.3], page 454:

```
generic_instantiation ::=
    package defining_program_unit_name is
        new <generic_package_>name [generic_actual_part];
    | [overriding_indicator]
    procedure defining_program_unit_name is
        new <generic_procedure_>name [generic_actual_part];
    | [overriding_indicator]
    function defining_designator is
        new <generic_function_>name [generic_actual_part];
```

Section 13.3 [12.3], page 454:

```
generic_actual_part ::=
    (generic_association {, generic_association})
```

Section 13.3 [12.3], page 454:

```
generic_association ::=
    [<generic_formal_parameter_>selector_name =>] explicit_generic_actual_parameter
```

Section 13.3 [12.3], page 454:

```
explicit_generic_actual_parameter ::= expression | <variable_>name
```

- | <subprogram_>name | <entry_>name | subtype_mark
- | <package_instance_>name

Section 13.4 [12.4], page 458:

```
formal_object_declaration ::=
    defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression];

    defining_identifier_list : mode access_definition [:= default_expression];
```

Section 13.5 [12.5], page 460:

```
formal_type_declaration ::=
    type defining_identifier[discriminant_part] is formal_type_definition;
```

Section 13.5 [12.5], page 460:

```
formal_type_definition ::=
    formal_private_type_definition
    | formal_derived_type_definition
    | formal_discrete_type_definition
    | formal_signed_integer_type_definition
    | formal_modular_type_definition
    | formal_floating_point_definition
    | formal_ordinary_fixed_point_definition
    | formal_decimal_fixed_point_definition
    | formal_array_type_definition
    | formal_access_type_definition
    | formal_interface_type_definition
```

Section 13.5.1 [12.5.1], page 462:
 formal_private_type_definition ::= [[abstract] tagged] [limited] private ■

Section 13.5.1 [12.5.1], page 462:
 formal_derived_type_definition ::=
 [abstract] [limited | synchronized] new subtype_mark [[and interface_list]with private]

Section 13.5.2 [12.5.2], page 466:
 formal_discrete_type_definition ::= (<>)

Section 13.5.2 [12.5.2], page 466:
 formal_signed_integer_type_definition ::= range <>

Section 13.5.2 [12.5.2], page 466:
 formal_modular_type_definition ::= mod <>

Section 13.5.2 [12.5.2], page 466:
 formal_floating_point_definition ::= digits <>

Section 13.5.2 [12.5.2], page 466:
 formal_ordinary_fixed_point_definition ::= delta <>

Section 13.5.2 [12.5.2], page 466:
 formal_decimal_fixed_point_definition ::= delta <> digits <>

Section 13.5.3 [12.5.3], page 467:
 formal_array_type_definition ::= array_type_definition

Section 13.5.4 [12.5.4], page 468:
 formal_access_type_definition ::= access_type_definition

Section 13.5.5 [12.5.5], page 470:
 formal_interface_type_definition ::= interface_type_definition

Section 13.6 [12.6], page 470:
 formal_subprogram_declaration ::= formal_concrete_subprogram_declaration ■

 | formal_abstract_subprogram_declaration

Section 13.6 [12.6], page 470:
 formal_concrete_subprogram_declaration ::=
 with subprogram_specification [is subprogram_default];

Section 13.6 [12.6], page 470:
 formal_abstract_subprogram_declaration ::=
 with subprogram_specification is abstract [subprogram_default];

Section 13.6 [12.6], page 470:
 subprogram_default ::= default_name | <> | null

Section 13.6 [12.6], page 470:
 default_name ::= name

Section 13.7 [12.7], page 474:
 formal_package_declaration ::=
 with package_defining_identifier is new <generic_package_>name formal_package_actual_part;

Section 13.7 [12.7], page 474:
 formal_package_actual_part ::=
 ([others =>] <>)

| [generic_actual_part]
 | (formal_package_association {, formal_package_association} [, others => <>])

Section 13.7 [12.7], page 474:
 formal_package_association ::=
 generic_association
 | <generic_formal_parameter_selector_name => <>

Section 14.1 [13.1], page 481:
 aspect_clause ::= attribute_definition_clause
 | enumeration_representation_clause
 | record_representation_clause
 | at_clause

Section 14.1 [13.1], page 481:
 local_name ::= direct_name
 | direct_name'attribute_designator
 | <library_unit_name>

Section 14.3 [13.3], page 486:
 attribute_definition_clause ::=
 for local_name'attribute_designator use expression;
 | for local_name'attribute_designator use name;

Section 14.4 [13.4], page 500:
 enumeration_representation_clause ::=
 for <first_subtype_local_name> use enumeration_aggregate;

Section 14.4 [13.4], page 500:
 enumeration_aggregate ::= array_aggregate

Section 14.5.1 [13.5.1], page 501:
 record_representation_clause ::=
 for <first_subtype_local_name> use
 record [mod_clause]
 {component_clause}
 end record;

Section 14.5.1 [13.5.1], page 501:
 component_clause ::=
 <component_local_name> at position range first_bit .. last_bit;

Section 14.5.1 [13.5.1], page 501:
 position ::= <static_expression>

Section 14.5.1 [13.5.1], page 501:
 first_bit ::= <static_simple_expression>

Section 14.5.1 [13.5.1], page 501:
 last_bit ::= <static_simple_expression>

Section 14.8 [13.8], page 518:
 code_statement ::= qualified_expression;

Section 14.12 [13.12], page 535:
 restriction ::= <restriction_identifier>
 | <restriction_parameter_identifier => restriction_parameter_argument

Section 14.12 [13.12], page 535:
 restriction_parameter_argument ::= name | expression
 Section 23.3 [J.3], page 1167:
 delta_constraint ::= delta <static_>expression [range_constraint]
 Section 23.7 [J.7], page 1171:
 at_clause ::= for direct_name use at expression;
 Section 23.8 [J.8], page 1174:
 mod_clause ::= at mod <static_>expression;

Syntax Cross Reference

1

In the following syntax cross reference, each syntactic category is followed by the clause number where it is defined. In addition, each syntactic category <S> is followed by a list of the categories that use <S> in their definitions. For example, the first listing below shows that `abort_statement` appears in the definition of `simple_statement`.

`abort_statement` Section 10.8 [9.8], page 385,
`simple_statement` Section 6.1 [5.1], page 240,
`abortable_part` Section 10.7.4 [9.7.4], page 383,
`asynchronous_select` Section 10.7.4 [9.7.4], page 383,
`abstract_subprogram_declaration` Section 4.9.3 [3.9.3], page 149,
`basic_declaration` Section 4.1 [3.1], page 49,
`accept_alternative` Section 10.7.1 [9.7.1], page 378,
`select_alternative` Section 10.7.1 [9.7.1], page 378,
`accept_statement` Section 10.5.2 [9.5.2], page 347,
`accept_alternative` Section 10.7.1 [9.7.1], page 378,
`compound_statement` Section 6.1 [5.1], page 240,
`access_definition` Section 4.10 [3.10], page 156,
`component_definition` Section 4.6 [3.6], page 114,
`discriminant_specification` Section 4.7 [3.7], page 123,
`formal_object_declaration` Section 13.4 [12.4], page 458,
`object_declaration` Section 4.3.1 [3.3.1], page 61,
`object_renaming_declaration` Section 9.5.1 [8.5.1], page 317,
`parameter_and_result_profile` Section 7.1 [6.1], page 255,
`parameter_specification` Section 7.1 [6.1], page 255,
`return_subtype_indication` Section 7.5 [6.5], page 272,
`access_to_object_definition` Section 4.10 [3.10], page 156,
`access_type_definition` Section 4.10 [3.10], page 156,
`access_to_subprogram_definition` Section 4.10 [3.10], page 156,
`access_type_definition` Section 4.10 [3.10], page 156,
`access_type_definition` Section 4.10 [3.10], page 156,
`formal_access_type_definition` Section 13.5.4 [12.5.4], page 468,
`type_definition` Section 4.2.1 [3.2.1], page 53,
`actual_parameter_part` Section 7.4 [6.4], page 266,
`entry_call_statement` Section 10.5.3 [9.5.3], page 352,

function_call Section 7.4 [6.4], page 266,
procedure_call_statement Section 7.4 [6.4], page 266,
aggregate Section 5.3 [4.3], page 190,
primary Section 5.4 [4.4], page 201,
qualified_expression Section 5.7 [4.7], page 229,
allocator Section 5.8 [4.8], page 230,
primary Section 5.4 [4.4], page 201,
ancestor_part Section 5.3.2 [4.3.2], page 194,
extension_aggregate Section 5.3.2 [4.3.2], page 194,
array_aggregate Section 5.3.3 [4.3.3], page 196,
aggregate Section 5.3 [4.3], page 190,
enumeration_aggregate Section 14.4 [13.4], page 500,
array_component_association Section 5.3.3 [4.3.3], page 196,
named_array_aggregate Section 5.3.3 [4.3.3], page 196,
array_type_definition Section 4.6 [3.6], page 114,
formal_array_type_definition Section 13.5.3 [12.5.3], page 467,
object_declaration Section 4.3.1 [3.3.1], page 61,
type_definition Section 4.2.1 [3.2.1], page 53,
aspect_clause Section 14.1 [13.1], page 481,
basic_declarative_item Section 4.11 [3.11], page 175,
component_item Section 4.8 [3.8], page 130,
protected_operation_declaration Section 10.4 [9.4], page 337,
protected_operation_item Section 10.4 [9.4], page 337,
task_item Section 10.1 [9.1], page 329,
assignment_statement Section 6.2 [5.2], page 242,
simple_statement Section 6.1 [5.1], page 240,
asynchronous_select Section 10.7.4 [9.7.4], page 383,
select_statement Section 10.7 [9.7], page 377,
at_clause Section 23.7 [J.7], page 1171,
aspect_clause Section 14.1 [13.1], page 481,
attribute_definition_clause Section 14.3 [13.3], page 486,
aspect_clause Section 14.1 [13.1], page 481,
attribute_designator Section 5.1.4 [4.1.4], page 187,
attribute_definition_clause Section 14.3 [13.3], page 486,
attribute_reference Section 5.1.4 [4.1.4], page 187,
local_name Section 14.1 [13.1], page 481,
attribute_reference Section 5.1.4 [4.1.4], page 187,
name Section 5.1 [4.1], page 179,
base Section 3.4.2 [2.4.2], page 41,
based_literal Section 3.4.2 [2.4.2], page 41,
based_literal Section 3.4.2 [2.4.2], page 41,
numeric_literal Section 3.4 [2.4], page 39,
based_numeral Section 3.4.2 [2.4.2], page 41,
based_literal Section 3.4.2 [2.4.2], page 41,

basic_declaration Section 4.1 [3.1], page 49,
basic_declarative_item Section 4.11 [3.11], page 175,
basic_declarative_item Section 4.11 [3.11], page 175,
declarative_item Section 4.11 [3.11], page 175,
package_specification Section 8.1 [7.1], page 279,
binary_adding_operator Section 5.5 [4.5], page 203,
simple_expression Section 5.4 [4.4], page 201,
block_statement Section 6.6 [5.6], page 251,
compound_statement Section 6.1 [5.1], page 240,
body Section 4.11 [3.11], page 175,
declarative_item Section 4.11 [3.11], page 175,
body_stub Section 11.1.3 [10.1.3], page 403,
body Section 4.11 [3.11], page 175,
case_statement Section 6.4 [5.4], page 246,
compound_statement Section 6.1 [5.1], page 240,
case_statement_alternative Section 6.4 [5.4], page 246,
case_statement Section 6.4 [5.4], page 246,
character Section 3.1 [2.1], page 32,
comment Section 3.7 [2.7], page 43,
character_literal Section 3.5 [2.5], page 42,
defining_character_literal Section 4.5.1 [3.5.1], page 92,
name Section 5.1 [4.1], page 179,
selector_name Section 5.1.3 [4.1.3], page 183,
choice_parameter_specification Section 12.2 [11.2], page 420,
exception_handler Section 12.2 [11.2], page 420,
code_statement Section 14.8 [13.8], page 518,
simple_statement Section 6.1 [5.1], page 240,
compilation_unit Section 11.1.1 [10.1.1], page 394,
compilation Section 11.1.1 [10.1.1], page 394,
component_choice_list Section 5.3.1 [4.3.1], page 191,
record_component_association Section 5.3.1 [4.3.1], page 191,
component_clause Section 14.5.1 [13.5.1], page 501,
record_representation_clause Section 14.5.1 [13.5.1], page 501,
component_declaration Section 4.8 [3.8], page 130,
component_item Section 4.8 [3.8], page 130,
protected_element_declaration Section 10.4 [9.4], page 337,
component_definition Section 4.6 [3.6], page 114,
component_declaration Section 4.8 [3.8], page 130,
constrained_array_definition Section 4.6 [3.6], page 114,
unconstrained_array_definition Section 4.6 [3.6], page 114,
component_item Section 4.8 [3.8], page 130,
component_list Section 4.8 [3.8], page 130,

component_list Section 4.8 [3.8], page 130,
record_definition Section 4.8 [3.8], page 130,
variant Section 4.8.1 [3.8.1], page 134,
composite_constraint Section 4.2.2 [3.2.2], page 55,
constraint Section 4.2.2 [3.2.2], page 55,
compound_statement Section 6.1 [5.1], page 240,
statement Section 6.1 [5.1], page 240,
condition Section 6.3 [5.3], page 245,
entry_barrier Section 10.5.2 [9.5.2], page 347,
exit_statement Section 6.7 [5.7], page 252,
guard Section 10.7.1 [9.7.1], page 378,
if_statement Section 6.3 [5.3], page 245,
iteration_scheme Section 6.5 [5.5], page 248,
conditional_entry_call Section 10.7.3 [9.7.3], page 382,
select_statement Section 10.7 [9.7], page 377,
constrained_array_definition Section 4.6 [3.6], page 114,
array_type_definition Section 4.6 [3.6], page 114,
constraint Section 4.2.2 [3.2.2], page 55,
subtype_indication Section 4.2.2 [3.2.2], page 55,
context_clause Section 11.1.2 [10.1.2], page 399,
compilation_unit Section 11.1.1 [10.1.1], page 394,
context_item Section 11.1.2 [10.1.2], page 399,
context_clause Section 11.1.2 [10.1.2], page 399,
decimal_fixed_point_definition Section 4.5.9 [3.5.9], page 106,
fixed_point_definition Section 4.5.9 [3.5.9], page 106,
decimal_literal Section 3.4.1 [2.4.1], page 40,
numeric_literal Section 3.4 [2.4], page 39,
declarative_item Section 4.11 [3.11], page 175,
declarative_part Section 4.11 [3.11], page 175,
declarative_part Section 4.11 [3.11], page 175,
block_statement Section 6.6 [5.6], page 251,
entry_body Section 10.5.2 [9.5.2], page 347,
package_body Section 8.2 [7.2], page 281,
subprogram_body Section 7.3 [6.3], page 261,
task_body Section 10.1 [9.1], page 329,
default_expression Section 4.7 [3.7], page 123,
component_declaration Section 4.8 [3.8], page 130,
discriminant_specification Section 4.7 [3.7], page 123,
formal_object_declaration Section 13.4 [12.4], page 458,
parameter_specification Section 7.1 [6.1], page 255,
default_name Section 13.6 [12.6], page 470,
subprogram_default Section 13.6 [12.6], page 470,
defining_character_literal Section 4.5.1 [3.5.1], page 92,
enumeration_literal_specification Section 4.5.1 [3.5.1], page 92,

defining_designator Section 7.1 [6.1], page 255,
function_specification Section 7.1 [6.1], page 255,
generic_instantiation Section 13.3 [12.3], page 454,
defining_identifier Section 4.1 [3.1], page 49,
choice_parameter_specification Section 12.2 [11.2], page 420,
defining_identifier_list Section 4.3.1 [3.3.1], page 61,
defining_program_unit_name Section 7.1 [6.1], page 255,
entry_body Section 10.5.2 [9.5.2], page 347,
entry_declaration Section 10.5.2 [9.5.2], page 347,
entry_index_specification Section 10.5.2 [9.5.2], page 347,
enumeration_literal_specification Section 4.5.1 [3.5.1], page 92,
exception_renaming_declaration Section 9.5.2 [8.5.2], page 318,
extended_return_statement Section 7.5 [6.5], page 272,
formal_package_declaration Section 13.7 [12.7], page 474,
formal_type_declaration Section 13.5 [12.5], page 460,
full_type_declaration Section 4.2.1 [3.2.1], page 53,
incomplete_type_declaration Section 4.10.1 [3.10.1], page 160,
loop_parameter_specification Section 6.5 [5.5], page 248,
object_renaming_declaration Section 9.5.1 [8.5.1], page 317,
package_body_stub Section 11.1.3 [10.1.3], page 403,
private_extension_declaration Section 8.3 [7.3], page 283,
private_type_declaration Section 8.3 [7.3], page 283,
protected_body Section 10.4 [9.4], page 337,
protected_body_stub Section 11.1.3 [10.1.3], page 403,
protected_type_declaration Section 10.4 [9.4], page 337,
single_protected_declaration Section 10.4 [9.4], page 337,
single_task_declaration Section 10.1 [9.1], page 329,
subtype_declaration Section 4.2.2 [3.2.2], page 55,
task_body Section 10.1 [9.1], page 329,
task_body_stub Section 11.1.3 [10.1.3], page 403,
task_type_declaration Section 10.1 [9.1], page 329,
defining_identifier_list Section 4.3.1 [3.3.1], page 61,
component_declaration Section 4.8 [3.8], page 130,
discriminant_specification Section 4.7 [3.7], page 123,
exception_declaration Section 12.1 [11.1], page 419,
formal_object_declaration Section 13.4 [12.4], page 458,
number_declaration Section 4.3.2 [3.3.2], page 65,
object_declaration Section 4.3.1 [3.3.1], page 61,
parameter_specification Section 7.1 [6.1], page 255,
defining_operator_symbol Section 7.1 [6.1], page 255,
defining_designator Section 7.1 [6.1], page 255,
defining_program_unit_name Section 7.1 [6.1], page 255,
defining_designator Section 7.1 [6.1], page 255,
generic_instantiation Section 13.3 [12.3], page 454,
generic_renaming_declaration Section 9.5.5 [8.5.5], page 323,
package_body Section 8.2 [7.2], page 281,

package_renaming_declaration Section 9.5.3 [8.5.3], page 319,
package_specification Section 8.1 [7.1], page 279,
procedure_specification Section 7.1 [6.1], page 255,
delay_alternative Section 10.7.1 [9.7.1], page 378,
select_alternative Section 10.7.1 [9.7.1], page 378,
timed_entry_call Section 10.7.2 [9.7.2], page 381,
delay_relative_statement Section 10.6 [9.6], page 358,
delay_statement Section 10.6 [9.6], page 358,
delay_statement Section 10.6 [9.6], page 358,
delay_alternative Section 10.7.1 [9.7.1], page 378,
simple_statement Section 6.1 [5.1], page 240,
triggering_statement Section 10.7.4 [9.7.4], page 383,
delay_until_statement Section 10.6 [9.6], page 358,
delay_statement Section 10.6 [9.6], page 358,
delta_constraint Section 23.3 [J.3], page 1167,
scalar_constraint Section 4.2.2 [3.2.2], page 55,
derived_type_definition Section 4.4 [3.4], page 66,
type_definition Section 4.2.1 [3.2.1], page 53,
designator Section 7.1 [6.1], page 255,
subprogram_body Section 7.3 [6.3], page 261,
digit Section 3.4.1 [2.4.1], page 40,
extended_digit Section 3.4.2 [2.4.2], page 41,
numeral Section 3.4.1 [2.4.1], page 40,
digits_constraint Section 4.5.9 [3.5.9], page 106,
scalar_constraint Section 4.2.2 [3.2.2], page 55,
direct_name Section 5.1 [4.1], page 179,
accept_statement Section 10.5.2 [9.5.2], page 347,
at_clause Section 23.7 [J.7], page 1171,
local_name Section 14.1 [13.1], page 481,
name Section 5.1 [4.1], page 179,
statement_identifier Section 6.1 [5.1], page 240,
variant_part Section 4.8.1 [3.8.1], page 134,
discrete_choice Section 4.8.1 [3.8.1], page 134,
discrete_choice_list Section 4.8.1 [3.8.1], page 134,
discrete_choice_list Section 4.8.1 [3.8.1], page 134,
array_component_association Section 5.3.3 [4.3.3], page 196,
case_statement_alternative Section 6.4 [5.4], page 246,
variant Section 4.8.1 [3.8.1], page 134,
discrete_range Section 4.6.1 [3.6.1], page 117,
discrete_choice Section 4.8.1 [3.8.1], page 134,
index_constraint Section 4.6.1 [3.6.1], page 117,
slice Section 5.1.2 [4.1.2], page 182,
discrete_subtype_definition Section 4.6 [3.6], page 114,
constrained_array_definition Section 4.6 [3.6], page 114,

entry_declaration Section 10.5.2 [9.5.2], page 347,
entry_index_specification Section 10.5.2 [9.5.2], page 347,
loop_parameter_specification Section 6.5 [5.5], page 248,
discriminant_association Section 4.7.1 [3.7.1], page 127,
discriminant_constraint Section 4.7.1 [3.7.1], page 127,
discriminant_constraint Section 4.7.1 [3.7.1], page 127,
composite_constraint Section 4.2.2 [3.2.2], page 55,
discriminant_part Section 4.7 [3.7], page 123,
formal_type_declaration Section 13.5 [12.5], page 460,
incomplete_type_declaration Section 4.10.1 [3.10.1], page 160,
private_extension_declaration Section 8.3 [7.3], page 283,
private_type_declaration Section 8.3 [7.3], page 283,
discriminant_specification Section 4.7 [3.7], page 123,
known_discriminant_part Section 4.7 [3.7], page 123,
entry_barrier Section 10.5.2 [9.5.2], page 347,
entry_body Section 10.5.2 [9.5.2], page 347,
entry_body Section 10.5.2 [9.5.2], page 347,
protected_operation_item Section 10.4 [9.4], page 337,
entry_body_formal_part Section 10.5.2 [9.5.2], page 347,
entry_body Section 10.5.2 [9.5.2], page 347,
entry_call_alternative Section 10.7.2 [9.7.2], page 381,
conditional_entry_call Section 10.7.3 [9.7.3], page 382,
timed_entry_call Section 10.7.2 [9.7.2], page 381,
entry_call_statement Section 10.5.3 [9.5.3], page 352,
procedure_or_entry_call Section 10.7.2 [9.7.2], page 381,
simple_statement Section 6.1 [5.1], page 240,
entry_declaration Section 10.5.2 [9.5.2], page 347,
protected_operation_declaration Section 10.4 [9.4], page 337,
task_item Section 10.1 [9.1], page 329,
entry_index Section 10.5.2 [9.5.2], page 347,
accept_statement Section 10.5.2 [9.5.2], page 347,
entry_index_specification Section 10.5.2 [9.5.2], page 347,
entry_body_formal_part Section 10.5.2 [9.5.2], page 347,
enumeration_aggregate Section 14.4 [13.4], page 500,
enumeration_representation_clause Section 14.4 [13.4], page 500,
enumeration_literal_specification Section 4.5.1 [3.5.1], page 92,
enumeration_type_definition Section 4.5.1 [3.5.1], page 92,
enumeration_representation_clause Section 14.4 [13.4], page 500,
aspect_clause Section 14.1 [13.1], page 481,
enumeration_type_definition Section 4.5.1 [3.5.1], page 92,
type_definition Section 4.2.1 [3.2.1], page 53,
exception_choice Section 12.2 [11.2], page 420,
exception_handler Section 12.2 [11.2], page 420,

exception_declaration Section 12.1 [11.1], page 419,
basic_declaration Section 4.1 [3.1], page 49,
exception_handler Section 12.2 [11.2], page 420,
handled_sequence_of_statements Section 12.2 [11.2], page 420,
exception_renaming_declaration Section 9.5.2 [8.5.2], page 318,
renaming_declaration Section 9.5 [8.5], page 316,
exit_statement Section 6.7 [5.7], page 252,
simple_statement Section 6.1 [5.1], page 240,
explicit_actual_parameter Section 7.4 [6.4], page 266,
parameter_association Section 7.4 [6.4], page 266,
explicit_dereference Section 5.1 [4.1], page 179,
name Section 5.1 [4.1], page 179,
explicit_generic_actual_parameter Section 13.3 [12.3], page 454,
generic_association Section 13.3 [12.3], page 454,
exponent Section 3.4.1 [2.4.1], page 40,
based_literal Section 3.4.2 [2.4.2], page 41,
decimal_literal Section 3.4.1 [2.4.1], page 40,
expression Section 5.4 [4.4], page 201,
ancestor_part Section 5.3.2 [4.3.2], page 194,
array_component_association Section 5.3.3 [4.3.3], page 196,
assignment_statement Section 6.2 [5.2], page 242,
at_clause Section 23.7 [J.7], page 1171,
attribute_definition_clause Section 14.3 [13.3], page 486,
attribute_designator Section 5.1.4 [4.1.4], page 187,
case_statement Section 6.4 [5.4], page 246,
condition Section 6.3 [5.3], page 245,
decimal_fixed_point_definition Section 4.5.9 [3.5.9], page 106,
default_expression Section 4.7 [3.7], page 123,
delay_relative_statement Section 10.6 [9.6], page 358,
delay_until_statement Section 10.6 [9.6], page 358,
delta_constraint Section 23.3 [J.3], page 1167,
digits_constraint Section 4.5.9 [3.5.9], page 106,
discrete_choice Section 4.8.1 [3.8.1], page 134,
discriminant_association Section 4.7.1 [3.7.1], page 127,
entry_index Section 10.5.2 [9.5.2], page 347,
explicit_actual_parameter Section 7.4 [6.4], page 266,
explicit_generic_actual_parameter Section 13.3 [12.3], page 454,
extended_return_statement Section 7.5 [6.5], page 272,
floating_point_definition Section 4.5.7 [3.5.7], page 103,
indexed_component Section 5.1.1 [4.1.1], page 181,
mod_clause Section 23.8 [J.8], page 1174,
modular_type_definition Section 4.5.4 [3.5.4], page 95,
number_declaration Section 4.3.2 [3.3.2], page 65,
object_declaration Section 4.3.1 [3.3.1], page 61,
ordinary_fixed_point_definition Section 4.5.9 [3.5.9], page 106,

position Section 14.5.1 [13.5.1], page 501,
positional_array_aggregate Section 5.3.3 [4.3.3], page 196,
pragma_argument_association Section 3.8 [2.8], page 44,
primary Section 5.4 [4.4], page 201,
qualified_expression Section 5.7 [4.7], page 229,
raise_statement Section 12.3 [11.3], page 421,
range_attribute_designator Section 5.1.4 [4.1.4], page 187,
record_component_association Section 5.3.1 [4.3.1], page 191,
restriction_parameter_argument Section 14.12 [13.12], page 535,
simple_return_statement Section 7.5 [6.5], page 272,
type_conversion Section 5.6 [4.6], page 219,
extended_digit Section 3.4.2 [2.4.2], page 41,
based_numeral Section 3.4.2 [2.4.2], page 41,
extended_return_statement Section 7.5 [6.5], page 272,
compound_statement Section 6.1 [5.1], page 240,
extension_aggregate Section 5.3.2 [4.3.2], page 194,
aggregate Section 5.3 [4.3], page 190,
factor Section 5.4 [4.4], page 201,
term Section 5.4 [4.4], page 201,
first_bit Section 14.5.1 [13.5.1], page 501,
component_clause Section 14.5.1 [13.5.1], page 501,
fixed_point_definition Section 4.5.9 [3.5.9], page 106,
real_type_definition Section 4.5.6 [3.5.6], page 102,
floating_point_definition Section 4.5.7 [3.5.7], page 103,
real_type_definition Section 4.5.6 [3.5.6], page 102,
formal_abstract_subprogram_declaration Section 13.6 [12.6],
page 470,
formal_subprogram_declaration Section 13.6 [12.6], page 470,
formal_access_type_definition Section 13.5.4 [12.5.4], page 468,
formal_type_definition Section 13.5 [12.5], page 460,
formal_array_type_definition Section 13.5.3 [12.5.3], page 467,
formal_type_definition Section 13.5 [12.5], page 460,
formal_concrete_subprogram_declaration Section 13.6 [12.6],
page 470,
formal_subprogram_declaration Section 13.6 [12.6], page 470,
formal_decimal_fixed_point_definition Section 13.5.2 [12.5.2],
page 466,
formal_type_definition Section 13.5 [12.5], page 460,
formal_derived_type_definition Section 13.5.1 [12.5.1], page 462,
formal_type_definition Section 13.5 [12.5], page 460,
formal_discrete_type_definition Section 13.5.2 [12.5.2], page 466,
formal_type_definition Section 13.5 [12.5], page 460,
formal_floating_point_definition Section 13.5.2 [12.5.2], page 466,
formal_type_definition Section 13.5 [12.5], page 460,

formal_interface_type_definition Section 13.5.5 [12.5.5], page 470,
formal_type_definition Section 13.5 [12.5], page 460,
formal_modular_type_definition Section 13.5.2 [12.5.2], page 466,
formal_type_definition Section 13.5 [12.5], page 460,
formal_object_declaration Section 13.4 [12.4], page 458,
generic_formal_parameter_declaration Section 13.1 [12.1], page 450,
formal_ordinary_fixed_point_definition Section 13.5.2 [12.5.2],
page 466,
formal_type_definition Section 13.5 [12.5], page 460,
formal_package_actual_part Section 13.7 [12.7], page 474,
formal_package_declaration Section 13.7 [12.7], page 474,
formal_package_association Section 13.7 [12.7], page 474,
formal_package_actual_part Section 13.7 [12.7], page 474,
formal_package_declaration Section 13.7 [12.7], page 474,
generic_formal_parameter_declaration Section 13.1 [12.1], page 450,
formal_part Section 7.1 [6.1], page 255,
parameter_and_result_profile Section 7.1 [6.1], page 255,
parameter_profile Section 7.1 [6.1], page 255,
formal_private_type_definition Section 13.5.1 [12.5.1], page 462,
formal_type_definition Section 13.5 [12.5], page 460,
formal_signed_integer_type_definition Section 13.5.2 [12.5.2],
page 466,
formal_type_definition Section 13.5 [12.5], page 460,
formal_subprogram_declaration Section 13.6 [12.6], page 470,
generic_formal_parameter_declaration Section 13.1 [12.1], page 450,
formal_type_declaration Section 13.5 [12.5], page 460,
generic_formal_parameter_declaration Section 13.1 [12.1], page 450,
formal_type_definition Section 13.5 [12.5], page 460,
formal_type_declaration Section 13.5 [12.5], page 460,
full_type_declaration Section 4.2.1 [3.2.1], page 53,
type_declaration Section 4.2.1 [3.2.1], page 53,
function_call Section 7.4 [6.4], page 266,
name Section 5.1 [4.1], page 179,
function_specification Section 7.1 [6.1], page 255,
subprogram_specification Section 7.1 [6.1], page 255,
general_access_modifier Section 4.10 [3.10], page 156,
access_to_object_definition Section 4.10 [3.10], page 156,
generic_actual_part Section 13.3 [12.3], page 454,
formal_package_actual_part Section 13.7 [12.7], page 474,
generic_instantiation Section 13.3 [12.3], page 454,
generic_association Section 13.3 [12.3], page 454,
formal_package_association Section 13.7 [12.7], page 474,
generic_actual_part Section 13.3 [12.3], page 454,

generic_declaration Section 13.1 [12.1], page 450,
basic_declaration Section 4.1 [3.1], page 49,
library_unit_declaration Section 11.1.1 [10.1.1], page 394,
generic_formal_parameter_declaration Section 13.1 [12.1], page 450,
generic_formal_part Section 13.1 [12.1], page 450,
generic_formal_part Section 13.1 [12.1], page 450,
generic_package_declaration Section 13.1 [12.1], page 450,
generic_subprogram_declaration Section 13.1 [12.1], page 450,
generic_instantiation Section 13.3 [12.3], page 454,
basic_declaration Section 4.1 [3.1], page 49,
library_unit_declaration Section 11.1.1 [10.1.1], page 394,
generic_package_declaration Section 13.1 [12.1], page 450,
generic_declaration Section 13.1 [12.1], page 450,
generic_renaming_declaration Section 9.5.5 [8.5.5], page 323,
library_unit_renaming_declaration Section 11.1.1 [10.1.1], page 394,
renaming_declaration Section 9.5 [8.5], page 316,
generic_subprogram_declaration Section 13.1 [12.1], page 450,
generic_declaration Section 13.1 [12.1], page 450,
goto_statement Section 6.8 [5.8], page 253,
simple_statement Section 6.1 [5.1], page 240,
graphic_character Section 3.1 [2.1], page 32,
character_literal Section 3.5 [2.5], page 42,
string_element Section 3.6 [2.6], page 42,
guard Section 10.7.1 [9.7.1], page 378,
selective_accept Section 10.7.1 [9.7.1], page 378,
handled_sequence_of_statements Section 12.2 [11.2], page 420,
accept_statement Section 10.5.2 [9.5.2], page 347,
block_statement Section 6.6 [5.6], page 251,
entry_body Section 10.5.2 [9.5.2], page 347,
extended_return_statement Section 7.5 [6.5], page 272,
package_body Section 8.2 [7.2], page 281,
subprogram_body Section 7.3 [6.3], page 261,
task_body Section 10.1 [9.1], page 329,
identifier Section 3.3 [2.3], page 38,
accept_statement Section 10.5.2 [9.5.2], page 347,
attribute_designator Section 5.1.4 [4.1.4], page 187,
block_statement Section 6.6 [5.6], page 251,
defining_identifier Section 4.1 [3.1], page 49,
designator Section 7.1 [6.1], page 255,
direct_name Section 5.1 [4.1], page 179,
entry_body Section 10.5.2 [9.5.2], page 347,
loop_statement Section 6.5 [5.5], page 248,
package_body Section 8.2 [7.2], page 281,
package_specification Section 8.1 [7.1], page 279,

pragma Section 3.8 [2.8], page 44,
pragma_argument_association Section 3.8 [2.8], page 44,
protected_body Section 10.4 [9.4], page 337,
protected_definition Section 10.4 [9.4], page 337,
restriction Section 14.12 [13.12], page 535,
selector_name Section 5.1.3 [4.1.3], page 183,
task_body Section 10.1 [9.1], page 329,
task_definition Section 10.1 [9.1], page 329,
identifier_extend Section 3.3 [2.3], page 38,
identifier Section 3.3 [2.3], page 38,
identifier_start Section 3.3 [2.3], page 38,
identifier Section 3.3 [2.3], page 38,
if_statement Section 6.3 [5.3], page 245,
compound_statement Section 6.1 [5.1], page 240,
implicit_dereference Section 5.1 [4.1], page 179,
prefix Section 5.1 [4.1], page 179,
incomplete_type_declaration Section 4.10.1 [3.10.1], page 160,
type_declaration Section 4.2.1 [3.2.1], page 53,
index_constraint Section 4.6.1 [3.6.1], page 117,
composite_constraint Section 4.2.2 [3.2.2], page 55,
index_subtype_definition Section 4.6 [3.6], page 114,
unconstrained_array_definition Section 4.6 [3.6], page 114,
indexed_component Section 5.1.1 [4.1.1], page 181,
name Section 5.1 [4.1], page 179,
integer_type_definition Section 4.5.4 [3.5.4], page 95,
type_definition Section 4.2.1 [3.2.1], page 53,
interface_list Section 4.9.4 [3.9.4], page 152,
derived_type_definition Section 4.4 [3.4], page 66,
formal_derived_type_definition Section 13.5.1 [12.5.1], page 462,
interface_type_definition Section 4.9.4 [3.9.4], page 152,
private_extension_declaration Section 8.3 [7.3], page 283,
protected_type_declaration Section 10.4 [9.4], page 337,
single_protected_declaration Section 10.4 [9.4], page 337,
single_task_declaration Section 10.1 [9.1], page 329,
task_type_declaration Section 10.1 [9.1], page 329,
interface_type_definition Section 4.9.4 [3.9.4], page 152,
formal_interface_type_definition Section 13.5.5 [12.5.5], page 470,
type_definition Section 4.2.1 [3.2.1], page 53,
iteration_scheme Section 6.5 [5.5], page 248,
loop_statement Section 6.5 [5.5], page 248,
known_discriminant_part Section 4.7 [3.7], page 123,
discriminant_part Section 4.7 [3.7], page 123,
full_type_declaration Section 4.2.1 [3.2.1], page 53,

protected_type_declaration Section 10.4 [9.4], page 337,
task_type_declaration Section 10.1 [9.1], page 329,
label Section 6.1 [5.1], page 240,
statement Section 6.1 [5.1], page 240,
last_bit Section 14.5.1 [13.5.1], page 501,
component_clause Section 14.5.1 [13.5.1], page 501,
letter_lowercase ...
 identifier_start Section 3.3 [2.3], page 38,
letter_modifier ...
 identifier_start Section 3.3 [2.3], page 38,
letter_other ...
 identifier_start Section 3.3 [2.3], page 38,
letter_titlecase ...
 identifier_start Section 3.3 [2.3], page 38,
letter_uppercase ...
 identifier_start Section 3.3 [2.3], page 38,
library_item Section 11.1.1 [10.1.1], page 394,
 compilation_unit Section 11.1.1 [10.1.1], page 394,
library_unit_body Section 11.1.1 [10.1.1], page 394,
 library_item Section 11.1.1 [10.1.1], page 394,
library_unit_declaration Section 11.1.1 [10.1.1], page 394,
 library_item Section 11.1.1 [10.1.1], page 394,
library_unit_renaming_declaration Section 11.1.1 [10.1.1], page 394,
 library_item Section 11.1.1 [10.1.1], page 394,
limited_with_clause Section 11.1.2 [10.1.2], page 399,
 with_clause Section 11.1.2 [10.1.2], page 399,
local_name Section 14.1 [13.1], page 481,
 attribute_definition_clause Section 14.3 [13.3], page 486,
 component_clause Section 14.5.1 [13.5.1], page 501,
 enumeration_representation_clause Section 14.4 [13.4], page 500,
 record_representation_clause Section 14.5.1 [13.5.1], page 501,
loop_parameter_specification Section 6.5 [5.5], page 248,
 iteration_scheme Section 6.5 [5.5], page 248,
loop_statement Section 6.5 [5.5], page 248,
 compound_statement Section 6.1 [5.1], page 240,
mark_non_spacing ...
 identifier_extend Section 3.3 [2.3], page 38,
mark_spacing_combining ...
 identifier_extend Section 3.3 [2.3], page 38,
mod_clause Section 23.8 [J.8], page 1174,
 record_representation_clause Section 14.5.1 [13.5.1], page 501,

mode Section 7.1 [6.1], page 255,
formal_object_declaration Section 13.4 [12.4], page 458,
parameter_specification Section 7.1 [6.1], page 255,
modular_type_definition Section 4.5.4 [3.5.4], page 95,
integer_type_definition Section 4.5.4 [3.5.4], page 95,
multiplying_operator Section 5.5 [4.5], page 203,
term Section 5.4 [4.4], page 201,
name Section 5.1 [4.1], page 179,
abort_statement Section 10.8 [9.8], page 385,
assignment_statement Section 6.2 [5.2], page 242,
attribute_definition_clause Section 14.3 [13.3], page 486,
default_name Section 13.6 [12.6], page 470,
entry_call_statement Section 10.5.3 [9.5.3], page 352,
exception_choice Section 12.2 [11.2], page 420,
exception_renaming_declaration Section 9.5.2 [8.5.2], page 318,
exit_statement Section 6.7 [5.7], page 252,
explicit_actual_parameter Section 7.4 [6.4], page 266,
explicit_dereference Section 5.1 [4.1], page 179,
explicit_generic_actual_parameter Section 13.3 [12.3], page 454,
formal_package_declaration Section 13.7 [12.7], page 474,
function_call Section 7.4 [6.4], page 266,
generic_instantiation Section 13.3 [12.3], page 454,
generic_renaming_declaration Section 9.5.5 [8.5.5], page 323,
goto_statement Section 6.8 [5.8], page 253,
implicit_dereference Section 5.1 [4.1], page 179,
limited_with_clause Section 11.1.2 [10.1.2], page 399,
local_name Section 14.1 [13.1], page 481,
nonlimited_with_clause Section 11.1.2 [10.1.2], page 399,
object_renaming_declaration Section 9.5.1 [8.5.1], page 317,
package_renaming_declaration Section 9.5.3 [8.5.3], page 319,
parent_unit_name Section 11.1.1 [10.1.1], page 394,
pragma_argument_association Section 3.8 [2.8], page 44,
prefix Section 5.1 [4.1], page 179,
primary Section 5.4 [4.4], page 201,
procedure_call_statement Section 7.4 [6.4], page 266,
raise_statement Section 12.3 [11.3], page 421,
requeue_statement Section 10.5.4 [9.5.4], page 356,
restriction_parameter_argument Section 14.12 [13.12], page 535,
subprogram_renaming_declaration Section 9.5.4 [8.5.4], page 319,
subtype_mark Section 4.2.2 [3.2.2], page 55,
type_conversion Section 5.6 [4.6], page 219,
use_package_clause Section 9.4 [8.4], page 314,
named_array_aggregate Section 5.3.3 [4.3.3], page 196,
array_aggregate Section 5.3.3 [4.3.3], page 196,
nonlimited_with_clause Section 11.1.2 [10.1.2], page 399,
with_clause Section 11.1.2 [10.1.2], page 399,

null_exclusion Section 4.10 [3.10], page 156,
access_definition Section 4.10 [3.10], page 156,
access_type_definition Section 4.10 [3.10], page 156,
discriminant_specification Section 4.7 [3.7], page 123,
formal_object_declaration Section 13.4 [12.4], page 458,
object_renaming_declaration Section 9.5.1 [8.5.1], page 317,
parameter_and_result_profile Section 7.1 [6.1], page 255,
parameter_specification Section 7.1 [6.1], page 255,
subtype_indication Section 4.2.2 [3.2.2], page 55,
null_procedure_declaration Section 7.7 [6.7], page 277,
basic_declaration Section 4.1 [3.1], page 49,
null_statement Section 6.1 [5.1], page 240,
simple_statement Section 6.1 [5.1], page 240,
number_decimal ...
 identifier_extend Section 3.3 [2.3], page 38,
number_declaration Section 4.3.2 [3.3.2], page 65,
 basic_declaration Section 4.1 [3.1], page 49,
number_letter ...
 identifier_start Section 3.3 [2.3], page 38,
numeral Section 3.4.1 [2.4.1], page 40,
 base Section 3.4.2 [2.4.2], page 41,
 decimal_literal Section 3.4.1 [2.4.1], page 40,
 exponent Section 3.4.1 [2.4.1], page 40,
numeric_literal Section 3.4 [2.4], page 39,
 primary Section 5.4 [4.4], page 201,
object_declaration Section 4.3.1 [3.3.1], page 61,
 basic_declaration Section 4.1 [3.1], page 49,
object_renaming_declaration Section 9.5.1 [8.5.1], page 317,
 renaming_declaration Section 9.5 [8.5], page 316,
operator_symbol Section 7.1 [6.1], page 255,
 defining_operator_symbol Section 7.1 [6.1], page 255,
 designator Section 7.1 [6.1], page 255,
 direct_name Section 5.1 [4.1], page 179,
 selector_name Section 5.1.3 [4.1.3], page 183,
ordinary_fixed_point_definition Section 4.5.9 [3.5.9], page 106,
 fixed_point_definition Section 4.5.9 [3.5.9], page 106,
other_format ...
 identifier_extend Section 3.3 [2.3], page 38,
overriding_indicator Section 9.3.1 [8.3.1], page 312,
 abstract_subprogram_declaration Section 4.9.3 [3.9.3], page 149,
 entry_declaration Section 10.5.2 [9.5.2], page 347,
 generic_instantiation Section 13.3 [12.3], page 454,
 null_procedure_declaration Section 7.7 [6.7], page 277,
 subprogram_body Section 7.3 [6.3], page 261,

subprogram_body_stub Section 11.1.3 [10.1.3], page 403,
subprogram_declaration Section 7.1 [6.1], page 255,
subprogram_renaming_declaration Section 9.5.4 [8.5.4], page 319,
package_body Section 8.2 [7.2], page 281,
library_unit_body Section 11.1.1 [10.1.1], page 394,
proper_body Section 4.11 [3.11], page 175,
package_body_stub Section 11.1.3 [10.1.3], page 403,
body_stub Section 11.1.3 [10.1.3], page 403,
package_declaration Section 8.1 [7.1], page 279,
basic_declaration Section 4.1 [3.1], page 49,
library_unit_declaration Section 11.1.1 [10.1.1], page 394,
package_renaming_declaration Section 9.5.3 [8.5.3], page 319,
library_unit_renaming_declaration Section 11.1.1 [10.1.1], page 394,
renaming_declaration Section 9.5 [8.5], page 316,
package_specification Section 8.1 [7.1], page 279,
generic_package_declaration Section 13.1 [12.1], page 450,
package_declaration Section 8.1 [7.1], page 279,
parameter_and_result_profile Section 7.1 [6.1], page 255,
access_definition Section 4.10 [3.10], page 156,
access_to_subprogram_definition Section 4.10 [3.10], page 156,
function_specification Section 7.1 [6.1], page 255,
parameter_association Section 7.4 [6.4], page 266,
actual_parameter_part Section 7.4 [6.4], page 266,
parameter_profile Section 7.1 [6.1], page 255,
accept_statement Section 10.5.2 [9.5.2], page 347,
access_definition Section 4.10 [3.10], page 156,
access_to_subprogram_definition Section 4.10 [3.10], page 156,
entry_body_formal_part Section 10.5.2 [9.5.2], page 347,
entry_declaration Section 10.5.2 [9.5.2], page 347,
procedure_specification Section 7.1 [6.1], page 255,
parameter_specification Section 7.1 [6.1], page 255,
formal_part Section 7.1 [6.1], page 255,
parent_unit_name Section 11.1.1 [10.1.1], page 394,
defining_program_unit_name Section 7.1 [6.1], page 255,
designator Section 7.1 [6.1], page 255,
package_body Section 8.2 [7.2], page 281,
package_specification Section 8.1 [7.1], page 279,
subunit Section 11.1.3 [10.1.3], page 403,
position Section 14.5.1 [13.5.1], page 501,
component_clause Section 14.5.1 [13.5.1], page 501,
positional_array_aggregate Section 5.3.3 [4.3.3], page 196,
array_aggregate Section 5.3.3 [4.3.3], page 196,
pragma_argument_association Section 3.8 [2.8], page 44,
pragma Section 3.8 [2.8], page 44,

prefix Section 5.1 [4.1], page 179,
attribute_reference Section 5.1.4 [4.1.4], page 187,
function_call Section 7.4 [6.4], page 266,
indexed_component Section 5.1.1 [4.1.1], page 181,
procedure_call_statement Section 7.4 [6.4], page 266,
range_attribute_reference Section 5.1.4 [4.1.4], page 187,
selected_component Section 5.1.3 [4.1.3], page 183,
slice Section 5.1.2 [4.1.2], page 182,
primary Section 5.4 [4.4], page 201,
factor Section 5.4 [4.4], page 201,
private_extension_declaration Section 8.3 [7.3], page 283,
type_declaration Section 4.2.1 [3.2.1], page 53,
private_type_declaration Section 8.3 [7.3], page 283,
type_declaration Section 4.2.1 [3.2.1], page 53,
procedure_call_statement Section 7.4 [6.4], page 266,
procedure_or_entry_call Section 10.7.2 [9.7.2], page 381,
simple_statement Section 6.1 [5.1], page 240,
procedure_or_entry_call Section 10.7.2 [9.7.2], page 381,
entry_call_alternative Section 10.7.2 [9.7.2], page 381,
triggering_statement Section 10.7.4 [9.7.4], page 383,
procedure_specification Section 7.1 [6.1], page 255,
null_procedure_declaration Section 7.7 [6.7], page 277,
subprogram_specification Section 7.1 [6.1], page 255,
proper_body Section 4.11 [3.11], page 175,
body Section 4.11 [3.11], page 175,
subunit Section 11.1.3 [10.1.3], page 403,
protected_body Section 10.4 [9.4], page 337,
proper_body Section 4.11 [3.11], page 175,
protected_body_stub Section 11.1.3 [10.1.3], page 403,
body_stub Section 11.1.3 [10.1.3], page 403,
protected_definition Section 10.4 [9.4], page 337,
protected_type_declaration Section 10.4 [9.4], page 337,
single_protected_declaration Section 10.4 [9.4], page 337,
protected_element_declaration Section 10.4 [9.4], page 337,
protected_definition Section 10.4 [9.4], page 337,
protected_operation_declaration Section 10.4 [9.4], page 337,
protected_definition Section 10.4 [9.4], page 337,
protected_element_declaration Section 10.4 [9.4], page 337,
protected_operation_item Section 10.4 [9.4], page 337,
protected_body Section 10.4 [9.4], page 337,
protected_type_declaration Section 10.4 [9.4], page 337,
full_type_declaration Section 4.2.1 [3.2.1], page 53,
punctuation_connector ...
identifier_extend Section 3.3 [2.3], page 38,

qualified_expression Section 5.7 [4.7], page 229,
allocator Section 5.8 [4.8], page 230,
code_statement Section 14.8 [13.8], page 518,
primary Section 5.4 [4.4], page 201,
raise_statement Section 12.3 [11.3], page 421,
simple_statement Section 6.1 [5.1], page 240,
range Section 4.5 [3.5], page 76,
discrete_range Section 4.6.1 [3.6.1], page 117,
discrete_subtype_definition Section 4.6 [3.6], page 114,
range_constraint Section 4.5 [3.5], page 76,
relation Section 5.4 [4.4], page 201,
range_attribute_designator Section 5.1.4 [4.1.4], page 187,
range_attribute_reference Section 5.1.4 [4.1.4], page 187,
range_attribute_reference Section 5.1.4 [4.1.4], page 187,
range Section 4.5 [3.5], page 76,
range_constraint Section 4.5 [3.5], page 76,
delta_constraint Section 23.3 [J.3], page 1167,
digits_constraint Section 4.5.9 [3.5.9], page 106,
scalar_constraint Section 4.2.2 [3.2.2], page 55,
real_range_specification Section 4.5.7 [3.5.7], page 103,
decimal_fixed_point_definition Section 4.5.9 [3.5.9], page 106,
floating_point_definition Section 4.5.7 [3.5.7], page 103,
ordinary_fixed_point_definition Section 4.5.9 [3.5.9], page 106,
real_type_definition Section 4.5.6 [3.5.6], page 102,
type_definition Section 4.2.1 [3.2.1], page 53,
record_aggregate Section 5.3.1 [4.3.1], page 191,
aggregate Section 5.3 [4.3], page 190,
record_component_association Section 5.3.1 [4.3.1], page 191,
record_component_association_list Section 5.3.1 [4.3.1], page 191,
record_component_association_list Section 5.3.1 [4.3.1], page 191,
extension_aggregate Section 5.3.2 [4.3.2], page 194,
record_aggregate Section 5.3.1 [4.3.1], page 191,
record_definition Section 4.8 [3.8], page 130,
record_extension_part Section 4.9.1 [3.9.1], page 143,
record_type_definition Section 4.8 [3.8], page 130,
record_extension_part Section 4.9.1 [3.9.1], page 143,
derived_type_definition Section 4.4 [3.4], page 66,
record_representation_clause Section 14.5.1 [13.5.1], page 501,
aspect_clause Section 14.1 [13.1], page 481,
record_type_definition Section 4.8 [3.8], page 130,
type_definition Section 4.2.1 [3.2.1], page 53,
relation Section 5.4 [4.4], page 201,
expression Section 5.4 [4.4], page 201,

relational_operator Section 5.5 [4.5], page 203,
relation Section 5.4 [4.4], page 201,
renaming_declaration Section 9.5 [8.5], page 316,
basic_declaration Section 4.1 [3.1], page 49,
requeue_statement Section 10.5.4 [9.5.4], page 356,
simple_statement Section 6.1 [5.1], page 240,
restriction_parameter_argument Section 14.12 [13.12], page 535,
restriction Section 14.12 [13.12], page 535,
return_subtype_indication Section 7.5 [6.5], page 272,
extended_return_statement Section 7.5 [6.5], page 272,
scalar_constraint Section 4.2.2 [3.2.2], page 55,
constraint Section 4.2.2 [3.2.2], page 55,
select_alternative Section 10.7.1 [9.7.1], page 378,
selective_accept Section 10.7.1 [9.7.1], page 378,
select_statement Section 10.7 [9.7], page 377,
compound_statement Section 6.1 [5.1], page 240,
selected_component Section 5.1.3 [4.1.3], page 183,
name Section 5.1 [4.1], page 179,
selective_accept Section 10.7.1 [9.7.1], page 378,
select_statement Section 10.7 [9.7], page 377,
selector_name Section 5.1.3 [4.1.3], page 183,
component_choice_list Section 5.3.1 [4.3.1], page 191,
discriminant_association Section 4.7.1 [3.7.1], page 127,
formal_package_association Section 13.7 [12.7], page 474,
generic_association Section 13.3 [12.3], page 454,
parameter_association Section 7.4 [6.4], page 266,
selected_component Section 5.1.3 [4.1.3], page 183,
sequence_of_statements Section 6.1 [5.1], page 240,
abortable_part Section 10.7.4 [9.7.4], page 383,
accept_alternative Section 10.7.1 [9.7.1], page 378,
case_statement_alternative Section 6.4 [5.4], page 246,
conditional_entry_call Section 10.7.3 [9.7.3], page 382,
delay_alternative Section 10.7.1 [9.7.1], page 378,
entry_call_alternative Section 10.7.2 [9.7.2], page 381,
exception_handler Section 12.2 [11.2], page 420,
handled_sequence_of_statements Section 12.2 [11.2], page 420,
if_statement Section 6.3 [5.3], page 245,
loop_statement Section 6.5 [5.5], page 248,
selective_accept Section 10.7.1 [9.7.1], page 378,
triggering_alternative Section 10.7.4 [9.7.4], page 383,
signed_integer_type_definition Section 4.5.4 [3.5.4], page 95,
integer_type_definition Section 4.5.4 [3.5.4], page 95,
simple_expression Section 5.4 [4.4], page 201,
first_bit Section 14.5.1 [13.5.1], page 501,

last_bit Section 14.5.1 [13.5.1], page 501,
range Section 4.5 [3.5], page 76,
real_range_specification Section 4.5.7 [3.5.7], page 103,
relation Section 5.4 [4.4], page 201,
signed_integer_type_definition Section 4.5.4 [3.5.4], page 95,
simple_return_statement Section 7.5 [6.5], page 272,
simple_statement Section 6.1 [5.1], page 240,
simple_statement Section 6.1 [5.1], page 240,
statement Section 6.1 [5.1], page 240,
single_protected_declaration Section 10.4 [9.4], page 337,
object_declaration Section 4.3.1 [3.3.1], page 61,
single_task_declaration Section 10.1 [9.1], page 329,
object_declaration Section 4.3.1 [3.3.1], page 61,
slice Section 5.1.2 [4.1.2], page 182,
name Section 5.1 [4.1], page 179,
statement Section 6.1 [5.1], page 240,
sequence_of_statements Section 6.1 [5.1], page 240,
statement_identifier Section 6.1 [5.1], page 240,
block_statement Section 6.6 [5.6], page 251,
label Section 6.1 [5.1], page 240,
loop_statement Section 6.5 [5.5], page 248,
string_element Section 3.6 [2.6], page 42,
string_literal Section 3.6 [2.6], page 42,
string_literal Section 3.6 [2.6], page 42,
operator_symbol Section 7.1 [6.1], page 255,
primary Section 5.4 [4.4], page 201,
subprogram_body Section 7.3 [6.3], page 261,
library_unit_body Section 11.1.1 [10.1.1], page 394,
proper_body Section 4.11 [3.11], page 175,
protected_operation_item Section 10.4 [9.4], page 337,
subprogram_body_stub Section 11.1.3 [10.1.3], page 403,
body_stub Section 11.1.3 [10.1.3], page 403,
subprogram_declaration Section 7.1 [6.1], page 255,
basic_declaration Section 4.1 [3.1], page 49,
library_unit_declaration Section 11.1.1 [10.1.1], page 394,
protected_operation_declaration Section 10.4 [9.4], page 337,
protected_operation_item Section 10.4 [9.4], page 337,
subprogram_default Section 13.6 [12.6], page 470,
formal_abstract_subprogram_declaration Section 13.6 [12.6],
page 470,
formal_concrete_subprogram_declaration Section 13.6 [12.6],
page 470,

subprogram_renaming_declaration Section 9.5.4 [8.5.4], page 319,
library_unit_renaming_declaration Section 11.1.1 [10.1.1], page 394,
renaming_declaration Section 9.5 [8.5], page 316,
subprogram_specification Section 7.1 [6.1], page 255,
 abstract_subprogram_declaration Section 4.9.3 [3.9.3], page 149,
 formal_abstract_subprogram_declaration Section 13.6 [12.6],
 page 470,
 formal_concrete_subprogram_declaration Section 13.6 [12.6],
 page 470,
 generic_subprogram_declaration Section 13.1 [12.1], page 450,
 subprogram_body Section 7.3 [6.3], page 261,
 subprogram_body_stub Section 11.1.3 [10.1.3], page 403,
 subprogram_declaration Section 7.1 [6.1], page 255,
 subprogram_renaming_declaration Section 9.5.4 [8.5.4], page 319,
subtype_declaration Section 4.2.2 [3.2.2], page 55,
 basic_declaration Section 4.1 [3.1], page 49,
subtype_indication Section 4.2.2 [3.2.2], page 55,
 access_to_object_definition Section 4.10 [3.10], page 156,
 allocator Section 5.8 [4.8], page 230,
 component_definition Section 4.6 [3.6], page 114,
 derived_type_definition Section 4.4 [3.4], page 66,
 discrete_range Section 4.6.1 [3.6.1], page 117,
 discrete_subtype_definition Section 4.6 [3.6], page 114,
 object_declaration Section 4.3.1 [3.3.1], page 61,
 private_extension_declaration Section 8.3 [7.3], page 283,
 return_subtype_indication Section 7.5 [6.5], page 272,
 subtype_declaration Section 4.2.2 [3.2.2], page 55,
subtype_mark Section 4.2.2 [3.2.2], page 55,
 access_definition Section 4.10 [3.10], page 156,
 ancestor_part Section 5.3.2 [4.3.2], page 194,
 discriminant_specification Section 4.7 [3.7], page 123,
 explicit_generic_actual_parameter Section 13.3 [12.3], page 454,
 formal_derived_type_definition Section 13.5.1 [12.5.1], page 462,
 formal_object_declaration Section 13.4 [12.4], page 458,
 index_subtype_definition Section 4.6 [3.6], page 114,
 interface_list Section 4.9.4 [3.9.4], page 152,
 object_renaming_declaration Section 9.5.1 [8.5.1], page 317,
 parameter_and_result_profile Section 7.1 [6.1], page 255,
 parameter_specification Section 7.1 [6.1], page 255,
 qualified_expression Section 5.7 [4.7], page 229,
 relation Section 5.4 [4.4], page 201,
 subtype_indication Section 4.2.2 [3.2.2], page 55,
 type_conversion Section 5.6 [4.6], page 219,
 use_type_clause Section 9.4 [8.4], page 314,
subunit Section 11.1.3 [10.1.3], page 403,
 compilation_unit Section 11.1.1 [10.1.1], page 394,

`task_body` Section 10.1 [9.1], page 329,
`proper_body` Section 4.11 [3.11], page 175,
`task_body_stub` Section 11.1.3 [10.1.3], page 403,
`body_stub` Section 11.1.3 [10.1.3], page 403,
`task_definition` Section 10.1 [9.1], page 329,
`single_task_declaration` Section 10.1 [9.1], page 329,
`task_type_declaration` Section 10.1 [9.1], page 329,
`task_item` Section 10.1 [9.1], page 329,
`task_definition` Section 10.1 [9.1], page 329,
`task_type_declaration` Section 10.1 [9.1], page 329,
`full_type_declaration` Section 4.2.1 [3.2.1], page 53,
`term` Section 5.4 [4.4], page 201,
`simple_expression` Section 5.4 [4.4], page 201,
`terminate_alternative` Section 10.7.1 [9.7.1], page 378,
`select_alternative` Section 10.7.1 [9.7.1], page 378,
`timed_entry_call` Section 10.7.2 [9.7.2], page 381,
`select_statement` Section 10.7 [9.7], page 377,
`triggering_alternative` Section 10.7.4 [9.7.4], page 383,
`asynchronous_select` Section 10.7.4 [9.7.4], page 383,
`triggering_statement` Section 10.7.4 [9.7.4], page 383,
`triggering_alternative` Section 10.7.4 [9.7.4], page 383,
`type_conversion` Section 5.6 [4.6], page 219,
`name` Section 5.1 [4.1], page 179,
`type_declaration` Section 4.2.1 [3.2.1], page 53,
`basic_declaration` Section 4.1 [3.1], page 49,
`type_definition` Section 4.2.1 [3.2.1], page 53,
`full_type_declaration` Section 4.2.1 [3.2.1], page 53,
`unary_adding_operator` Section 5.5 [4.5], page 203,
`simple_expression` Section 5.4 [4.4], page 201,
`unconstrained_array_definition` Section 4.6 [3.6], page 114,
`array_type_definition` Section 4.6 [3.6], page 114,
`underline` ...
`based_numeral` Section 3.4.2 [2.4.2], page 41,
`numeral` Section 3.4.1 [2.4.1], page 40,
`unknown_discriminant_part` Section 4.7 [3.7], page 123,
`discriminant_part` Section 4.7 [3.7], page 123,
`use_clause` Section 9.4 [8.4], page 314,
`basic_declarative_item` Section 4.11 [3.11], page 175,
`context_item` Section 11.1.2 [10.1.2], page 399,
`generic_formal_part` Section 13.1 [12.1], page 450,
`use_package_clause` Section 9.4 [8.4], page 314,
`use_clause` Section 9.4 [8.4], page 314,

use_type_clause Section 9.4 [8.4], page 314,
use_clause Section 9.4 [8.4], page 314,
variant Section 4.8.1 [3.8.1], page 134,
variant_part Section 4.8.1 [3.8.1], page 134,
variant_part Section 4.8.1 [3.8.1], page 134,
component_list Section 4.8 [3.8], page 130,
with_clause Section 11.1.2 [10.1.2], page 399,
context_item Section 11.1.2 [10.1.2], page 399,

29 Annex Q Language-Defined Entities

1/2

This annex lists the language–defined entities of the language. A list of language–defined library units can be found in Chapter 15 [Annex A], page 553, "Chapter 15 [Annex A], page 553, Predefined Language Environment".

29.1 Q.1 Language-Defined Packages

1/2

This clause lists all language–defined packages.

Ada [4856], page 565,

Address_To_Access_Conversions

<child of> System [4643], page 517,

Arithmetic

<child of> Ada.Calendar [3769], page 364,

ASCII

<in> Standard [4845], page 562,

Assertions

<child of> Ada [4172], page 428,

Asynchronous_Task_Control

<child of> Ada [6972], page 1016,

Bounded

<child of> Ada.Strings [5216], page 611,

Bounded_IO

<child of> Ada.Text_IO [5830], page 739,

<child of> Ada.Wide_Text_IO [5854], page 745,

<child of> Ada.Wide_Wide_Text_IO [5855], page 745,

C

<child of> Interfaces [6470], page 902,

Calendar

<child of> Ada [3741], page 359,

Characters

<child of> Ada [4857], page 565,

COBOL

<child of> Interfaces [6581], page 932,

Command_Line

<child of> Ada [5918], page 755,

Complex_Arrays

<child of> Ada.Numerics [7371], page 1136,

Complex_Elementary_Functions

<child of> Ada.Numerics [7255], page 1092,

Complex_Text_IO

<child of> Ada [7273], page 1098,

Complex_Types

<child of> Ada.Numerics [7228], page 1087,

Complex_IO

<child of> Ada.Text_IO [7263], page 1097,

<child of> Ada.Wide_Text_IO [7275], page 1103,

<child of> Ada.Wide_Wide_Text_IO [7277], page 1103,

Constants

<child of> Ada.Strings.Maps [5324], page 636,

Containers

<child of> Ada [5989], page 779,

Conversions

<child of> Ada.Characters [5124], page 580,

Decimal

<child of> Ada [7172], page 1055,

Decimal_Conversions

<in> Interfaces.COBOL [6615], page 934,

Decimal_IO

<in> Ada.Text_IO [5795], page 707,

Decimal_Output

<in> Ada.Text_IO.Editing [7195], page 1074,

Direct_IO

<child of> Ada [5619], page 691,

Directories

<child of> Ada [5927], page 757,

Discrete_Random

<child of> Ada.Numerics [5459], page 656,

Dispatching

<child of> Ada [6809], page 978,

Doubly_Linked_Lists

<child of> Ada.Containers [6082], page 811,

Dynamic_Priorities

<child of> Ada [6888], page 996,

EDF

<child of> Ada.Dispatching [6852], page 988,

Editing

<child of> Ada.Text_IO [7183], page 1073,

<child of> Ada.Wide_Text_IO [7203], page 1081,

<child of> Ada.Wide_Wide_Text_IO [7205], page 1081,

Elementary_Functions

<child of> Ada.Numerics [5439], page 650,

Enumeration_IO
 <in> Ada.Text_IO [5805], page 708,
Environment_Variables
 <child of> Ada [5976], page 775,
Exceptions
 <child of> Ada [4140], page 423,
Execution_Time
 <child of> Ada [6989], page 1021,
Finalization
 <child of> Ada [3299], page 296,
Fixed
 <child of> Ada.Strings [5180], page 592,
Fixed_IO
 <in> Ada.Text_IO [5785], page 706,
Float_Random
 <child of> Ada.Numerics [5447], page 655,
Float_Text_IO
 <child of> Ada [5829], page 735,
Float_Wide_Text_IO
 <child of> Ada [5850], page 745,
Float_Wide_Wide_Text_IO
 <child of> Ada [5853], page 745,
Float_IO
 <in> Ada.Text_IO [5775], page 705,
Formatting
 <child of> Ada.Calendar [3773], page 365,
Fortran
 <child of> Interfaces [6635], page 945,
Generic_Complex_Arrays
 <child of> Ada.Numerics [7335], page 1130,
Generic_Complex_Elementary_Functions
 <child of> Ada.Numerics [7234], page 1091,
Generic_Complex_Types
 <child of> Ada.Numerics [7207], page 1084,
Generic_Dispatching_Constructor
 <child of> Ada.Tags [2049], page 141,
Generic_Elementary_Functions
 <child of> Ada.Numerics [5410], page 649,
Generic_Bounded_Length
 <in> Ada.Strings.Bounded [5217], page 611,
Generic_Keys
 <in> Ada.Containers.Hashed_Sets [6301], page 871,
 <in> Ada.Containers.Ordered_Sets [6370], page 881,

Generic_Real_Arrays
 <child of> Ada.Numerics [7319], page 1119,

Generic_Sorting
 <in> Ada.Containers.Doubly_Linked_Lists [6123], page 815,
 <in> Ada.Containers.Vectors [6061], page 787,

Group_Budgets
 <child of> Ada.Execution_Time [7017], page 1027,

Handling
 <child of> Ada.Characters [4860], page 566,

Hashed_Maps
 <child of> Ada.Containers [6150], page 840,

Hashed_Sets
 <child of> Ada.Containers [6258], page 867,

Indefinite_Doubly_Linked_Lists
 <child of> Ada.Containers [6390], page 888,

Indefinite_Hashed_Maps
 <child of> Ada.Containers [6391], page 889,

Indefinite_Hashed_Sets
 <child of> Ada.Containers [6393], page 890,

Indefinite_Ordered_Maps
 <child of> Ada.Containers [6392], page 890,

Indefinite_Ordered_Sets
 <child of> Ada.Containers [6394], page 891,

Indefinite_Vectors
 <child of> Ada.Containers [6389], page 887,

Information
 <child of> Ada.Directories [5974], page 774,

Integer_Text_IO
 <child of> Ada [5828], page 730,

Integer_Wide_Text_IO
 <child of> Ada [5849], page 745,

Integer_Wide_Wide_Text_IO
 <child of> Ada [5852], page 745,

Integer_IO
 <in> Ada.Text_IO [5757], page 703,

Interfaces [6465], page 900,

Interrupts
 <child of> Ada [6697], page 957,

IO_Exceptions
 <child of> Ada [5905], page 752,

Latin_1
 <child of> Ada.Characters [4893], page 573,

Machine_Code
 <child of> System [4652], page 519,

Maps
 <child of> Ada.Strings [5156], page 585,

Modular_IO
 <in> Ada.Text_IO [5766], page 704,

Names
 <child of> Ada.Interrupts [6707], page 958,

Numerics
 <child of> Ada [5406], page 648,

Ordered_Maps
 <child of> Ada.Containers [6194], page 846,

Ordered_Sets
 <child of> Ada.Containers [6321], page 876,

Pointers
 <child of> Interfaces.C [6554], page 923,

Real_Arrays
 <child of> Ada.Numerics [7331], page 1122,

Real_Time
 <child of> Ada [6931], page 1008,

Round_Robin
 <child of> Ada.Dispatching [6843], page 985,

RPC
 <child of> System [7155], page 1050,

Sequential_IO
 <child of> Ada [5596], page 683,

Single_Precision_Complex_Types
 <in> Interfaces.Fortran [6640], page 945,

Standard [4836], page 557,

Storage_Elements
 <child of> System [4631], page 516,

Storage_IO
 <child of> Ada [5649], page 695,

Storage_Pools
 <child of> System [4685], page 526,

Stream_IO
 <child of> Ada.Streams [5861], page 746,

Streams
 <child of> Ada [4759], page 538,

Strings
 <child of> Ada [5143], page 584,
 <child of> Interfaces.C [6532], page 916,

Synchronous_Task_Control

<child of> Ada [6963], page 1015,

System [4604], page 510,

Tags

<child of> Ada [2026], page 137,

Task_Attributes

<child of> Ada [6764], page 967,

Task_Identification

<child of> Ada [6746], page 965,

Task_Termination

<child of> Ada [6776], page 971,

Text_Streams

<child of> Ada.Text_IO [5896], page 751,

<child of> Ada.Wide_Text_IO [5899], page 751,

<child of> Ada.Wide_Wide_Text_IO [5902], page 752,

Text_IO

<child of> Ada [5668], page 698,

Time_Zones

<child of> Ada.Calendar [3765], page 364,

Timers

<child of> Ada.Execution_Time [7002], page 1024,

Timing_Events

<child of> Ada.Real_Time [7041], page 1031,

Unbounded

<child of> Ada.Strings [5274], page 625,

Unbounded_IO

<child of> Ada.Text_IO [5839], page 742,

<child of> Ada.Wide_Text_IO [5856], page 745,

<child of> Ada.Wide_Wide_Text_IO [5857], page 745,

Vectors

<child of> Ada.Containers [5996], page 780,

Wide_Bounded

<child of> Ada.Strings [5340], page 636,

Wide_Constants

<child of> Ada.Strings.Wide_Maps [5346], page 636, [5399], page 644,

Wide_Fixed

<child of> Ada.Strings [5339], page 636,

Wide_Hash

<child of> Ada.Strings [5342], page 636,

Wide_Maps

<child of> Ada.Strings [5347], page 637,

Wide_Text_IO

<child of> Ada [5848], page 745,

Wide_Unbounded
 <child of> Ada.Strings [5341], page 636,
Wide_Characters
 <child of> Ada [4858], page 566,
Wide_Wide_Constants
 <child of> Ada.Strings.Wide_Wide_Maps [5377], page 641,
Wide_Wide_Hash
 <child of> Ada.Strings [5373], page 641,
Wide_Wide_Text_IO
 <child of> Ada [5851], page 745,
Wide_Wide_Bounded
 <child of> Ada.Strings [5371], page 641,
Wide_Wide_Characters
 <child of> Ada [4859], page 566,
Wide_Wide_Fixed
 <child of> Ada.Strings [5370], page 641,
Wide_Wide_Maps
 <child of> Ada.Strings [5378], page 642,
Wide_Wide_Unbounded
 <child of> Ada.Strings [5372], page 641,

29.2 Q.2 Language-Defined Types and Subtypes

1/2

This clause lists all language–defined types and subtypes.

Address
 <in> System [4616], page 511,
Alignment
 <in> Ada.Strings [5151], page 584,
Alphanumeric
 <in> Interfaces.COBOL [6593], page 932,
Any_Priority <subtype of> Integer
 <in> System [4626], page 512,
Attribute_Handle
 <in> Ada.Task_Attributes [6765], page 967,
Binary
 <in> Interfaces.COBOL [6584], page 932,
Binary_Format
 <in> Interfaces.COBOL [6605], page 933,
Bit_Order
 <in> System [4622], page 512,

Boolean
 <in> Standard [4837], page 557,

Bounded_String
 <in> Ada.Strings.Bounded [5219], page 611,

Buffer_Type <subtype of> Storage_Array
 <in> Ada.Storage_IO [5651], page 695,

Byte
 <in> Interfaces.COBOL [6612], page 934,

Byte_Array
 <in> Interfaces.COBOL [6613], page 934,

C_float
 <in> Interfaces.C [6486], page 903,

Cause_Of_Termination
 <in> Ada.Task_Termination [6777], page 971,

char
 <in> Interfaces.C [6489], page 903,

char16_array
 <in> Interfaces.C [6513], page 905,

char16_t
 <in> Interfaces.C [6509], page 905,

char32_array
 <in> Interfaces.C [6523], page 906,

char32_t
 <in> Interfaces.C [6519], page 906,

char_array
 <in> Interfaces.C [6493], page 903,

char_array_access
 <in> Interfaces.C.Strings [6533], page 916,

Character
 <in> Standard [4842], page 560,

Character_Mapping
 <in> Ada.Strings.Maps [5170], page 587,

Character_Mapping_Function
 <in> Ada.Strings.Maps [5176], page 587,

Character_Range
 <in> Ada.Strings.Maps [5159], page 585,

Character_Ranges
 <in> Ada.Strings.Maps [5160], page 585,

Character_Sequence <subtype of> String
 <in> Ada.Strings.Maps [5166], page 586,

Character_Set
 <in> Ada.Strings.Maps [5157], page 585,
 <in> Interfaces.Fortran [6645], page 946,

chars_ptr
 <in> Interfaces.C.Strings [6534], page 916,

chars_ptr_array
 <in> Interfaces.C.Strings [6535], page 916,

COBOL_Character
 <in> Interfaces.COBOL [6590], page 932,

Complex
 <in> Ada.Numerics.Generic_Complex_Types [7208], page 1084,
 <in> Interfaces.Fortran [6641], page 945,

Complex_Matrix
 <in> Ada.Numerics.Generic_Complex_Arrays [7337], page 1130,

Complex_Vector
 <in> Ada.Numerics.Generic_Complex_Arrays [7336], page 1130,

Controlled
 <in> Ada.Finalization [3300], page 296,

Count
 <in> Ada.Direct_IO [5622], page 691,
 <in> Ada.Streams.Stream_IO [5865], page 746,
 <in> Ada.Text_IO [5671], page 698,

CPU_Time
 <in> Ada.Execution_Time [6990], page 1021,

Cursor
 <in> Ada.Containers.Doubly_Linked_Lists [6084], page 811,
 <in> Ada.Containers.Hashed_Maps [6152], page 840,
 <in> Ada.Containers.Hashed_Sets [6260], page 867,
 <in> Ada.Containers.Ordered_Maps [6197], page 846,
 <in> Ada.Containers.Ordered_Sets [6324], page 876,
 <in> Ada.Containers.Vectors [6000], page 780,

Day_Count
 <in> Ada.Calendar.Arithmetic [3770], page 364,

Day_Duration <subtype of> Duration
 <in> Ada.Calendar [3746], page 360,

Day_Name
 <in> Ada.Calendar.Formatting [3774], page 365,

Day_Number <subtype of> Integer
 <in> Ada.Calendar [3745], page 360,

Deadline <subtype of> Time
 <in> Ada.Dispatching.EDF [6853], page 988,

Decimal_Element
 <in> Interfaces.COBOL [6588], page 932,

Direction
 <in> Ada.Strings [5154], page 585,

Directory_Entry_Type
 <in> Ada.Directories [5949], page 759,

Display_Format
 <in> Interfaces.COBOL [6599], page 933,

double
 <in> Interfaces.C [6487], page 903,

Double_Precision
 <in> Interfaces.Fortran [6638], page 945,

Duration
 <in> Standard [4850], page 563,

Exception_Id
 <in> Ada.Exceptions [4141], page 423,

Exception_Occurrence
 <in> Ada.Exceptions [4146], page 423,

Exception_Occurrence_Access
 <in> Ada.Exceptions [4147], page 424,

Exit_Status
 <in> Ada.Command_Line [5922], page 755,

Extended_Index <subtype of> Index_Type'Base
 <in> Ada.Containers.Vectors [5997], page 780,

Field <subtype of> Integer
 <in> Ada.Text_IO [5674], page 698,

File_Access
 <in> Ada.Text_IO [5696], page 700,

File_Kind
 <in> Ada.Directories [5943], page 759,

File_Mode
 <in> Ada.Direct_IO [5621], page 691,
 <in> Ada.Sequential_IO [5598], page 683,
 <in> Ada.Streams.Stream_IO [5864], page 746,
 <in> Ada.Text_IO [5670], page 698,

File_Size
 <in> Ada.Directories [5944], page 759,

File_Type
 <in> Ada.Direct_IO [5620], page 691,
 <in> Ada.Sequential_IO [5597], page 683,
 <in> Ada.Streams.Stream_IO [5863], page 746,
 <in> Ada.Text_IO [5669], page 698,

Filter_Type
 <in> Ada.Directories [5950], page 759,

Float
 <in> Standard [4841], page 558,

Floating
 <in> Interfaces.COBOL [6582], page 932,

Fortran_Character
 <in> Interfaces.Fortran [6646], page 946,

Fortran_Integer
 <in> Interfaces.Fortran [6636], page 945,

Generator
 <in> Ada.Numerics.Discrete_Random [5460], page 656,
 <in> Ada.Numerics.Float_Random [5448], page 655,

Group_Budget
 <in> Ada.Execution_Time.Group_Budgets [7018], page 1027,

Group_Budget_Handler
 <in> Ada.Execution_Time.Group_Budgets [7019], page 1027,

Hash_Type
 <in> Ada.Containers [5990], page 779,

Hour_Number <subtype of> Natural
 <in> Ada.Calendar.Formatting [3783], page 366,

Imaginary
 <in> Ada.Numerics.Generic_Complex_Types [7209], page 1084,

Imaginary <subtype of> Imaginary
 <in> Interfaces.Fortran [6642], page 946,

int
 <in> Interfaces.C [6475], page 902,

Integer
 <in> Standard [4838], page 557,

Integer_Address
 <in> System.Storage_Elements [4637], page 517,

Interrupt_ID
 <in> Ada.Interrupts [6698], page 957,

Interrupt_Priority <subtype of> Any_Priority
 <in> System [4628], page 512,

ISO_646 <subtype of> Character
 <in> Ada.Characters.Handling [4878], page 567,

Leap_Seconds_Count <subtype of> Integer
 <in> Ada.Calendar.Arithmetic [3771], page 365,

Length_Range <subtype of> Natural
 <in> Ada.Strings.Bounded [5221], page 611,

Limited_Controlled
 <in> Ada.Finalization [3304], page 296,

List
 <in> Ada.Containers.Doubly_Linked_Lists [6083], page 811,

Logical
 <in> Interfaces.Fortran [6639], page 945,

long
 <in> Interfaces.C [6477], page 902,
Long_Binary
 <in> Interfaces.COBOL [6585], page 932,
long_double
 <in> Interfaces.C [6488], page 903,
Long_Floating
 <in> Interfaces.COBOL [6583], page 932,
Map
 <in> Ada.Containers.Hashing_Maps [6151], page 840,
 <in> Ada.Containers.Ordered_Maps [6196], page 846,
Membership
 <in> Ada.Strings [5153], page 585,
Minute_Number <subtype of> Natural
 <in> Ada.Calendar.Formatting [3784], page 366,
Month_Number <subtype of> Integer
 <in> Ada.Calendar [3744], page 360,
Name
 <in> System [4605], page 510,
Natural <subtype of> Integer
 <in> Standard [4839], page 557,
Number_Base <subtype of> Integer
 <in> Ada.Text_IO [5675], page 698,
Numeric
 <in> Interfaces.COBOL [6598], page 933,
Packed_Decimal
 <in> Interfaces.COBOL [6589], page 932,
Packed_Format
 <in> Interfaces.COBOL [6609], page 933,
Parameterless_Handler
 <in> Ada.Interrupts [6699], page 957,
Params_Stream_Type
 <in> System.RPC [7158], page 1051,
Partition_Id
 <in> System.RPC [7156], page 1050,
Picture
 <in> Ada.Text_IO.Editing [7184], page 1073,
plain_char
 <in> Interfaces.C [6483], page 902,
Pointer
 <in> Interfaces.C.Pointers [6555], page 923,
Positive <subtype of> Integer
 <in> Standard [4840], page 558,

Positive_Count <subtype of> Count
<in> Ada.Direct_IO [5623], page 691,
<in> Ada.Streams.Stream_IO [5866], page 746,
<in> Ada.Text_IO [5672], page 698,

Priority <subtype of> Any_Priority
<in> System [4627], page 512,

ptrdiff_t
<in> Interfaces.C [6484], page 903,

Real
<in> Interfaces.Fortran [6637], page 945,

Real_Matrix
<in> Ada.Numerics.Generic_Real_Arrays [7321], page 1119,

Real_Vector
<in> Ada.Numerics.Generic_Real_Arrays [7320], page 1119,

Root_Storage_Pool
<in> System.Storage_Pools [4686], page 526,

Root_Stream_Type
<in> Ada.Streams [4761], page 538,

RPC_Receiver
<in> System.RPC [7163], page 1051,

Search_Type
<in> Ada.Directories [5951], page 760,

Second_Duration <subtype of> Day_Duration
<in> Ada.Calendar.Formatting [3786], page 366,

Second_Number <subtype of> Natural
<in> Ada.Calendar.Formatting [3785], page 366,

Seconds_Count
<in> Ada.Real_Time [6950], page 1009,

Set
<in> Ada.Containers.Hashed_Sets [6259], page 867,
<in> Ada.Containers.Ordered_Sets [6323], page 876,

short
<in> Interfaces.C [6476], page 902,

signed_char
<in> Interfaces.C [6478], page 902,

size_t
<in> Interfaces.C [6485], page 903,

State
<in> Ada.Numerics.Discrete_Random [5464], page 656,
<in> Ada.Numerics.Float_Random [5453], page 655,

Storage_Array
<in> System.Storage_Elements [4635], page 516,

Storage_Count <subtype of> Storage_Offset
<in> System.Storage_Elements [4633], page 516,

Storage_Element
<in> System.Storage_Elements [4634], page 516,

Storage_Offset
<in> System.Storage_Elements [4632], page 516,

Stream_Access
<in> Ada.Streams.Stream_IO [5862], page 746,
<in> Ada.Text_IO.Text_Streams [5897], page 751,
<in> Ada.Wide_Text_IO.Text_Streams [5900], page 751,
<in> Ada.Wide_Wide_Text_IO.Text_Streams [5903], page 752,

Stream_Element
<in> Ada.Streams [4762], page 538,

Stream_Element_Array
<in> Ada.Streams [4765], page 539,

Stream_Element_Count <subtype of> Stream_Element_Offset
<in> Ada.Streams [4764], page 539,

Stream_Element_Offset
<in> Ada.Streams [4763], page 538,

String
<in> Standard [4847], page 562,

String_Access
<in> Ada.Strings.Unbounded [5278], page 625,

Suspension_Object
<in> Ada.Synchronous_Task_Control [6964], page 1015,

Tag
<in> Ada.Tags [2027], page 137,

Tag_Array
<in> Ada.Tags [2037], page 138,

Task_Array
<in> Ada.Execution_Time.Group_Budgets [7020], page 1028,

Task_Id
<in> Ada.Task_Identification [6747], page 965,

Termination_Handler
<in> Ada.Task_Termination [6778], page 971,

Time
<in> Ada.Calendar [3742], page 359,
<in> Ada.Real_Time [6932], page 1008,

Time_Offset
<in> Ada.Calendar.Time_Zones [3766], page 364,

Time_Span
<in> Ada.Real_Time [6936], page 1008,

Timer
 <in> Ada.Execution_Time.Timers [7003], page 1025,

Timer_Handler
 <in> Ada.Execution_Time.Timers [7004], page 1025,

Timing_Event
 <in> Ada.Real_Time.Timing_Events [7042], page 1031,

Timing_Event_Handler
 <in> Ada.Real_Time.Timing_Events [7043], page 1031,

Trim_End
 <in> Ada.Strings [5155], page 585,

Truncation
 <in> Ada.Strings [5152], page 585,

Type_Set
 <in> Ada.Text_IO [5676], page 698,

Unbounded_String
 <in> Ada.Strings.Unbounded [5275], page 625,

Uniformly_Distributed <subtype of> Float
 <in> Ada.Numerics.Float_Random [5449], page 655,

unsigned
 <in> Interfaces.C [6479], page 902,

unsigned_char
 <in> Interfaces.C [6482], page 902,

unsigned_long
 <in> Interfaces.C [6481], page 902,

unsigned_short
 <in> Interfaces.C [6480], page 902,

Vector
 <in> Ada.Containers.Vectors [5999], page 780,

wchar_array
 <in> Interfaces.C [6503], page 904,

wchar_t
 <in> Interfaces.C [6499], page 904,

Wide_Character
 <in> Standard [4843], page 562,

Wide_Character_Mapping
 <in> Ada.Strings.Wide_Maps [5361], page 638,

Wide_Character_Mapping_Function
 <in> Ada.Strings.Wide_Maps [5367], page 639,

Wide_Character_Range
 <in> Ada.Strings.Wide_Maps [5350], page 637,

Wide_Character_Ranges
 <in> Ada.Strings.Wide_Maps [5351], page 637,

Wide_Character_Sequence <subtype of> Wide_String
 <in> Ada.Strings.Wide_Maps [5357], page 638,

Wide_Character_Set
 <in> Ada.Strings.Wide_Maps [5348], page 637,

Wide_String
 <in> Standard [4848], page 563,

Wide_Wide_Character
 <in> Standard [4844], page 562,

Wide_Wide_Character_Mapping
 <in> Ada.Strings.Wide_Wide_Maps [5392], page 644,

Wide_Wide_Character_Mapping_Function
 <in> Ada.Strings.Wide_Wide_Maps [5398], page 644,

Wide_Wide_Character_Range
 <in> Ada.Strings.Wide_Wide_Maps [5381], page 642,

Wide_Wide_Character_Ranges
 <in> Ada.Strings.Wide_Wide_Maps [5382], page 642,

Wide_Wide_Character_Sequence <subtype of> Wide_Wide_String
 <in> Ada.Strings.Wide_Wide_Maps [5388], page 643,

Wide_Wide_Character_Set
 <in> Ada.Strings.Wide_Wide_Maps [5379], page 642,

Wide_Wide_String
 <in> Standard [4849], page 563,

Year_Number <subtype of> Integer
 <in> Ada.Calendar [3743], page 360,

29.3 Q.3 Language-Defined Subprograms

1/2

This clause lists all language–defined subprograms.

Abort_Task <in> Ada.Task_Identification [6751], page 965,

Actual_Quantum
 <in> Ada.Dispatching.Round_Robin [6847], page 986,

Add
 <in> Ada.Execution_Time.Group_Budgets [7028], page 1028,

Add_Task
 <in> Ada.Execution_Time.Group_Budgets [7022], page 1028,

Adjust <in> Ada.Finalization [3302], page 296,

Allocate <in> System.Storage_Pools [4687], page 526,

Append
 <in> Ada.Containers.Doubly_Linked_Lists [6099], page 813,
 <in> Ada.Containers.Vectors [6032], page 784, [6033], page 784,
 <in> Ada.Strings.Bounded [5226], page 612, [5227], page 612, [5228], page 612, [5229],

page 612, [5230], page 612, [5231], page 612, [5232], page 612, [5233], page 613,
<in> Ada.Strings.Unbounded [5284], page 626, [5285], page 626, [5286], page 626,

Arccos
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7244], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5425], page 649,

Arccosh
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7252], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5436], page 650,

Arccot
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7246], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5430], page 649,

Arccoth
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7254], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5438], page 650,

Arcsin
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7243], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5424], page 649,

Arcsinh
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7251], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5435], page 650,

Arctan
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7245], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5428], page 649,

Arctanh
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7253], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5437], page 650,

Argument
<in> Ada.Command_Line [5920], page 755,
<in> Ada.Numerics.Generic_Complex_Arrays [7345], page 1131, [7359], page 1133,
<in> Ada.Numerics.Generic_Complex_Types [7222], page 1085,

Argument_Count <in> Ada.Command_Line [5919], page 755,

Attach_Handler <in> Ada.Interrupts [6703], page 957,

Base_Name <in> Ada.Directories [5941], page 759,

Blank_When_Zero
<in> Ada.Text_IO.Editing [7188], page 1073,

Bounded_Slice <in> Ada.Strings.Bounded [5237], page 614, [5238], page 614,

Budget_Has_Expired
<in> Ada.Execution_Time.Group_Budgets [7029], page 1028,

Budget_Remaining
<in> Ada.Execution_Time.Group_Budgets [7030], page 1028,

Cancel_Handler
<in> Ada.Execution_Time.Group_Budgets [7033], page 1028,

- <in> Ada.Execution_Time.Timers [7009], page 1025,
- <in> Ada.Real_Time.Timing_Events [7047], page 1031,

Capacity

- <in> Ada.Containers.Hashtable [6155], page 840,
- <in> Ada.Containers.Hashtable_Sets [6265], page 868,
- <in> Ada.Containers.Vectors [6005], page 781,

Ceiling

- <in> Ada.Containers.Ordered_Maps [6232], page 850,
- <in> Ada.Containers.Ordered_Sets [6365], page 880, [6379], page 882,

Clear

- <in> Ada.Containers.Doubly_Linked_Lists [6089], page 811,
- <in> Ada.Containers.Hashtable [6159], page 840,
- <in> Ada.Containers.Hashtable_Sets [6269], page 868,
- <in> Ada.Containers.Ordered_Maps [6202], page 847,
- <in> Ada.Containers.Ordered_Sets [6331], page 877,
- <in> Ada.Containers.Vectors [6010], page 782,
- <in> Ada.Environment_Variables [5981], page 775,

Clock

- <in> Ada.Calendar [3747], page 360,
- <in> Ada.Execution_Time [6995], page 1022,
- <in> Ada.Real_Time [6942], page 1008,

Close

- <in> Ada.Direct_IO [5626], page 691,
- <in> Ada.Sequential_IO [5601], page 684,
- <in> Ada.Streams.Stream_IO [5869], page 747,
- <in> Ada.Text_IO [5679], page 699,

Col <in> Ada.Text_IO [5732], page 702,

Command_Name <in> Ada.Command_Line [5921], page 755,

Compose <in> Ada.Directories [5942], page 759,

Compose_From_Cartesian

- <in> Ada.Numerics.Generic_Complex_Arrays [7343], page 1131, [7355], page 1133,
- <in> Ada.Numerics.Generic_Complex_Types [7218], page 1085,

Compose_From_Polar

- <in> Ada.Numerics.Generic_Complex_Arrays [7348], page 1131, [7360], page 1133,
- <in> Ada.Numerics.Generic_Complex_Types [7225], page 1085,

Conjugate

- <in> Ada.Numerics.Generic_Complex_Arrays [7349], page 1131, [7362], page 1134,
- <in> Ada.Numerics.Generic_Complex_Types [7226], page 1085, [7227], page 1086,

Containing_Directory

- <in> Ada.Directories [5939], page 758,

Contains

- <in> Ada.Containers.Doubly_Linked_Lists [6119], page 814,
- <in> Ada.Containers.Hashtable [6179], page 843,
- <in> Ada.Containers.Hashtable_Sets [6295], page 871, [6308], page 872,

- <in> Ada.Containers.Ordered_Maps [6233], page 850,
- <in> Ada.Containers.Ordered_Sets [6366], page 880, [6380], page 882,
- <in> Ada.Containers.Vectors [6057], page 787,

Continue

- <in> Ada.Asynchronous_Task_Control [6974], page 1016,

Copy_Array <in> Interfaces.C.Pointers [6563], page 924,

Copy_File <in> Ada.Directories [5936], page 758,

Copy_Terminated_Array

- <in> Interfaces.C.Pointers [6562], page 924,

Cos

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7240], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5417], page 649,

Cosh

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7248], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5432], page 650,

Cot

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7242], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5422], page 649,

Coth

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7250], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5434], page 650,

Count

- <in> Ada.Strings.Bounded [5247], page 617, [5248], page 617, [5249], page 617,
- <in> Ada.Strings.Fixed [5190], page 594, [5191], page 594, [5192], page 594,
- <in> Ada.Strings.Unbounded [5300], page 630, [5301], page 630, [5302], page 630,

Create

- <in> Ada.Direct_IO [5624], page 691,
- <in> Ada.Sequential_IO [5599], page 683,
- <in> Ada.Streams.Stream_IO [5867], page 746,
- <in> Ada.Text_IO [5677], page 699,

Create_Directory <in> Ada.Directories [5930], page 758,

Create_Path <in> Ada.Directories [5932], page 758,

Current_Directory <in> Ada.Directories [5928], page 757,

Current_Error <in> Ada.Text_IO [5695], page 700, [5702], page 700,

Current_Handler

- <in> Ada.Execution_Time.Group_Budgets [7032], page 1028,
- <in> Ada.Execution_Time.Timers [7008], page 1025,
- <in> Ada.Interrupts [6702], page 957,
- <in> Ada.Real_Time.Timing_Events [7046], page 1031,

Current_Input <in> Ada.Text_IO [5693], page 700, [5700], page 700,

Current_Output <in> Ada.Text_IO [5694], page 700, [5701], page 700,

Current_State

- <in> Ada.Synchronous_Task_Control [6967], page 1015,

Current_Task

<in> Ada.Task_Identification [6750], page 965,

Current_Task_Fallback_Handler

<in> Ada.Task_Termination [6780], page 972,

Day

<in> Ada.Calendar [3750], page 360,

<in> Ada.Calendar.Formatting [3789], page 366,

Day_of_Week

<in> Ada.Calendar.Formatting [3782], page 366,

Deallocate <in> System.Storage_Pools [4688], page 527,

Decrement <in> Interfaces.C.Pointers [6560], page 923,

Delay_Until_And_Set_Deadline

<in> Ada.Dispatching.EDF [6856], page 988,

Delete

<in> Ada.Containers.Doubly_Linked_Lists [6100], page 813,

<in> Ada.Containers.Hashed_Maps [6172], page 842, [6173], page 842,

<in> Ada.Containers.Hashed_Sets [6279], page 869, [6280], page 869, [6306], page 872,

<in> Ada.Containers.Ordered_Maps [6215], page 848, [6216], page 848,

<in> Ada.Containers.Ordered_Sets [6341], page 878, [6342], page 878, [6376], page 882,

<in> Ada.Containers.Vectors [6036], page 785, [6037], page 785,

<in> Ada.Direct_IO [5627], page 691,

<in> Ada.Sequential_IO [5602], page 684,

<in> Ada.Streams.Stream_IO [5870], page 747,

<in> Ada.Strings.Bounded [5261], page 619, [5262], page 619,

<in> Ada.Strings.Fixed [5204], page 596, [5205], page 596,

<in> Ada.Strings.Unbounded [5314], page 632, [5315], page 632,

<in> Ada.Text_IO [5680], page 699,

Delete_Directory <in> Ada.Directories [5931], page 758,

Delete_File <in> Ada.Directories [5934], page 758,

Delete_First

<in> Ada.Containers.Doubly_Linked_Lists [6101], page 813,

<in> Ada.Containers.Ordered_Maps [6217], page 849,

<in> Ada.Containers.Ordered_Sets [6343], page 878,

<in> Ada.Containers.Vectors [6038], page 785,

Delete_Last

<in> Ada.Containers.Doubly_Linked_Lists [6102], page 813,

<in> Ada.Containers.Ordered_Maps [6218], page 849,

<in> Ada.Containers.Ordered_Sets [6344], page 878,

<in> Ada.Containers.Vectors [6039], page 785,

Delete_Tree <in> Ada.Directories [5933], page 758,

Dereference_Error

<in> Interfaces.C.Strings [6541], page 916,

Descendant_Tag <in> Ada.Tags [2034], page 138,

Detach_Handler <in> Ada.Interrupts [6705], page 958,

Determinant

- <in> Ada.Numerics.Generic_Complex_Arrays [7367], page 1135,
- <in> Ada.Numerics.Generic_Real_Arrays [7327], page 1121,

Difference

- <in> Ada.Calendar.Arithmetic [3772], page 365,
- <in> Ada.Containers.Hash_Sets [6285], page 870, [6286], page 870,
- <in> Ada.Containers.Ordered_Sets [6349], page 879, [6350], page 879,

Divide <in> Ada.Decimal [7178], page 1055,

Do_APC <in> System.RPC [7162], page 1051,

Do_RPC <in> System.RPC [7161], page 1051,

Eigensystem

- <in> Ada.Numerics.Generic_Complex_Arrays [7369], page 1136,
- <in> Ada.Numerics.Generic_Real_Arrays [7329], page 1122,

Eigenvalues

- <in> Ada.Numerics.Generic_Complex_Arrays [7368], page 1136,
- <in> Ada.Numerics.Generic_Real_Arrays [7328], page 1122,

Element

- <in> Ada.Containers.Doubly_Linked_Lists [6090], page 811,
- <in> Ada.Containers.Hash_Maps [6161], page 841, [6178], page 842,
- <in> Ada.Containers.Hash_Sets [6270], page 868, [6303], page 872,
- <in> Ada.Containers.Ordered_Maps [6204], page 847, [6230], page 850,
- <in> Ada.Containers.Ordered_Sets [6332], page 877, [6373], page 882,
- <in> Ada.Containers.Vectors [6013], page 782, [6014], page 782,
- <in> Ada.Strings.Bounded [5234], page 613,
- <in> Ada.Strings.Unbounded [5287], page 627,

End_Of_File

- <in> Ada.Direct_IO [5641], page 692,
- <in> Ada.Sequential_IO [5611], page 684,
- <in> Ada.Streams.Stream_IO [5877], page 747,
- <in> Ada.Text_IO [5726], page 701,

End_Of_Line <in> Ada.Text_IO [5717], page 701,

End_Of_Page <in> Ada.Text_IO [5723], page 701,

End_Search <in> Ada.Directories [5953], page 760,

Equivalent_Elements

- <in> Ada.Containers.Hash_Sets [6297], page 871, [6298], page 871, [6299], page 871,
- <in> Ada.Containers.Ordered_Sets [6322], page 876,

Equivalent_Keys

- <in> Ada.Containers.Hash_Maps [6181], page 843, [6182], page 843, [6183], page 843,
- <in> Ada.Containers.Ordered_Maps [6195], page 846,
- <in> Ada.Containers.Ordered_Sets [6371], page 881,

Equivalent_Sets

- <in> Ada.Containers.Hash_Sets [6263], page 868,
- <in> Ada.Containers.Ordered_Sets [6327], page 877,

Establish_RPC_Receiver <in> System.RPC [7164], page 1051,

Exception_Identity <in> Ada.Exceptions [4152], page 424,
Exception_Information
 <in> Ada.Exceptions [4156], page 424,
Exception_Message <in> Ada.Exceptions [4150], page 424,
Exception_Name <in> Ada.Exceptions [4143], page 423, [4153], page 424,
Exchange_Handler <in> Ada.Interrupts [6704], page 957,
Exclude
 <in> Ada.Containers.Hashing_Maps [6171], page 842,
 <in> Ada.Containers.Hashing_Sets [6278], page 869, [6305], page 872,
 <in> Ada.Containers.Ordered_Maps [6214], page 848,
 <in> Ada.Containers.Ordered_Sets [6340], page 878, [6375], page 882,
Exists
 <in> Ada.Directories [5945], page 759,
 <in> Ada.Environment_Variables [5978], page 775,
Exp
 <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7237], page 1092,
 <in> Ada.Numerics.Generic_Elementary_Functions [5414], page 649,
Expanded_Name <in> Ada.Tags [2029], page 138,
Extension <in> Ada.Directories [5940], page 759,
External_Tag <in> Ada.Tags [2032], page 138,
Finalize <in> Ada.Finalization [3303], page 296, [3306], page 296,
Find
 <in> Ada.Containers.Doubly_Linked_Lists [6117], page 814,
 <in> Ada.Containers.Hashing_Maps [6177], page 842,
 <in> Ada.Containers.Hashing_Sets [6294], page 871, [6307], page 872,
 <in> Ada.Containers.Ordered_Maps [6229], page 849,
 <in> Ada.Containers.Ordered_Sets [6363], page 880, [6377], page 882,
 <in> Ada.Containers.Vectors [6054], page 786,
Find_Index <in> Ada.Containers.Vectors [6053], page 786,
Find-Token
 <in> Ada.Strings.Bounded [5250], page 617,
 <in> Ada.Strings.Fixed [5193], page 594,
 <in> Ada.Strings.Unbounded [5303], page 630,
First
 <in> Ada.Containers.Doubly_Linked_Lists [6109], page 814,
 <in> Ada.Containers.Hashing_Maps [6174], page 842,
 <in> Ada.Containers.Hashing_Sets [6291], page 870,
 <in> Ada.Containers.Ordered_Maps [6219], page 849,
 <in> Ada.Containers.Ordered_Sets [6355], page 879,
 <in> Ada.Containers.Vectors [6044], page 785,
First_Element
 <in> Ada.Containers.Doubly_Linked_Lists [6110], page 814,
 <in> Ada.Containers.Ordered_Maps [6220], page 849,

- <in> Ada.Containers.Ordered_Sets [6356], page 879,
- <in> Ada.Containers.Vectors [6045], page 785,

First_Index <in> Ada.Containers.Vectors [6043], page 785,

First_Key

- <in> Ada.Containers.Ordered_Maps [6221], page 849,

Floor

- <in> Ada.Containers.Ordered_Maps [6231], page 850,
- <in> Ada.Containers.Ordered_Sets [6364], page 880, [6378], page 882,

Flush

- <in> Ada.Streams.Stream_IO [5887], page 748,
- <in> Ada.Text_IO [5703], page 700,

Form

- <in> Ada.Direct_IO [5632], page 692,
- <in> Ada.Sequential_IO [5607], page 684,
- <in> Ada.Streams.Stream_IO [5875], page 747,
- <in> Ada.Text_IO [5685], page 699,

Free

- <in> Ada.Strings.Unbounded [5279], page 625,
- <in> Interfaces.C.Strings [6540], page 916,

Full_Name <in> Ada.Directories [5937], page 758, [5957], page 760,

Generic_Array_Sort

- <child of> Ada.Containers [6395], page 891,

Generic_Constrained_Array_Sort

- <child of> Ada.Containers [6397], page 892,

Get

- <in> Ada.Text_IO [5738], page 702, [5747], page 703, [5761], page 703, [5764], page 704, [5769], page 704, [5773], page 705, [5779], page 705, [5783], page 705, [5789], page 706, [5793], page 706, [5799], page 707, [5803], page 707, [5809], page 708, [5812], page 708,
- <in> Ada.Text_IO.Complex_IO [7267], page 1098, [7271], page 1098,

Get_Deadline <in> Ada.Dispatching.EDF [6857], page 988,

Get_Immediate <in> Ada.Text_IO [5744], page 702, [5745], page 702,

Get_Line

- <in> Ada.Text_IO [5751], page 703, [5754], page 703,
- <in> Ada.Text_IO.Bounded_IO [5835], page 740, [5836], page 740, [5837], page 740, [5838], page 740,
- <in> Ada.Text_IO.Unbounded_IO [5844], page 742, [5845], page 743, [5846], page 743, [5847], page 743,

Get_Next_Entry <in> Ada.Directories [5955], page 760,

Get_Priority

- <in> Ada.Dynamic_Priorities [6890], page 997,

Has_Element

- <in> Ada.Containers.Doubly_Linked_Lists [6120], page 815,
- <in> Ada.Containers.Hashed_Maps [6180], page 843,

- <in> Ada.Containers.Hashed_Sets [6296], page 871,
- <in> Ada.Containers.Ordered_Maps [6234], page 850,
- <in> Ada.Containers.Ordered_Sets [6367], page 880,
- <in> Ada.Containers.Vectors [6058], page 787,

Hash

- <child of> Ada.Strings [5403], page 646,
- <child of> Ada.Strings.Bounded [5404], page 647,
- <child of> Ada.Strings.Unbounded [5405], page 647,

Head

- <in> Ada.Strings.Bounded [5267], page 619, [5268], page 620,
- <in> Ada.Strings.Fixed [5210], page 597, [5211], page 597,
- <in> Ada.Strings.Unbounded [5320], page 633, [5321], page 633,

Hold <in> Ada.Asynchronous_Task_Control [6973], page 1016,

Hour <in> Ada.Calendar.Formatting [3790], page 366,

Im

- <in> Ada.Numerics.Generic_Complex_Arrays [7339], page 1130, [7352], page 1133,
- <in> Ada.Numerics.Generic_Complex_Types [7214], page 1084,

Image

- <in> Ada.Calendar.Formatting [3801], page 368, [3803], page 369,
- <in> Ada.Numerics.Discrete_Random [5468], page 656,
- <in> Ada.Numerics.Float_Random [5457], page 655,
- <in> Ada.Task_Identification [6749], page 965,
- <in> Ada.Text_IO.Editing [7198], page 1074,

Include

- <in> Ada.Containers.Hashed_Maps [6169], page 842,
- <in> Ada.Containers.Hashed_Sets [6276], page 869,
- <in> Ada.Containers.Ordered_Maps [6212], page 848,
- <in> Ada.Containers.Ordered_Sets [6338], page 878,

Increment <in> Interfaces.C.Pointers [6559], page 923,

Index

- <in> Ada.Direct_IO [5639], page 692,
- <in> Ada.Streams.Stream_IO [5884], page 748,
- <in> Ada.Strings.Bounded [5239], page 615, [5240], page 615, [5241], page 616, [5242], page 616, [5243], page 616, [5244], page 616,
- <in> Ada.Strings.Fixed [5182], page 592, [5183], page 593, [5184], page 593, [5185], page 593, [5186], page 593, [5187], page 593,
- <in> Ada.Strings.Unbounded [5292], page 629, [5293], page 629, [5294], page 629, [5295], page 629, [5296], page 629, [5297], page 630,

Index_Non_Blank

- <in> Ada.Strings.Bounded [5245], page 616, [5246], page 616,
- <in> Ada.Strings.Fixed [5188], page 593, [5189], page 593,
- <in> Ada.Strings.Unbounded [5298], page 630, [5299], page 630,

Initialize <in> Ada.Finalization [3301], page 296, [3305], page 296,

Insert

- <in> Ada.Containers.Doubly_Linked_Lists [6095], page 812, [6096], page 812, [6097],

page 812,

- <in> Ada.Containers.Hashtable [6166], page 841, [6167], page 841, [6168], page 841,
- <in> Ada.Containers.Hashtable_Sets [6274], page 868, [6275], page 869,
- <in> Ada.Containers.Ordered_Maps [6209], page 848, [6210], page 848, [6211], page 848,
- <in> Ada.Containers.Ordered_Sets [6336], page 877, [6337], page 877,
- <in> Ada.Containers.Vectors [6022], page 783, [6023], page 783, [6024], page 783, [6025], page 783, [6026], page 783, [6027], page 784, [6028], page 784, [6029], page 784,
- <in> Ada.Strings.Bounded [5257], page 618, [5258], page 618,
- <in> Ada.Strings.Fixed [5200], page 595, [5201], page 595,
- <in> Ada.Strings.Unbounded [5310], page 631, [5311], page 632,

Insert_Space

- <in> Ada.Containers.Vectors [6034], page 784, [6035], page 784,

Interface_Anccestor_Tags <in> Ada.Tags [2038], page 138,

Internal_Tag <in> Ada.Tags [2033], page 138,

Intersection

- <in> Ada.Containers.Hashtable_Sets [6283], page 869, [6284], page 870,
- <in> Ada.Containers.Ordered_Sets [6347], page 878, [6348], page 879,

Inverse

- <in> Ada.Numerics.Generic_Complex_Arrays [7366], page 1135,
- <in> Ada.Numerics.Generic_Real_Arrays [7326], page 1121,

Is_A_Group_Member

- <in> Ada.Execution_Time.Group_Budgets [7025], page 1028,

Is_Alphanumeric

- <in> Ada.Characters.Handling [4870], page 567,

Is_Attached <in> Ada.Interrupts [6701], page 957,

Is_Basic <in> Ada.Characters.Handling [4866], page 566,

Is_Callable

- <in> Ada.Task_Identification [6753], page 965,

Is_Character

- <in> Ada.Characters.Conversions [5127], page 580,

Is_Control <in> Ada.Characters.Handling [4861], page 566,

Is_Decimal_Digit

- <in> Ada.Characters.Handling [4868], page 566,

Is_Descendant_At_Same_Level

- <in> Ada.Tags [2035], page 138,

Is_Digit <in> Ada.Characters.Handling [4867], page 566,

Is_Empty

- <in> Ada.Containers.Doubly_Linked_Lists [6088], page 811,
- <in> Ada.Containers.Hashtable [6158], page 840,
- <in> Ada.Containers.Hashtable_Sets [6268], page 868,
- <in> Ada.Containers.Ordered_Maps [6201], page 847,
- <in> Ada.Containers.Ordered_Sets [6330], page 877,
- <in> Ada.Containers.Vectors [6009], page 781,

Is_Graphic <in> Ada.Characters.Handling [4862], page 566,
Is_Held
 <in> Ada.Asynchronous_Task_Control [6975], page 1016,
Is_Hexadecimal_Digit
 <in> Ada.Characters.Handling [4869], page 566,
Is_In
 <in> Ada.Strings.Maps [5164], page 586,
 <in> Ada.Strings.Wide_Maps [5355], page 638,
 <in> Ada.Strings.Wide_Wide_Maps [5386], page 643,
Is_ISO_646 <in> Ada.Characters.Handling [4880], page 567,
Is_Letter <in> Ada.Characters.Handling [4863], page 566,
Is_Lower <in> Ada.Characters.Handling [4864], page 566,
Is_Member
 <in> Ada.Execution_Time.Group_Budgets [7024], page 1028,
Is_Nul_Terminated <in> Interfaces.C [6494], page 903, [6504], page 905, [6524], page 907,
[6514], page 906,
Is_Open
 <in> Ada.Direct_IO [5633], page 692,
 <in> Ada.Sequential_IO [5608], page 684,
 <in> Ada.Streams.Stream_IO [5876], page 747,
 <in> Ada.Text_IO [5686], page 699,
Is_Reserved <in> Ada.Interrupts [6700], page 957,
Is_Round_Robin
 <in> Ada.Dispatching.Round_Robin [6848], page 986,
Is_Sorted
 <in> Ada.Containers.Doubly_Linked_Lists [6124], page 815,
 <in> Ada.Containers.Vectors [6062], page 787,
Is_Special <in> Ada.Characters.Handling [4871], page 567,
Is_String
 <in> Ada.Characters.Conversions [5128], page 580,
Is_Subset
 <in> Ada.Containers.Hashed_Sets [6290], page 870,
 <in> Ada.Containers.Ordered_Sets [6354], page 879,
 <in> Ada.Strings.Maps [5165], page 586,
 <in> Ada.Strings.Wide_Maps [5356], page 638,
 <in> Ada.Strings.Wide_Wide_Maps [5387], page 643,
Is_Terminated
 <in> Ada.Task_Identification [6752], page 965,
Is_Upper <in> Ada.Characters.Handling [4865], page 566,
Is_Wide_Character
 <in> Ada.Characters.Conversions [5129], page 580,
Is_Wide_String
 <in> Ada.Characters.Conversions [5130], page 580,

Iterate

- <in> Ada.Containers.Doubly_Linked_Lists [6121], page 815,
- <in> Ada.Containers.Hashed_Maps [6184], page 843,
- <in> Ada.Containers.Hashed_Sets [6300], page 871,
- <in> Ada.Containers.Ordered_Maps [6235], page 850,
- <in> Ada.Containers.Ordered_Sets [6368], page 881,
- <in> Ada.Containers.Vectors [6059], page 787,
- <in> Ada.Environment_Variables [5982], page 775,

Key

- <in> Ada.Containers.Hashed_Maps [6160], page 840,
- <in> Ada.Containers.Hashed_Sets [6302], page 872,
- <in> Ada.Containers.Ordered_Maps [6203], page 847,
- <in> Ada.Containers.Ordered_Sets [6372], page 882,

Kind <in> Ada.Directories [5946], page 759, [5958], page 760,

Last

- <in> Ada.Containers.Doubly_Linked_Lists [6111], page 814,
- <in> Ada.Containers.Ordered_Maps [6222], page 849,
- <in> Ada.Containers.Ordered_Sets [6357], page 880,
- <in> Ada.Containers.Vectors [6047], page 786,

Last_Element

- <in> Ada.Containers.Doubly_Linked_Lists [6112], page 814,
- <in> Ada.Containers.Ordered_Maps [6223], page 849,
- <in> Ada.Containers.Ordered_Sets [6358], page 880,
- <in> Ada.Containers.Vectors [6048], page 786,

Last_Index <in> Ada.Containers.Vectors [6046], page 786,

Last_Key

- <in> Ada.Containers.Ordered_Maps [6224], page 849,

Length

- <in> Ada.Containers.Doubly_Linked_Lists [6087], page 811,
- <in> Ada.Containers.Hashed_Maps [6157], page 840,
- <in> Ada.Containers.Hashed_Sets [6267], page 868,
- <in> Ada.Containers.Ordered_Maps [6200], page 847,
- <in> Ada.Containers.Ordered_Sets [6329], page 877,
- <in> Ada.Containers.Vectors [6007], page 781,
- <in> Ada.Strings.Bounded [5222], page 611,
- <in> Ada.Strings.Unbounded [5277], page 625,
- <in> Ada.Text_IO.Editing [7196], page 1074,
- <in> Interfaces.COBOLE [6617], page 934, [6621], page 935, [6625], page 935,

Line <in> Ada.Text_IO [5733], page 702,

Line_Length <in> Ada.Text_IO [5710], page 700,

Log

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7236], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5412], page 649,

Look_Ahead <in> Ada.Text_IO [5741], page 702,

Members

<in> Ada.Execution_Time.Group_Budgets [7026], page 1028,

Merge

<in> Ada.Containers.Doubly_Linked_Lists [6126], page 815,

<in> Ada.Containers.Vectors [6064], page 787,

Microseconds <in> Ada.Real_Time [6946], page 1009,

Milliseconds <in> Ada.Real_Time [6947], page 1009,

Minute <in> Ada.Calendar.Formatting [3791], page 366,

Minutes <in> Ada.Real_Time [6949], page 1009,

Mode

<in> Ada.Direct_IO [5630], page 691,

<in> Ada.Sequential_IO [5605], page 684,

<in> Ada.Streams.Stream_IO [5873], page 747,

<in> Ada.Text_IO [5683], page 699,

Modification_Time <in> Ada.Directories [5948], page 759, [5960], page 761,

Modulus

<in> Ada.Numerics.Generic_Complex_Arrays [7344], page 1131, [7357], page 1133,

<in> Ada.Numerics.Generic_Complex_Types [7221], page 1085,

Month

<in> Ada.Calendar [3749], page 360,

<in> Ada.Calendar.Formatting [3788], page 366,

More_Entries <in> Ada.Directories [5954], page 760,

Move

<in> Ada.Containers.Doubly_Linked_Lists [6094], page 812,

<in> Ada.Containers.Hashed_Maps [6165], page 841,

<in> Ada.Containers.Hashed_Sets [6273], page 868,

<in> Ada.Containers.Ordered_Maps [6208], page 848,

<in> Ada.Containers.Ordered_Sets [6335], page 877,

<in> Ada.Containers.Vectors [6021], page 783,

<in> Ada.Strings.Fixed [5181], page 592,

Name

<in> Ada.Direct_IO [5631], page 691,

<in> Ada.Sequential_IO [5606], page 684,

<in> Ada.Streams.Stream_IO [5874], page 747,

<in> Ada.Text_IO [5684], page 699,

Nanoseconds <in> Ada.Real_Time [6945], page 1009,

New_Char_Array

<in> Interfaces.C.Strings [6538], page 916,

New_Line <in> Ada.Text_IO [5714], page 701,

New_Page <in> Ada.Text_IO [5719], page 701,

New_String <in> Interfaces.C.Strings [6539], page 916,

Next

<in> Ada.Containers.Doubly_Linked_Lists [6113], page 814, [6115], page 814,

- <in> Ada.Containers.Hashtable [6175], page 842, [6176], page 842,
- <in> Ada.Containers.Hashtable_Sets [6292], page 870, [6293], page 871,
- <in> Ada.Containers.Ordered_Containers [6225], page 849, [6226], page 849,
- <in> Ada.Containers.Ordered_Sets [6359], page 880, [6360], page 880,
- <in> Ada.Containers.Vectors [6049], page 786, [6050], page 786,

Null_Task_Id

- <in> Ada.Task_Identification [6748], page 965,

Open

- <in> Ada.Direct_IO [5625], page 691,
- <in> Ada.Sequential_IO [5600], page 684,
- <in> Ada.Streams.Stream_IO [5868], page 746,
- <in> Ada.Text_IO [5678], page 699,

Overlap

- <in> Ada.Containers.Hashtable_Sets [6289], page 870,
- <in> Ada.Containers.Ordered_Sets [6353], page 879,

Overwrite

- <in> Ada.Strings.Bounded [5259], page 618, [5260], page 619,
- <in> Ada.Strings.Fixed [5202], page 595, [5203], page 596,
- <in> Ada.Strings.Unbounded [5312], page 632, [5313], page 632,

Page <in> Ada.Text_IO [5736], page 702,

Page_Length <in> Ada.Text_IO [5711], page 700,

Parent_Tag <in> Ada.Tags [2036], page 138,

Pic_String <in> Ada.Text_IO.Editing [7187], page 1073,

Prepend

- <in> Ada.Containers.Doubly_Linked_Lists [6098], page 812,
- <in> Ada.Containers.Vectors [6030], page 784, [6031], page 784,

Previous

- <in> Ada.Containers.Doubly_Linked_Lists [6114], page 814, [6116], page 814,
- <in> Ada.Containers.Ordered_Containers [6227], page 849, [6228], page 849,
- <in> Ada.Containers.Ordered_Sets [6361], page 880, [6362], page 880,
- <in> Ada.Containers.Vectors [6051], page 786, [6052], page 786,

Put

- <in> Ada.Text_IO [5739], page 702, [5750], page 703, [5762], page 704, [5774], page 705, [5781], page 705, [5784], page 706, [5792], page 706, [5794], page 706, [5801], page 707, [5804], page 707, [5810], page 708, [5813], page 708,
- <in> Ada.Text_IO.Bounded_IO [5831], page 739, [5832], page 739,
- <in> Ada.Text_IO.Complex_IO [7270], page 1098, [7272], page 1098,
- <in> Ada.Text_IO.Editing [7199], page 1074, [7200], page 1074, [7201], page 1075,
- <in> Ada.Text_IO.Unbounded_IO [5840], page 742, [5841], page 742,

Put_Line

- <in> Ada.Text_IO [5756], page 703,
- <in> Ada.Text_IO.Bounded_IO [5833], page 740, [5834], page 740,
- <in> Ada.Text_IO.Unbounded_IO [5842], page 742, [5843], page 742,

Query_Element

- <in> Ada.Containers.Doubly_Linked_Lists [6092], page 812,
- <in> Ada.Containers.Hashed_Maps [6163], page 841,
- <in> Ada.Containers.Hashed_Sets [6272], page 868,
- <in> Ada.Containers.Ordered_Maps [6206], page 847,
- <in> Ada.Containers.Ordered_Sets [6334], page 877,
- <in> Ada.Containers.Vectors [6017], page 782, [6018], page 782,

Raise_Exception <in> Ada.Exceptions [4149], page 424,

Random

- <in> Ada.Numerics.Discrete_Random [5461], page 656,
- <in> Ada.Numerics.Float_Random [5450], page 655,

Re

- <in> Ada.Numerics.Generic_Complex_Arrays [7338], page 1130, [7351], page 1133,
- <in> Ada.Numerics.Generic_Complex_Types [7212], page 1084,

Read

- <in> Ada.Direct_IO [5634], page 692,
- <in> Ada.Sequential_IO [5609], page 684,
- <in> Ada.Storage_IO [5652], page 695,
- <in> Ada.Streams [4766], page 539,
- <in> Ada.Streams.Stream_IO [5879], page 747, [5880], page 747,
- <in> System.RPC [7159], page 1051,

Reference

- <in> Ada.Interrupts [6706], page 958,
- <in> Ada.Task_Attributes [6767], page 968,

Reinitialize <in> Ada.Task_Attributes [6769], page 968,

Remove_Task

- <in> Ada.Execution_Time.Group_Budgets [7023], page 1028,

Rename <in> Ada.Directories [5935], page 758,

Replace

- <in> Ada.Containers.Hashed_Maps [6170], page 842,
- <in> Ada.Containers.Hashed_Sets [6277], page 869, [6304], page 872,
- <in> Ada.Containers.Ordered_Maps [6213], page 848,
- <in> Ada.Containers.Ordered_Sets [6339], page 878, [6374], page 882,

Replace_Element

- <in> Ada.Containers.Doubly_Linked_Lists [6091], page 811,
- <in> Ada.Containers.Hashed_Maps [6162], page 841,
- <in> Ada.Containers.Hashed_Sets [6271], page 868,
- <in> Ada.Containers.Ordered_Maps [6205], page 847,
- <in> Ada.Containers.Ordered_Sets [6333], page 877,
- <in> Ada.Containers.Vectors [6015], page 782, [6016], page 782,
- <in> Ada.Strings.Bounded [5235], page 613,
- <in> Ada.Strings.Unbounded [5288], page 627,

Replace_Slice

- <in> Ada.Strings.Bounded [5255], page 618, [5256], page 618,

- <in> Ada.Strings.Fixed [5198], page 595, [5199], page 595,
- <in> Ada.Strings.Unbounded [5308], page 631, [5309], page 631,

Replenish

- <in> Ada.Execution_Time.Group_Budgets [7027], page 1028,

Replicate <in> Ada.Strings.Bounded [5271], page 620, [5272], page 621, [5273], page 621,

Reraise_Occurrence <in> Ada.Exceptions [4151], page 424,

Reserve_Capacity

- <in> Ada.Containers.Hashtable [6156], page 840,
- <in> Ada.Containers.Hashtable [6266], page 868,
- <in> Ada.Containers.Vectors [6006], page 781,

Reset

- <in> Ada.Direct_IO [5629], page 691,
- <in> Ada.Numerics.Discrete_Random [5462], page 656, [5466], page 656,
- <in> Ada.Numerics.Float_Random [5452], page 655, [5455], page 655,
- <in> Ada.Sequential_IO [5603], page 684,
- <in> Ada.Streams.Stream_IO [5872], page 747,
- <in> Ada.Text_IO [5682], page 699,

Reverse_Elements

- <in> Ada.Containers.Doubly_Linked_Lists [6103], page 813,
- <in> Ada.Containers.Vectors [6040], page 785,

Reverse_Find

- <in> Ada.Containers.Doubly_Linked_Lists [6118], page 814,
- <in> Ada.Containers.Vectors [6056], page 786,

Reverse_Find_Index

- <in> Ada.Containers.Vectors [6055], page 786,

Reverse_Iterate

- <in> Ada.Containers.Doubly_Linked_Lists [6122], page 815,
- <in> Ada.Containers.Ordered_Maps [6236], page 851,
- <in> Ada.Containers.Ordered_Sets [6369], page 881,
- <in> Ada.Containers.Vectors [6060], page 787,

Save

- <in> Ada.Numerics.Discrete_Random [5465], page 656,
- <in> Ada.Numerics.Float_Random [5454], page 655,

Save_Occurrence <in> Ada.Exceptions [4158], page 424,

Second <in> Ada.Calendar.Formatting [3792], page 367,

Seconds

- <in> Ada.Calendar [3751], page 360,
- <in> Ada.Real_Time [6948], page 1009,

Seconds_Of <in> Ada.Calendar.Formatting [3794], page 367,

Set <in> Ada.Environment_Variables [5979], page 775,

Set_Bounded_String

- <in> Ada.Strings.Bounded [5225], page 612,

Set_Col <in> Ada.Text_IO [5728], page 701,

Set_Deadline <in> Ada.Dispatching.EDF [6855], page 988,
Set_Dependents_Fallback_Handler
 <in> Ada.Task_Termination [6779], page 972,
Set_Directory <in> Ada.Directories [5929], page 758,
Set_Error <in> Ada.Text_IO [5689], page 699,
Set_Exit_Status <in> Ada.Command_Line [5925], page 755,
Set_False
 <in> Ada.Synchronous_Task_Control [6966], page 1015,
Set_Handler
 <in> Ada.Execution_Time.Group_Budgets [7031], page 1028,
 <in> Ada.Execution_Time.Timers [7007], page 1025,
 <in> Ada.Real_Time.Timing_Events [7044], page 1031,
Set_Im
 <in> Ada.Numerics.Generic_Complex_Arrays [7341], page 1131, [7354], page 1133,
 <in> Ada.Numerics.Generic_Complex_Types [7216], page 1085,
Set_Index
 <in> Ada.Direct_IO [5638], page 692,
 <in> Ada.Streams.Stream_IO [5883], page 748,
Set_Input <in> Ada.Text_IO [5687], page 699,
Set_Length <in> Ada.Containers.Vectors [6008], page 781,
Set_Line <in> Ada.Text_IO [5730], page 701,
Set_Line_Length <in> Ada.Text_IO [5705], page 700,
Set_Mode <in> Ada.Streams.Stream_IO [5886], page 748,
Set_Output <in> Ada.Text_IO [5688], page 699,
Set_Page_Length <in> Ada.Text_IO [5708], page 700,
Set_Priority
 <in> Ada.Dynamic_Priorities [6889], page 996,
Set_Quantum
 <in> Ada.Dispatching.Round_Robin [6845], page 986,
Set_Re
 <in> Ada.Numerics.Generic_Complex_Arrays [7340], page 1131, [7353], page 1133,
 <in> Ada.Numerics.Generic_Complex_Types [7215], page 1085,
Set_Specific_Handler
 <in> Ada.Task_Termination [6781], page 972,
Set_True
 <in> Ada.Synchronous_Task_Control [6965], page 1015,
Set_Unbounded_String
 <in> Ada.Strings.Unbounded [5283], page 626,
Set_Value <in> Ada.Task_Attributes [6768], page 968,
Simple_Name <in> Ada.Directories [5938], page 758, [5956], page 760,

Sin

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7239], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5416], page 649,

Sinh

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7247], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5431], page 650,

Size

- <in> Ada.Direct_IO [5640], page 692,
- <in> Ada.Directories [5947], page 759, [5959], page 761,
- <in> Ada.Streams.Stream_IO [5885], page 748,

Skip_Line <in> Ada.Text_IO [5715], page 701,

Skip_Page <in> Ada.Text_IO [5721], page 701,

Slice

- <in> Ada.Strings.Bounded [5236], page 613,
- <in> Ada.Strings.Unbounded [5289], page 627,

Solve

- <in> Ada.Numerics.Generic_Complex_Arrays [7364], page 1135,
- <in> Ada.Numerics.Generic_Real_Arrays [7324], page 1121,

Sort

- <in> Ada.Containers.Doubly_Linked_Lists [6125], page 815,
- <in> Ada.Containers.Vectors [6063], page 787,

Specific_Handler

- <in> Ada.Task_Termination [6782], page 972,

Splice

<in> Ada.Containers.Doubly_Linked_Lists [6106], page 813, [6107], page 813, [6108], page 813,

Split

- <in> Ada.Calendar [3752], page 360,
- <in> Ada.Calendar.Formatting [3795], page 367, [3798], page 368, [3799], page 368, [3800], page 368,
- <in> Ada.Execution_Time [6996], page 1022,
- <in> Ada.Real_Time [6951], page 1009,

Sqrt

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7235], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5411], page 649,

Standard_Error <in> Ada.Text_IO [5692], page 699, [5699], page 700,

Standard_Input <in> Ada.Text_IO [5690], page 699, [5697], page 700,

Standard_Output <in> Ada.Text_IO [5691], page 699, [5698], page 700,

Start_Search <in> Ada.Directories [5952], page 760,

Storage_Size <in> System.Storage_Pools [4689], page 527,

Stream

- <in> Ada.Streams.Stream_IO [5878], page 747,
- <in> Ada.Text_IO.Text_Streams [5898], page 751,

- <in> Ada.Wide_Text_IO.Text_Streams [5901], page 751,
- <in> Ada.Wide_Wide_Text_IO.Text_Streams [5904], page 752,
- Strlen <in> Interfaces.C.Strings [6546], page 917,
- Sub_Second <in> Ada.Calendar.Formatting [3793], page 367,
- Suspend_Until_True
 - <in> Ada.Synchronous_Task_Control [6968], page 1015,
- Swap
 - <in> Ada.Containers.Doubly_Linked_Lists [6104], page 813,
 - <in> Ada.Containers.Vectors [6041], page 785, [6042], page 785,
- Swap_Links
 - <in> Ada.Containers.Doubly_Linked_Lists [6105], page 813,
- Symmetric_Difference
 - <in> Ada.Containers.Hashed_Sets [6287], page 870, [6288], page 870,
 - <in> Ada.Containers.Ordered_Sets [6351], page 879, [6352], page 879,
- Tail
 - <in> Ada.Strings.Bounded [5269], page 620, [5270], page 620,
 - <in> Ada.Strings.Fixed [5212], page 597, [5213], page 597,
 - <in> Ada.Strings.Unbounded [5322], page 633, [5323], page 633,
- Tan
 - <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7241], page 1092,
 - <in> Ada.Numerics.Generic_Elementary_Functions [5420], page 649,
- Tanh
 - <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7249], page 1092,
 - <in> Ada.Numerics.Generic_Elementary_Functions [5433], page 650,
- Time_Of
 - <in> Ada.Calendar [3753], page 360,
 - <in> Ada.Calendar.Formatting [3796], page 367, [3797], page 367,
 - <in> Ada.Execution_Time [6997], page 1022,
 - <in> Ada.Real_Time [6952], page 1009,
- Time_Of_Event
 - <in> Ada.Real_Time.Timing_Events [7048], page 1031,
- Time_Remaining
 - <in> Ada.Execution_Time.Timers [7010], page 1025,
- To_Ada
 - <in> Interfaces.C [6492], page 903, [6496], page 904, [6498], page 904, [6502], page 904, [6506], page 905, [6508], page 905, [6518], page 906, [6522], page 906, [6526], page 907, [6528], page 907, [6512], page 905, [6516], page 906,
 - <in> Interfaces.COBOl [6595], page 933, [6597], page 933,
 - <in> Interfaces.Fortran [6648], page 946, [6650], page 946, [6652], page 946,
- To_Address
 - <in> System.Address_To_Access_Conversions [4645], page 518,
 - <in> System.Storage_Elements [4638], page 517,
- To_Basic <in> Ada.Characters.Handling [4874], page 567, [4877], page 567,

To_Binary <in> Interfaces.COBOL [6627], page 935, [6630], page 935,

To_Bounded_String
<in> Ada.Strings.Bounded [5223], page 611,

To_C <in> Interfaces.C [6491], page 903, [6495], page 904, [6497], page 904, [6501], page 904, [6505], page 905, [6507], page 905, [6521], page 906, [6525], page 907, [6527], page 907, [6511], page 905, [6515], page 906, [6517], page 906,

To_Character
<in> Ada.Characters.Conversions [5137], page 580,

To_Chars_Ptr <in> Interfaces.C.Strings [6537], page 916,

To_COBOL <in> Interfaces.COBOL [6594], page 933, [6596], page 933,

To_Cursor <in> Ada.Containers.Vectors [6011], page 782,

To_Decimal <in> Interfaces.COBOL [6618], page 934, [6622], page 935, [6626], page 935, [6629], page 935,

To_Display <in> Interfaces.COBOL [6619], page 934,

To_Domain
<in> Ada.Strings.Maps [5174], page 587,
<in> Ada.Strings.Wide_Maps [5365], page 639,
<in> Ada.Strings.Wide_Wide_Maps [5396], page 644,

To_Duration <in> Ada.Real_Time [6943], page 1009,

To_Fortran <in> Interfaces.Fortran [6647], page 946, [6649], page 946, [6651], page 946,

To_Index <in> Ada.Containers.Vectors [6012], page 782,

To_Integer <in> System.Storage_Elements [4639], page 517,

To_ISO_646 <in> Ada.Characters.Handling [4881], page 567, [4882], page 567,

To_Long_Binary <in> Interfaces.COBOL [6631], page 935,

To_Lower <in> Ada.Characters.Handling [4872], page 567, [4875], page 567,

To_Mapping
<in> Ada.Strings.Maps [5173], page 587,
<in> Ada.Strings.Wide_Maps [5364], page 639,
<in> Ada.Strings.Wide_Wide_Maps [5395], page 644,

To_Packed <in> Interfaces.COBOL [6623], page 935,

To_Picture <in> Ada.Text_IO.Editing [7186], page 1073,

To_Pointer
<in> System.Address_To_Access_Conversions [4644], page 518,

To_Range
<in> Ada.Strings.Maps [5175], page 587,
<in> Ada.Strings.Wide_Maps [5366], page 639,
<in> Ada.Strings.Wide_Wide_Maps [5397], page 644,

To_Ranges
<in> Ada.Strings.Maps [5163], page 586,
<in> Ada.Strings.Wide_Maps [5354], page 637,
<in> Ada.Strings.Wide_Wide_Maps [5385], page 642,

To_Sequence

- <in> Ada.Strings.Maps [5169], page 586,
- <in> Ada.Strings.Wide_Maps [5360], page 638,
- <in> Ada.Strings.Wide_Wide_Maps [5391], page 643,

To_Set

- <in> Ada.Containers.Hashed_Sets [6264], page 868,
- <in> Ada.Containers.Ordered_Sets [6328], page 877,
- <in> Ada.Strings.Maps [5161], page 585, [5162], page 585, [5167], page 586, [5168], page 586,
- <in> Ada.Strings.Wide_Maps [5352], page 637, [5353], page 637, [5358], page 638, [5359], page 638,
- <in> Ada.Strings.Wide_Wide_Maps [5383], page 642, [5384], page 642, [5389], page 643, [5390], page 643,

To_String

- <in> Ada.Characters.Conversions [5140], page 580,
- <in> Ada.Strings.Bounded [5224], page 611,
- <in> Ada.Strings.Unbounded [5282], page 626,

To_Time_Span <in> Ada.Real_Time [6944], page 1009,

To_Unbounded_String

- <in> Ada.Strings.Unbounded [5280], page 625, [5281], page 626,

To_Upper <in> Ada.Characters.Handling [4873], page 567, [4876], page 567,

To_Vector <in> Ada.Containers.Vectors [6003], page 781, [6004], page 781,

To_Wide_Character

- <in> Ada.Characters.Conversions [5131], page 580, [5141], page 581,

To_Wide_String

- <in> Ada.Characters.Conversions [5132], page 580, [5142], page 581,

To_Wide_Wide_Character

- <in> Ada.Characters.Conversions [5135], page 580,

To_Wide_Wide_String

- <in> Ada.Characters.Conversions [5134], page 580,

Translate

- <in> Ada.Strings.Bounded [5251], page 617, [5252], page 617, [5253], page 617, [5254], page 618,
- <in> Ada.Strings.Fixed [5194], page 594, [5195], page 594, [5196], page 594, [5197], page 595,
- <in> Ada.Strings.Unbounded [5304], page 631, [5305], page 631, [5306], page 631, [5307], page 631,

Transpose

- <in> Ada.Numerics.Generic_Complex_Arrays [7363], page 1134,
- <in> Ada.Numerics.Generic_Real_Arrays [7323], page 1121,

Trim

- <in> Ada.Strings.Bounded [5264], page 619, [5265], page 619, [5266], page 619,
- <in> Ada.Strings.Fixed [5206], page 596, [5207], page 596, [5208], page 596, [5209], page 596,

<in> Ada.Strings.Unbounded [5316], page 632, [5317], page 632, [5318], page 632, [5319], page 633,

Unbounded_Slice

<in> Ada.Strings.Unbounded [5290], page 627, [5291], page 627,

Unchecked_Conversion

<child of> Ada [4658], page 520,

Unchecked_Deallocation

<child of> Ada [4719], page 532,

Union

<in> Ada.Containers.Hashed_Sets [6281], page 869, [6282], page 869,

<in> Ada.Containers.Ordered_Sets [6345], page 878, [6346], page 878,

Unit_Matrix

<in> Ada.Numerics.Generic_Complex_Arrays [7370], page 1136,

<in> Ada.Numerics.Generic_Real_Arrays [7330], page 1122,

Unit_Vector

<in> Ada.Numerics.Generic_Complex_Arrays [7350], page 1133,

<in> Ada.Numerics.Generic_Real_Arrays [7322], page 1120,

Update <in> Interfaces.C.Strings [6547], page 917, [6548], page 917,

Update_Element

<in> Ada.Containers.Doubly_Linked_Lists [6093], page 812,

<in> Ada.Containers.Hashed_Maps [6164], page 841,

<in> Ada.Containers.Ordered_Maps [6207], page 847,

<in> Ada.Containers.Vectors [6019], page 782, [6020], page 783,

Update_Element_Preserving_Key

<in> Ada.Containers.Hashed_Sets [6309], page 872,

<in> Ada.Containers.Ordered_Sets [6381], page 883,

Update_Error <in> Interfaces.C.Strings [6549], page 917,

UTC_Time_Offset

<in> Ada.Calendar.Time_Zones [3768], page 364,

Valid

<in> Ada.Text_IO.Editing [7185], page 1073, [7197], page 1074,

<in> Interfaces.COBOLE [6616], page 934, [6620], page 935, [6624], page 935,

Value

<in> Ada.Calendar.Formatting [3802], page 369, [3804], page 369,

<in> Ada.Environment_Variables [5977], page 775,

<in> Ada.Numerics.Discrete_Random [5469], page 656,

<in> Ada.Numerics.Float_Random [5458], page 655,

<in> Ada.Strings.Maps [5171], page 587,

<in> Ada.Strings.Wide_Maps [5362], page 639,

<in> Ada.Strings.Wide_Wide_Maps [5393], page 644,

<in> Ada.Task_Attributes [6766], page 968,

<in> Interfaces.C.Pointers [6556], page 923, [6557], page 923,

<in> Interfaces.C.Strings [6542], page 916, [6543], page 916, [6544], page 917, [6545], page 917,

Virtual_Length

<in> Interfaces.C.Pointers [6561], page 924,

Wide_Hash

<child of> Ada.Strings.Wide_Bounded [5344], page 636,

<child of> Ada.Strings.Wide_Fixed [5343], page 636,

<child of> Ada.Strings.Wide_Unbounded [5345], page 636,

Wide_Exception_Name <in> Ada.Exceptions [4144], page 423, [4154], page 424,

Wide_Expanded_Name <in> Ada.Tags [2030], page 138,

Wide_Wide_Hash

<child of> Ada.Strings.Wide_Wide_Bounded [5375], page 641,

<child of> Ada.Strings.Wide_Wide_Fixed [5374], page 641,

<child of> Ada.Strings.Wide_Wide_Unbounded [5376], page 641,

Wide_Wide_Exception_Name

<in> Ada.Exceptions [4145], page 423, [4155], page 424,

Wide_Wide_Expanded_Name <in> Ada.Tags [2031], page 138,

Write

<in> Ada.Direct_IO [5636], page 692,

<in> Ada.Sequential_IO [5610], page 684,

<in> Ada.Storage_IO [5653], page 696,

<in> Ada.Streams [4767], page 539,

<in> Ada.Streams.Stream_IO [5881], page 748, [5882], page 748,

<in> System.RPC [7160], page 1051,

Year

<in> Ada.Calendar [3748], page 360,

<in> Ada.Calendar.Formatting [3787], page 366,

29.4 Q.4 Language-Defined Exceptions

1/2

This clause lists all language–defined exceptions.

Argument_Error

<in> Ada.Numerics [5407], page 648,

Communication_Error

<in> System.RPC [7157], page 1050,

Constraint_Error

<in> Standard [4851], page 563,

Conversion_Error

<in> Interfaces.COBOLE [6614], page 934,

Data_Error

<in> Ada.Direct_IO [5648], page 692,

<in> Ada.IO.Exceptions [5912], page 752,

<in> Ada.Sequential_IO [5618], page 684,

<in> Ada.Storage_IO [5654], page 696,

- <in> Ada.Streams.Stream_IO [5894], page 748,
- <in> Ada.Text_IO [5820], page 708,

Device_Error

- <in> Ada.Direct_IO [5646], page 692,
- <in> Ada.Directories [5964], page 761,
- <in> Ada.IO_Exceptions [5910], page 752,
- <in> Ada.Sequential_IO [5616], page 684,
- <in> Ada.Streams.Stream_IO [5892], page 748,
- <in> Ada.Text_IO [5818], page 708,

Dispatching_Policy_Error

- <in> Ada.Dispatching [6810], page 978,

End_Error

- <in> Ada.Direct_IO [5647], page 692,
- <in> Ada.IO_Exceptions [5911], page 752,
- <in> Ada.Sequential_IO [5617], page 684,
- <in> Ada.Streams.Stream_IO [5893], page 748,
- <in> Ada.Text_IO [5819], page 708,

Group_Budget_Error

- <in> Ada.Execution_Time.Group_Budgets [7034], page 1028,

Index_Error

- <in> Ada.Strings [5149], page 584,

Layout_Error

- <in> Ada.IO_Exceptions [5913], page 752,
- <in> Ada.Text_IO [5821], page 708,

Length_Error

- <in> Ada.Strings [5147], page 584,

Mode_Error

- <in> Ada.Direct_IO [5643], page 692,
- <in> Ada.IO_Exceptions [5907], page 752,
- <in> Ada.Sequential_IO [5613], page 684,
- <in> Ada.Streams.Stream_IO [5889], page 748,
- <in> Ada.Text_IO [5815], page 708,

Name_Error

- <in> Ada.Direct_IO [5644], page 692,
- <in> Ada.Directories [5962], page 761,
- <in> Ada.IO_Exceptions [5908], page 752,
- <in> Ada.Sequential_IO [5614], page 684,
- <in> Ada.Streams.Stream_IO [5890], page 748,
- <in> Ada.Text_IO [5816], page 708,

Pattern_Error

- <in> Ada.Strings [5148], page 584,

Picture_Error

- <in> Ada.Text_IO.Editing [7190], page 1073,

Pointer_Error

- <in> Interfaces.C.Pointers [6558], page 923,

Program_Error

<in> Standard [4852], page 563,

Status_Error

<in> Ada.Direct_IO [5642], page 692,
<in> Ada.Directories [5961], page 761,
<in> Ada.IO_Exceptions [5906], page 752,
<in> Ada.Sequential_IO [5612], page 684,
<in> Ada.Streams.Stream_IO [5888], page 748,
<in> Ada.Text_IO [5814], page 708,

Storage_Error

<in> Standard [4853], page 563,

Tag_Error

<in> Ada.Tags [2039], page 138,

Tasking_Error

<in> Standard [4854], page 563,

Terminator_Error

<in> Interfaces.C [6529], page 907,

Time_Error

<in> Ada.Calendar [3754], page 361,

Timer_Resource_Error

<in> Ada.Execution_Time.Timers [7011], page 1025,

Translation_Error

<in> Ada.Strings [5150], page 584,

Unknown_Zone_Error

<in> Ada.Calendar.Time_Zones [3767], page 364,

Use_Error

<in> Ada.Direct_IO [5645], page 692,
<in> Ada.Directories [5963], page 761,
<in> Ada.IO_Exceptions [5909], page 752,
<in> Ada.Sequential_IO [5615], page 684,
<in> Ada.Streams.Stream_IO [5891], page 748,
<in> Ada.Text_IO [5817], page 708,

29.5 Q.5 Language-Defined Objects

1/2

This clause lists all language–defined constants, variables, named numbers, and enumeration literals.

ACK <in> Ada.Characters.Latin_1 [4901], page 573,

Acute <in> Ada.Characters.Latin_1 [5047], page 577,

Ada_To_COBOL <in> Interfaces.COBOLE [6591], page 932,

Alphanumeric_Set

<in> Ada.Strings.Maps.Constants [5333], page 636,

Ampersand <in> Ada.Characters.Latin_1 [4933], page 574,
APC <in> Ada.Characters.Latin_1 [5024], page 577,
Apostrophe <in> Ada.Characters.Latin_1 [4934], page 574,
Asterisk <in> Ada.Characters.Latin_1 [4937], page 574,
Basic_Map
 <in> Ada.Strings.Maps.Constants [5338], page 636,
Basic_Set
 <in> Ada.Strings.Maps.Constants [5330], page 636,
BEL <in> Ada.Characters.Latin_1 [4902], page 573,
BPH <in> Ada.Characters.Latin_1 [4995], page 576,
Broken_Bar <in> Ada.Characters.Latin_1 [5032], page 577,
BS <in> Ada.Characters.Latin_1 [4903], page 573,
Buffer_Size <in> Ada.Storage_IO [5650], page 695,
CAN <in> Ada.Characters.Latin_1 [4919], page 574,
CCH <in> Ada.Characters.Latin_1 [5013], page 576,
Cedilla <in> Ada.Characters.Latin_1 [5052], page 578,
Cent_Sign <in> Ada.Characters.Latin_1 [5028], page 577,
char16_nul <in> Interfaces.C [6510], page 905,
char32_nul <in> Interfaces.C [6520], page 906,
CHAR_BIT <in> Interfaces.C [6471], page 902,
Character_Set
 <in> Ada.Strings.Wide_Maps [5368], page 641,
 <in> Ada.Strings.Wide_Maps.Wide_Constants [5400], page 646,
Circumflex <in> Ada.Characters.Latin_1 [4954], page 575,
COBOL_To_Ada <in> Interfaces.COBOL [6592], page 932,
Colon <in> Ada.Characters.Latin_1 [4944], page 575,
Comma <in> Ada.Characters.Latin_1 [4939], page 574,
Commercial_At
 <in> Ada.Characters.Latin_1 [4950], page 575,
Control_Set
 <in> Ada.Strings.Maps.Constants [5325], page 636,
Copyright_Sign
 <in> Ada.Characters.Latin_1 [5035], page 577,
CPU_Tick <in> Ada.Execution_Time [6994], page 1022,
CPU_Time_First <in> Ada.Execution_Time [6991], page 1022,
CPU_Time_Last <in> Ada.Execution_Time [6992], page 1022,
CPU_Time_Unit <in> Ada.Execution_Time [6993], page 1022,
CR <in> Ada.Characters.Latin_1 [4908], page 573,
CSI <in> Ada.Characters.Latin_1 [5020], page 577,

Currency_Sign

<in> Ada.Characters.Latin_1 [5030], page 577,

DC1 <in> Ada.Characters.Latin_1 [4912], page 574,

DC2 <in> Ada.Characters.Latin_1 [4913], page 574,

DC3 <in> Ada.Characters.Latin_1 [4914], page 574,

DC4 <in> Ada.Characters.Latin_1 [4915], page 574,

DCS <in> Ada.Characters.Latin_1 [5009], page 576,

Decimal_Digit_Set

<in> Ada.Strings.Maps.Constants [5331], page 636,

Default_Aft

<in> Ada.Text_IO [5777], page 705, [5787], page 706, [5797], page 707,

<in> Ada.Text_IO.Complex_IO [7265], page 1098,

Default_Base <in> Ada.Text_IO [5759], page 703, [5768], page 704,

Default_Bit_Order <in> System [4625], page 512,

Default_Currency

<in> Ada.Text_IO.Editing [7191], page 1073,

Default_Deadline

<in> Ada.Dispatching.EDF [6854], page 988,

Default_Exp

<in> Ada.Text_IO [5778], page 705, [5788], page 706, [5798], page 707,

<in> Ada.Text_IO.Complex_IO [7266], page 1098,

Default_Fill <in> Ada.Text_IO.Editing [7192], page 1073,

Default_Fore

<in> Ada.Text_IO [5776], page 705, [5786], page 706, [5796], page 707,

<in> Ada.Text_IO.Complex_IO [7264], page 1098,

Default_Priority <in> System [4629], page 512,

Default_Quantum

<in> Ada.Dispatching.Round_Robin [6844], page 986,

Default_Radix_Mark

<in> Ada.Text_IO.Editing [7194], page 1073,

Default_Separator

<in> Ada.Text_IO.Editing [7193], page 1073,

Default_Setting <in> Ada.Text_IO [5807], page 708,

Default_Width <in> Ada.Text_IO [5758], page 703, [5767], page 704, [5806], page 708,

Degree_Sign <in> Ada.Characters.Latin_1 [5042], page 577,

DEL <in> Ada.Characters.Latin_1 [4987], page 576,

Diaeresis <in> Ada.Characters.Latin_1 [5034], page 577,

Division_Sign

<in> Ada.Characters.Latin_1 [5115], page 579,

DLE <in> Ada.Characters.Latin_1 [4911], page 574,

Dollar_Sign <in> Ada.Characters.Latin_1 [4931], page 574,

e <in> Ada.Numerics [5409], page 648,
EM <in> Ada.Characters.Latin_1 [4920], page 574,
Empty_List
 <in> Ada.Containers.Doubly_Linked_Lists [6085], page 811,
Empty_Map
 <in> Ada.Containers.Hashtable [6153], page 840,
 <in> Ada.Containers.Ordered_Maps [6198], page 847,
Empty_Set
 <in> Ada.Containers.Hashtable [6261], page 867,
 <in> Ada.Containers.Ordered_Sets [6325], page 876,
Empty_Vector
 <in> Ada.Containers.Vectors [6001], page 780,
ENQ <in> Ada.Characters.Latin_1 [4900], page 573,
EOT <in> Ada.Characters.Latin_1 [4899], page 573,
EPA <in> Ada.Characters.Latin_1 [5016], page 576,
Equals_Sign <in> Ada.Characters.Latin_1 [4947], page 575,
ESA <in> Ada.Characters.Latin_1 [5000], page 576,
ESC <in> Ada.Characters.Latin_1 [4922], page 574,
ETB <in> Ada.Characters.Latin_1 [4918], page 574,
ETX <in> Ada.Characters.Latin_1 [4898], page 573,
Exclamation <in> Ada.Characters.Latin_1 [4928], page 574,
Failure <in> Ada.Command_Line [5924], page 755,
Feminine_Ordinal_Indicator
 <in> Ada.Characters.Latin_1 [5036], page 577,
FF <in> Ada.Characters.Latin_1 [4907], page 573,
Fine_Delta <in> System [4614], page 511,
Fraction_One_Half
 <in> Ada.Characters.Latin_1 [5057], page 578,
Fraction_One_Quarter
 <in> Ada.Characters.Latin_1 [5056], page 578,
Fraction_Three_Quarters
 <in> Ada.Characters.Latin_1 [5058], page 578,
Friday <in> Ada.Calendar.Formatting [3779], page 365,
FS <in> Ada.Characters.Latin_1 [4923], page 574,
Full_Stop <in> Ada.Characters.Latin_1 [4942], page 574,
Graphic_Set
 <in> Ada.Strings.Maps.Constants [5326], page 636,
Grave <in> Ada.Characters.Latin_1 [4956], page 575,
Greater_Than_Sign
 <in> Ada.Characters.Latin_1 [4948], page 575,
GS <in> Ada.Characters.Latin_1 [4924], page 574,

Hexadecimal_Digit_Set

<in> Ada.Strings.Maps.Constants [5332], page 636,

High_Order_First

<in> Interfaces.COBOLE [6606], page 933,

<in> System [4623], page 512,

HT <in> Ada.Characters.Latin_1 [4904], page 573,

HTJ <in> Ada.Characters.Latin_1 [5002], page 576,

HTS <in> Ada.Characters.Latin_1 [5001], page 576,

Hyphen <in> Ada.Characters.Latin_1 [4940], page 574,

i

<in> Ada.Numerics.Generic_Complex_Types [7210], page 1084,

<in> Interfaces.Fortran [6643], page 946,

Identity

<in> Ada.Strings.Maps [5172], page 587,

<in> Ada.Strings.Wide_Maps [5363], page 639,

<in> Ada.Strings.Wide_Wide_Maps [5394], page 644,

Inverted_Exclamation

<in> Ada.Characters.Latin_1 [5027], page 577,

Inverted_Question

<in> Ada.Characters.Latin_1 [5059], page 578,

IS1 <in> Ada.Characters.Latin_1 [4992], page 576,

IS2 <in> Ada.Characters.Latin_1 [4991], page 576,

IS3 <in> Ada.Characters.Latin_1 [4990], page 576,

IS4 <in> Ada.Characters.Latin_1 [4989], page 576,

ISO_646_Set

<in> Ada.Strings.Maps.Constants [5335], page 636,

j

<in> Ada.Numerics.Generic_Complex_Types [7211], page 1084,

<in> Interfaces.Fortran [6644], page 946,

LC_A <in> Ada.Characters.Latin_1 [4957], page 575,

LC_A_Acute <in> Ada.Characters.Latin_1 [5093], page 579,

LC_A_Circumflex

<in> Ada.Characters.Latin_1 [5094], page 579,

LC_A_Diaeresis

<in> Ada.Characters.Latin_1 [5096], page 579,

LC_A_Grave <in> Ada.Characters.Latin_1 [5092], page 579,

LC_A_Ring <in> Ada.Characters.Latin_1 [5097], page 579,

LC_A_Tilde <in> Ada.Characters.Latin_1 [5095], page 579,

LC_AE_Diphthong

<in> Ada.Characters.Latin_1 [5098], page 579,

LC_B <in> Ada.Characters.Latin_1 [4958], page 575,

LC_C <in> Ada.Characters.Latin_1 [4959], page 575,
LC_C_Cedilla
 <in> Ada.Characters.Latin_1 [5099], page 579,
LC_D <in> Ada.Characters.Latin_1 [4960], page 575,
LC_E <in> Ada.Characters.Latin_1 [4961], page 575,
LC_E_Acute <in> Ada.Characters.Latin_1 [5101], page 579,
LC_E_Circumflex
 <in> Ada.Characters.Latin_1 [5102], page 579,
LC_E_Diaeresis
 <in> Ada.Characters.Latin_1 [5103], page 579,
LC_E_Grave <in> Ada.Characters.Latin_1 [5100], page 579,
LC_F <in> Ada.Characters.Latin_1 [4962], page 575,
LC_G <in> Ada.Characters.Latin_1 [4963], page 575,
LC_German_Sharp_S
 <in> Ada.Characters.Latin_1 [5091], page 579,
LC_H <in> Ada.Characters.Latin_1 [4964], page 575,
LC_I <in> Ada.Characters.Latin_1 [4965], page 575,
LC_I_Acute <in> Ada.Characters.Latin_1 [5105], page 579,
LC_I_Circumflex
 <in> Ada.Characters.Latin_1 [5106], page 579,
LC_I_Diaeresis
 <in> Ada.Characters.Latin_1 [5107], page 579,
LC_I_Grave <in> Ada.Characters.Latin_1 [5104], page 579,
LC_Icelandic_Eth
 <in> Ada.Characters.Latin_1 [5108], page 579,
LC_Icelandic_Thorn
 <in> Ada.Characters.Latin_1 [5122], page 579,
LC_J <in> Ada.Characters.Latin_1 [4966], page 575,
LC_K <in> Ada.Characters.Latin_1 [4967], page 575,
LC_L <in> Ada.Characters.Latin_1 [4968], page 575,
LC_M <in> Ada.Characters.Latin_1 [4969], page 575,
LC_N <in> Ada.Characters.Latin_1 [4970], page 575,
LC_N_Tilde <in> Ada.Characters.Latin_1 [5109], page 579,
LC_O <in> Ada.Characters.Latin_1 [4971], page 575,
LC_O_Acute <in> Ada.Characters.Latin_1 [5111], page 579,
LC_O_Circumflex
 <in> Ada.Characters.Latin_1 [5112], page 579,
LC_O_Diaeresis
 <in> Ada.Characters.Latin_1 [5114], page 579,
LC_O_Grave <in> Ada.Characters.Latin_1 [5110], page 579,

LC_O_Oblique_Stroke
<in> Ada.Characters.Latin_1 [5116], page 579,
LC_O_Tilde <in> Ada.Characters.Latin_1 [5113], page 579,
LC_P <in> Ada.Characters.Latin_1 [4972], page 575,
LC_Q <in> Ada.Characters.Latin_1 [4973], page 575,
LC_R <in> Ada.Characters.Latin_1 [4974], page 575,
LC_S <in> Ada.Characters.Latin_1 [4975], page 575,
LC_T <in> Ada.Characters.Latin_1 [4976], page 575,
LC_U <in> Ada.Characters.Latin_1 [4977], page 575,
LC_U_Acute <in> Ada.Characters.Latin_1 [5118], page 579,
LC_U_Circumflex
<in> Ada.Characters.Latin_1 [5119], page 579,
LC_U_Diaeresis
<in> Ada.Characters.Latin_1 [5120], page 579,
LC_U_Grave <in> Ada.Characters.Latin_1 [5117], page 579,
LC_V <in> Ada.Characters.Latin_1 [4978], page 575,
LC_W <in> Ada.Characters.Latin_1 [4979], page 575,
LC_X <in> Ada.Characters.Latin_1 [4980], page 576,
LC_Y <in> Ada.Characters.Latin_1 [4981], page 576,
LC_Y_Acute <in> Ada.Characters.Latin_1 [5121], page 579,
LC_Y_Diaeresis
<in> Ada.Characters.Latin_1 [5123], page 579,
LC_Z <in> Ada.Characters.Latin_1 [4982], page 576,
Leading_Nonseparate
<in> Interfaces.COBOL [6603], page 933,
Leading_Separate <in> Interfaces.COBOL [6601], page 933,
Left_Angle_Quotation
<in> Ada.Characters.Latin_1 [5037], page 577,
Left_Curly_Bracket
<in> Ada.Characters.Latin_1 [4983], page 576,
Left_Parenthesis
<in> Ada.Characters.Latin_1 [4935], page 574,
Left_Square_Bracket
<in> Ada.Characters.Latin_1 [4951], page 575,
Less_Than_Sign
<in> Ada.Characters.Latin_1 [4946], page 575,
Letter_Set
<in> Ada.Strings.Maps.Constants [5327], page 636,
LF <in> Ada.Characters.Latin_1 [4905], page 573,
Low_Line <in> Ada.Characters.Latin_1 [4955], page 575,

Low_Order_First
 <in> Interfaces.COBOL [6607], page 933,
 <in> System [4624], page 512,

Lower_Case_Map
 <in> Ada.Strings.Maps.Constants [5336], page 636,

Lower_Set
 <in> Ada.Strings.Maps.Constants [5328], page 636,

Macron <in> Ada.Characters.Latin_1 [5041], page 577,

Masculine_Ordinal_Indicator
 <in> Ada.Characters.Latin_1 [5054], page 578,

Max_Base_Digits <in> System [4611], page 511,

Max_Binary_Modulus <in> System [4609], page 511,

Max_Decimal_Digits <in> Ada.Decimal [7177], page 1055,

Max_Delta <in> Ada.Decimal [7176], page 1055,

Max_Digits <in> System [4612], page 511,

Max_Digits_Binary <in> Interfaces.COBOL [6586], page 932,

Max_Digits_Long_Binary
 <in> Interfaces.COBOL [6587], page 932,

Max_Image_Width
 <in> Ada.Numerics.Discrete_Random [5467], page 656,
 <in> Ada.Numerics.Float_Random [5456], page 655,

Max_Int <in> System [4608], page 511,

Max_Length <in> Ada.Strings.Bounded [5218], page 611,

Max_Mantissa <in> System [4613], page 511,

Max_Nonbinary_Modulus <in> System [4610], page 511,

Max_Picture_Length
 <in> Ada.Text_IO.Editing [7189], page 1073,

Max_Scale <in> Ada.Decimal [7173], page 1055,

Memory_Size <in> System [4620], page 511,

Micro_Sign <in> Ada.Characters.Latin_1 [5048], page 577,

Middle_Dot <in> Ada.Characters.Latin_1 [5051], page 578,

Min_Delta <in> Ada.Decimal [7175], page 1055,

Min_Handler_Ceiling
 <in> Ada.Execution_Time.Group_Budgets [7021], page 1028,
 <in> Ada.Execution_Time.Timers [7005], page 1025,

Min_Int <in> System [4607], page 510,

Min_Scale <in> Ada.Decimal [7174], page 1055,

Minus_Sign <in> Ada.Characters.Latin_1 [4941], page 574,

Monday <in> Ada.Calendar.Formatting [3775], page 365,

Multiplication_Sign
 <in> Ada.Characters.Latin_1 [5083], page 578,

MW <in> Ada.Characters.Latin_1 [5014], page 576,
NAK <in> Ada.Characters.Latin_1 [4916], page 574,
Native_Binary <in> Interfaces.COBOL [6608], page 933,
NBH <in> Ada.Characters.Latin_1 [4996], page 576,
NBSP <in> Ada.Characters.Latin_1 [5026], page 577,
NEL <in> Ada.Characters.Latin_1 [4998], page 576,
No_Break_Space
 <in> Ada.Characters.Latin_1 [5025], page 577,
No_Element
 <in> Ada.Containers.Doubly_Linked_Lists [6086], page 811,
 <in> Ada.Containers.Hashed_Maps [6154], page 840,
 <in> Ada.Containers.Hashed_Sets [6262], page 867,
 <in> Ada.Containers.Ordered_Maps [6199], page 847,
 <in> Ada.Containers.Ordered_Sets [6326], page 876,
 <in> Ada.Containers.Vectors [6002], page 780,
No_Index <in> Ada.Containers.Vectors [5998], page 780,
No_Tag <in> Ada.Tags [2028], page 137,
Not_Sign <in> Ada.Characters.Latin_1 [5038], page 577,
NUL
 <in> Ada.Characters.Latin_1 [4895], page 573,
 <in> Interfaces.C [6490], page 903,
Null_Address <in> System [4617], page 511,
Null_Bounded_String
 <in> Ada.Strings.Bounded [5220], page 611,
Null_Id <in> Ada.Exceptions [4142], page 423,
Null_Occurrence <in> Ada.Exceptions [4148], page 424,
Null_Ptr <in> Interfaces.C.Strings [6536], page 916,
Null_Set
 <in> Ada.Strings.Maps [5158], page 585,
 <in> Ada.Strings.Wide_Maps [5349], page 637,
 <in> Ada.Strings.Wide_Wide_Maps [5380], page 642,
Null_Unbounded_String
 <in> Ada.Strings.Unbounded [5276], page 625,
Number_Sign <in> Ada.Characters.Latin_1 [4930], page 574,
OSC <in> Ada.Characters.Latin_1 [5022], page 577,
Packed_Signed <in> Interfaces.COBOL [6611], page 934,
Packed_Unsigned <in> Interfaces.COBOL [6610], page 934,
Paragraph_Sign
 <in> Ada.Characters.Latin_1 [5050], page 577,
Percent_Sign
 <in> Ada.Characters.Latin_1 [4932], page 574,

Pi <in> Ada.Numerics [5408], page 648,
Pilcrow_Sign
 <in> Ada.Characters.Latin_1 [5049], page 577,
PLD <in> Ada.Characters.Latin_1 [5004], page 576,
PLU <in> Ada.Characters.Latin_1 [5005], page 576,
Plus_Minus_Sign
 <in> Ada.Characters.Latin_1 [5044], page 577,
Plus_Sign <in> Ada.Characters.Latin_1 [4938], page 574,
PM <in> Ada.Characters.Latin_1 [5023], page 577,
Pound_Sign <in> Ada.Characters.Latin_1 [5029], page 577,
PU1 <in> Ada.Characters.Latin_1 [5010], page 576,
PU2 <in> Ada.Characters.Latin_1 [5011], page 576,
Question <in> Ada.Characters.Latin_1 [4949], page 575,
Quotation <in> Ada.Characters.Latin_1 [4929], page 574,
Registered_Trade_Mark_Sign
 <in> Ada.Characters.Latin_1 [5040], page 577,
Reserved_128
 <in> Ada.Characters.Latin_1 [4993], page 576,
Reserved_129
 <in> Ada.Characters.Latin_1 [4994], page 576,
Reserved_132
 <in> Ada.Characters.Latin_1 [4997], page 576,
Reserved_153
 <in> Ada.Characters.Latin_1 [5018], page 577,
Reverse_Solidus
 <in> Ada.Characters.Latin_1 [4952], page 575,
RI <in> Ada.Characters.Latin_1 [5006], page 576,
Right_Angle_Quotation
 <in> Ada.Characters.Latin_1 [5055], page 578,
Right_Curly_Bracket
 <in> Ada.Characters.Latin_1 [4985], page 576,
Right_Parenthesis
 <in> Ada.Characters.Latin_1 [4936], page 574,
Right_Square_Bracket
 <in> Ada.Characters.Latin_1 [4953], page 575,
Ring_Above <in> Ada.Characters.Latin_1 [5043], page 577,
RS <in> Ada.Characters.Latin_1 [4925], page 574,
Saturday <in> Ada.Calendar.Formatting [3780], page 365,
SCHAR_MAX <in> Interfaces.C [6473], page 902,
SCHAR_MIN <in> Interfaces.C [6472], page 902,
SCI <in> Ada.Characters.Latin_1 [5019], page 577,

Section_Sign

<in> Ada.Characters.Latin_1 [5033], page 577,
Semicolon <in> Ada.Characters.Latin_1 [4945], page 575,
SI <in> Ada.Characters.Latin_1 [4910], page 573,
SO <in> Ada.Characters.Latin_1 [4909], page 573,
Soft_Hyphen <in> Ada.Characters.Latin_1 [5039], page 577,
SOH <in> Ada.Characters.Latin_1 [4896], page 573,
Solidus <in> Ada.Characters.Latin_1 [4943], page 574,
SOS <in> Ada.Characters.Latin_1 [5017], page 577,
SPA <in> Ada.Characters.Latin_1 [5015], page 576,

Space

<in> Ada.Characters.Latin_1 [4927], page 574,
<in> Ada.Strings [5144], page 584,

Special_Set

<in> Ada.Strings.Maps.Constants [5334], page 636,
SS2 <in> Ada.Characters.Latin_1 [5007], page 576,
SS3 <in> Ada.Characters.Latin_1 [5008], page 576,
SSA <in> Ada.Characters.Latin_1 [4999], page 576,
ST <in> Ada.Characters.Latin_1 [5021], page 577,
Storage_Unit <in> System [4618], page 511,
STS <in> Ada.Characters.Latin_1 [5012], page 576,
STX <in> Ada.Characters.Latin_1 [4897], page 573,
SUB <in> Ada.Characters.Latin_1 [4921], page 574,
Success <in> Ada.Command_Line [5923], page 755,
Sunday <in> Ada.Calendar.Formatting [3781], page 365,

Superscript_One

<in> Ada.Characters.Latin_1 [5053], page 578,

Superscript_Three

<in> Ada.Characters.Latin_1 [5046], page 577,

Superscript_Two

<in> Ada.Characters.Latin_1 [5045], page 577,

SYN <in> Ada.Characters.Latin_1 [4917], page 574,

System_Name <in> System [4606], page 510,

Thursday <in> Ada.Calendar.Formatting [3778], page 365,

Tick

<in> Ada.Real_Time [6941], page 1008,
<in> System [4615], page 511,

Tilde <in> Ada.Characters.Latin_1 [4986], page 576,

Time_First <in> Ada.Real_Time [6933], page 1008,

Time_Last <in> Ada.Real_Time [6934], page 1008,

Time_Span_First <in> Ada.Real_Time [6937], page 1008,
Time_Span_Last <in> Ada.Real_Time [6938], page 1008,
Time_Span_Unit <in> Ada.Real_Time [6940], page 1008,
Time_Span_Zero <in> Ada.Real_Time [6939], page 1008,
Time_Unit <in> Ada.Real_Time [6935], page 1008,
Trailing_Nonseparate
 <in> Interfaces.COBOL [6604], page 933,
Trailing_Separate <in> Interfaces.COBOL [6602], page 933,
Tuesday <in> Ada.Calendar.Formatting [3776], page 365,
UC_A_Acute <in> Ada.Characters.Latin_1 [5061], page 578,
UC_A_Circumflex
 <in> Ada.Characters.Latin_1 [5062], page 578,
UC_A_Diaeresis
 <in> Ada.Characters.Latin_1 [5064], page 578,
UC_A_Grave <in> Ada.Characters.Latin_1 [5060], page 578,
UC_A_Ring <in> Ada.Characters.Latin_1 [5065], page 578,
UC_A_Tilde <in> Ada.Characters.Latin_1 [5063], page 578,
UC_AE_Diphthong
 <in> Ada.Characters.Latin_1 [5066], page 578,
UC_C_Cedilla
 <in> Ada.Characters.Latin_1 [5067], page 578,
UC_E_Acute <in> Ada.Characters.Latin_1 [5069], page 578,
UC_E_Circumflex
 <in> Ada.Characters.Latin_1 [5070], page 578,
UC_E_Diaeresis
 <in> Ada.Characters.Latin_1 [5071], page 578,
UC_E_Grave <in> Ada.Characters.Latin_1 [5068], page 578,
UC_I_Acute <in> Ada.Characters.Latin_1 [5073], page 578,
UC_I_Circumflex
 <in> Ada.Characters.Latin_1 [5074], page 578,
UC_I_Diaeresis
 <in> Ada.Characters.Latin_1 [5075], page 578,
UC_I_Grave <in> Ada.Characters.Latin_1 [5072], page 578,
UC_Icelandic_Eth
 <in> Ada.Characters.Latin_1 [5076], page 578,
UC_Icelandic_Thorn
 <in> Ada.Characters.Latin_1 [5090], page 578,
UC_N_Tilde <in> Ada.Characters.Latin_1 [5077], page 578,
UC_O_Acute <in> Ada.Characters.Latin_1 [5079], page 578,
UC_O_Circumflex
 <in> Ada.Characters.Latin_1 [5080], page 578,

UC_O_Diaeresis
 <in> Ada.Characters.Latin_1 [5082], page 578,
UC_O_Grave <in> Ada.Characters.Latin_1 [5078], page 578,
UC_O_Oblique_Stroke
 <in> Ada.Characters.Latin_1 [5084], page 578,
UC_O_Tilde <in> Ada.Characters.Latin_1 [5081], page 578,
UC_U_Acute <in> Ada.Characters.Latin_1 [5086], page 578,
UC_U_Circumflex
 <in> Ada.Characters.Latin_1 [5087], page 578,
UC_U_Diaeresis
 <in> Ada.Characters.Latin_1 [5088], page 578,
UC_U_Grave <in> Ada.Characters.Latin_1 [5085], page 578,
UC_Y_Acute <in> Ada.Characters.Latin_1 [5089], page 578,
UCHAR_MAX <in> Interfaces.C [6474], page 902,
Unbounded <in> Ada.Text_IO [5673], page 698,
Unsigned <in> Interfaces.COBOL [6600], page 933,
Upper_Case_Map
 <in> Ada.Strings.Maps.Constants [5337], page 636,
Upper_Set
 <in> Ada.Strings.Maps.Constants [5329], page 636,
US <in> Ada.Characters.Latin_1 [4926], page 574,
Vertical_Line
 <in> Ada.Characters.Latin_1 [4984], page 576,
VT <in> Ada.Characters.Latin_1 [4906], page 573,
VTS <in> Ada.Characters.Latin_1 [5003], page 576,
Wednesday <in> Ada.Calendar.Formatting [3777], page 365,
Wide_Character_Set
 <in> Ada.Strings.Wide_Maps.Wide_Constants [5401], page 646,
wide_nul <in> Interfaces.C [6500], page 904,
Wide_Space <in> Ada.Strings [5145], page 584,
Wide_Wide_Space <in> Ada.Strings [5146], page 584,
Word_Size <in> System [4619], page 511,
Yen_Sign <in> Ada.Characters.Latin_1 [5031], page 577,

30 Index

Index entries are given by paragraph number.

30.1 operators

& operator [2496], page 201, [2660], page 212,

* operator [2503], page 201, [2685], page 213,

** operator [2517], page 201, [2716], page 218,

+ operator [2488], page 201, [2652], page 211, [2675], page 213,

– operator [2492], page 201, [2656], page 211, [2679], page 213,

/ operator [2509], page 201, [2691], page 213,

/= operator [2466], page 201, [2618], page 206,

10646:2003, ISO/IEC standard [1109], page 30,

14882:2003, ISO/IEC standard [1112], page 30,

1539–1:2004, ISO/IEC standard [1092], page 30,

19769:2004, ISO/IEC technical report [1115], page 30,

1989:2002, ISO standard [1095], page 30,

6429:1992, ISO/IEC standard [1098], page 30,

646:1991, ISO/IEC standard [1089], page 30,

8859–1:1987, ISO/IEC standard [1103], page 30,

9899:1999, ISO/IEC standard [1106], page 30,

< operator [2470], page 201, [2622], page 206,

<= operator [2474], page 201, [2626], page 206,

= operator [2462], page 201, [2614], page 206,

> operator [2478], page 201, [2630], page 206,
>= operator [2482], page 201, [2634], page 206,

30.2 A

AARM [1002], page 4,
abnormal completion [3322], page 299,
abnormal state of an object [4666], page 522,
 <partial> [3888], page 387, [4224], page 449, [5916], page 754,
abnormal task [3878], page 386,
abort
 of a partition [7062], page 1035,
 of a task [3877], page 386,
 of the execution of a construct [3881], page 386,
abort completion point [3884], page 387,
abort–deferred operation [3882], page 386,
abort_statement [3872], page 385,
 <used> [2918], page 240, [8084], page 1298,
Abort_Task
 <in> Ada.Task_Identification [6751], page 965,
abortable_part [3859], page 384,
 <used> [3852], page 384, [8323], page 1305,
abs operator [2521], page 201, [2705], page 217,
absolute value [2523], page 201, [2707], page 217,
abstract data type (ADT)
 <See> private types and private extensions [3254], page 283,
 <See also> abstract type [2101], page 149,
abstract subprogram [2104], page 150, [2111], page 150,
abstract type [2109], page 150, [2100], page 149, [7671], page 1283,
abstract_subprogram_declaration [2106], page 150,
 <used> [1288], page 49, [7789], page 1290,
accept_alternative [3823], page 379,
 <used> [3820], page 378, [8307], page 1305,

accept_statement [3640], page 347,
 <used> [2927], page 241, [3824], page 379, [8092], page 1298,
acceptable interpretation [3470], page 325,
Access attribute [2223], page 168, [2233], page 173,
 <See also> Unchecked_Access attribute [4680], page 525,
access discriminant [1912], page 124,
access parameter [3073], page 257,
access paths
 distinct [3089], page 261,
access result type [3074], page 257,
access type [2142], page 156, [7672], page 1283,
access types
 input-output unspecified [5582], page 681,
access value [2143], page 156,
access-to-constant type [2178], page 157,
access-to-object type [2167], page 157,
access-to-subprogram type [2168], page 157, [2180], page 157,
access-to-variable type [2179], page 157,
Access_Check [4193], page 432,
 [<partial>] [2298], page 180, [2809], page 227,
access_definition [2160], page 156,
 <used> [1441], page 61, [1835], page 114, [1906], page 123, [3050], page 256, [3061],
page 256, [3189], page 272, [3431], page 317, [4308], page 458, [7900], page 1293,
access_to_object_definition [2152], page 156,
 <used> [2149], page 156, [7935], page 1294,
access_to_subprogram_definition [2156], page 156,
 <used> [2151], page 156, [7937], page 1294,
access_type_definition [2147], page 156,
 <used> [1368], page 54, [4371], page 468, [7811], page 1290,
accessibility
 from shared passive library units [7084], page 1039,
accessibility level [2209], page 164,
accessibility rule
 Access attribute [2226], page 172, [2234], page 173,
 requeue statement [3721], page 357,
 type conversion [2759], page 222, [2766], page 223,

type conversion, array components [2754], page 221,
Accessibility_Check [4202], page 443,
[<partial>] [2228], page 172, [2787], page 225, [2801], page 226, [2864], page 232, [3203],
page 274, [3206], page 274, [7140], page 1046,
accessible partition [7064], page 1035,
accuracy [2777], page 224, [7278], page 1103,
ACK
 <in> Ada.Characters.Latin_1 [4901], page 573,
acquire
 execution resource associated with protected object [3628], page 345,
activation
 of a task [3545], page 333,
activation failure [3547], page 333,
activator
 of a task [3548], page 334,
active partition [4049], page 412, [7057], page 1035,
active priority [6805], page 976,
actual [4286], page 455,
actual duration [6962], page 1014,
actual parameter
 for a formal parameter [3164], page 270,
actual subtype [1425], page 60, [4341], page 461,
 of an object [1460], page 62,
actual type [4343], page 461,
actual_parameter_part [3148], page 267,
 <used> [3143], page 267, [3147], page 267, [3690], page 352, [8167], page 1300,
Actual_Quantum
 <in> Ada.Dispatching.Round_Robin [6847], page 986,
Acute
 <in> Ada.Characters.Latin_1 [5047], page 577,
Ada [4856], page 565,
Ada calling convention [3109], page 263,
Ada.Assertions [4172], page 428,
Ada.Asynchronous_Task_Control [6972], page 1016,
Ada.Calendar [3741], page 359,
Ada.Calendar.Arithmetic [3769], page 364,

Ada.Calendar.Formatting [3773], page 365,
Ada.Calendar.Time_Zones [3765], page 364,
Ada.Characters [4857], page 565,
Ada.Characters.Conversions [5124], page 580,
Ada.Characters.Handling [4860], page 566,
Ada.Characters.Latin_1 [4893], page 573,
Ada.Command_Line [5918], page 755,
Ada.Complex_Text_IO [7273], page 1098,
Ada.Containers [5989], page 779,
Ada.Containers.Doubly_Linked_Lists [6082], page 811,
Ada.Containers.Generic_Array_Sort [6395], page 891,
Ada.Containers.Generic_Constrained_Array_Sort [6397], page 892,
Ada.Containers.Hashed_Maps [6150], page 840,
Ada.Containers.Hashed_Sets [6258], page 867,
Ada.Containers.Indefinite_Doubly_Linked_Lists [6390], page 888,
Ada.Containers.Indefinite_Hashed_Maps [6391], page 889,
Ada.Containers.Indefinite_Hashed_Sets [6393], page 890,
Ada.Containers.Indefinite_Ordered_Maps [6392], page 890,
Ada.Containers.Indefinite_Ordered_Sets [6394], page 891,
Ada.Containers.Indefinite_Vectors [6389], page 887,
Ada.Containers.Ordered_Maps [6194], page 846,
Ada.Containers.Ordered_Sets [6321], page 876,
Ada.Containers.Vectors [5996], page 780,
Ada.Decimal [7172], page 1055,
Ada.Direct_IO [5619], page 691,
Ada.Directories [5927], page 757,
Ada.Directories.Information [5974], page 774,
Ada.Dispatching [6809], page 978,
Ada.Dispatching.EDF [6852], page 988,
Ada.Dispatching.Round_Robin [6843], page 985,
Ada.Dynamic_Priorities [6888], page 996,
Ada.Environment_Variables [5976], page 775,
Ada.Exceptions [4140], page 423,
Ada.Execution_Time [6989], page 1021,
Ada.Execution_Time.Group_Budgets [7017], page 1027,
Ada.Execution_Time.Timers [7002], page 1024,

Ada.Finalization [3299], page 296,
Ada.Float_Text_IO [5829], page 735,
Ada.Float_Wide_Text_IO [5850], page 745,
Ada.Float_Wide_Wide_Text_IO [5853], page 745,
Ada.Integer_Text_IO [5828], page 730,
Ada.Integer_Wide_Text_IO [5849], page 745,
Ada.Integer_Wide_Wide_Text_IO [5852], page 745,
Ada.Interrupts [6697], page 957,
Ada.Interrupts.Names [6707], page 958,
Ada.IO_Exceptions [5905], page 752,
Ada.Numerics [5406], page 648,
Ada.Numerics.Complex_Arrays [7371], page 1136,
Ada.Numerics.Complex_Elementary_Functions [7255], page 1092,
Ada.Numerics.Complex_Types [7228], page 1087,
Ada.Numerics.Discrete_Random [5459], page 656,
Ada.Numerics.Elementary_Functions [5439], page 650,
Ada.Numerics.Float_Random [5447], page 655,
Ada.Numerics.Generic_Complex_Arrays [7335], page 1130,
Ada.Numerics.Generic_Complex_Elementary_Functions [7234], page 1091,
Ada.Numerics.Generic_Complex_Types [7207], page 1084,
Ada.Numerics.Generic_Elementary_Functions [5410], page 649,
Ada.Numerics.Generic_Real_Arrays [7319], page 1119,
Ada.Numerics.Real_Arrays [7331], page 1122,
Ada.Real_Time [6931], page 1008,
Ada.Real_Time.Timing_Events [7041], page 1031,
Ada.Sequential_IO [5596], page 683,
Ada.Storage_IO [5649], page 695,
Ada.Streams [4759], page 538,
Ada.Streams.Stream_IO [5861], page 746,
Ada.Strings [5143], page 584,
Ada.Strings.Bounded [5216], page 611,
Ada.Strings.Bounded.Hash [5404], page 647,
Ada.Strings.Fixed [5180], page 592,
Ada.Strings.Hash [5403], page 646,
Ada.Strings.Maps [5156], page 585,
Ada.Strings.Maps.Constants [5324], page 636,

Ada.Strings.Unbounded [5274], page 625,
Ada.Strings.Unbounded.Hash [5405], page 647,
Ada.Strings.Wide_Bounded [5340], page 636,
Ada.Strings.Wide_Bounded.Wide.Hash [5344], page 636,
Ada.Strings.Wide_Fixed [5339], page 636,
Ada.Strings.Wide_Fixed.Wide.Hash [5343], page 636,
Ada.Strings.Wide_Hash [5342], page 636,
Ada.Strings.Wide_Maps [5347], page 637,
Ada.Strings.Wide_Maps.Wide_Constants [5346], page 636, [5399], page 644,
Ada.Strings.Wide_Unbounded [5341], page 636,
Ada.Strings.Wide_Unbounded.Wide.Hash [5345], page 636,
Ada.Strings.Wide_Wide_Bounded [5371], page 641,
Ada.Strings.Wide_Wide_Bounded.Wide_Wide.Hash [5375], page 641,
Ada.Strings.Wide_Wide_Fixed [5370], page 641,
Ada.Strings.Wide_Wide_Fixed.Wide_Wide.Hash [5374], page 641,
Ada.Strings.Wide_Wide_Hash [5373], page 641,
Ada.Strings.Wide_Wide_Maps [5378], page 642,
Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants [5377], page 641,
Ada.Strings.Wide_Wide_Unbounded [5372], page 641,
Ada.Strings.Wide_Wide_Unbounded.Wide_Wide.Hash [5376], page 641,
Ada.Synchronous_Task_Control [6963], page 1015,
Ada.Tags [2026], page 137,
Ada.Tags.Generic_Dispatching_Constructor [2049], page 141,
Ada.Task_Attributes [6764], page 967,
Ada.Task_Identification [6746], page 965,
Ada.Task_Termination [6776], page 971,
Ada.Text_IO [5668], page 698,
Ada.Text_IO.Bounded_IO [5830], page 739,
Ada.Text_IO.Complex_IO [7263], page 1097,
Ada.Text_IO.Editing [7183], page 1073,
Ada.Text_IO.Text_Streams [5896], page 751,
Ada.Text_IO.Unbounded_IO [5839], page 742,
Ada.Unchecked_Conversion [4658], page 520,
Ada.Unchecked_Deallocation [4719], page 532,
Ada.Wide_Characters [4858], page 566,
Ada.Wide_Text_IO [5848], page 745,

Ada.Wide_Text_IO.Bounded_IO [5854], page 745,
Ada.Wide_Text_IO.Complex_IO [7275], page 1103,
Ada.Wide_Text_IO.Editing [7202], page 1081,
Ada.Wide_Text_IO.Text_Streams [5899], page 751,
Ada.Wide_Text_IO.Unbounded_IO [5856], page 745,
Ada.Wide_Wide_Characters [4859], page 566,
Ada.Wide_Wide_Text_IO [5851], page 745,
Ada.Wide_Wide_Text_IO.Bounded_IO [5855], page 745,
Ada.Wide_Wide_Text_IO.Complex_IO [7277], page 1103,
Ada.Wide_Wide_Text_IO.Editing [7204], page 1081,
Ada.Wide_Wide_Text_IO.Text_Streams [5902], page 752,
Ada.Wide_Wide_Text_IO.Unbounded_IO [5857], page 745,
Ada_To_COBOL
 <in> Interfaces.COBOL [6591], page 932,
adafinal [6462], page 898,
adainit [6461], page 898,
Add
 <in> Ada.Execution_Time.Group_Budgets [7028], page 1028,
Add_Task
 <in> Ada.Execution_Time.Group_Budgets [7022], page 1028,
address
 arithmetic [4636], page 516,
 comparison [4621], page 511,
 <in> System [4616], page 511,
Address attribute [4496], page 488, [7438], page 1171,
Address clause [4476], page 487, [4498], page 488,
Address_To_Access_Conversions
 <child of> System [4643], page 517,
Adjacent attribute [5535], page 673,
Adjust [3298], page 295,
 <in> Ada.Finalization [3302], page 296,
adjusting the value of an object [3311], page 298, [3313], page 298,
adjustment [3312], page 298, [3314], page 298,
 as part of assignment [2954], page 243,
ADT (abstract data type)
 <See> private types and private extensions [3255], page 283,

<See also> abstract type [2102], page 149,
advice [1048], page 23,
Aft attribute [1805], page 111,
aggregate [2369], page 190, [2371], page 190,
 <used> [2562], page 201, [2830], page 229, [8054], page 1297,
 <See also> composite type [1324], page 51,
aliased [2175], page 157, [7673], page 1283,
aliasing
 <See> distinct access paths [3090], page 261,
Alignment
 <in> Ada.Strings [5151], page 584,
Alignment attribute [4502], page 489, [4506], page 490,
Alignment clause [4477], page 487, [4504], page 490, [4508], page 490,
All_Calls_Remote pragma [7105], page 1042, [7502], page 1242,
All_Checks [4207], page 447,
Allocate
 <in> System.Storage_Pools [4687], page 526,
Allocation_Check [4203], page 443,
 [<partial>] [2867], page 232, [2870], page 232,
allocator [2844], page 231,
 <used> [2565], page 201, [8057], page 1297,
Alphanumeric
 <in> Interfaces.COBOL [6593], page 932,
alphanumeric character
 a category of Character [4891], page 570,
Alphanumeric_Set
 <in> Ada.Strings.Maps.Constants [5333], page 636,
ambiguous [3487], page 327,
ambiguous cursor
 of a vector [6071], page 808,
ampersand [1147], page 36,
 <in> Ada.Characters.Latin_1 [4933], page 574,
ampersand operator [2498], page 201, [2662], page 212,
ancestor [7674], page 1283,
 of a library unit [3950], page 396,
 of a type [1523], page 75,

ultimate [1525], page 75,
ancestor subtype
 of a formal derived type [4354], page 463,
 of a private_extension_declaration [3266], page 284,
ancestor_part [2405], page 194,
 <used> [2403], page 194, [8005], page 1296,
and operator [2454], page 201, [2594], page 205,
and then (short-circuit control form) [2460], page 201, [2589], page 204,
angle threshold [7312], page 1113,
Annex
 informative [1013], page 21,
 normative [1010], page 21,
 Specialized Needs [1007], page 20,
Annotated Ada Reference Manual [1001], page 4,
anonymous access type [2183], page 157,
anonymous array type [1433], page 61,
anonymous protected type [1435], page 61,
anonymous task type [1434], page 61,
anonymous type [1373], page 54,
Any_Priority <subtype of> Integer
 <in> System [4626], page 512,
APC
 <in> Ada.Characters.Latin_1 [5024], page 577,
apostrophe [1148], page 36,
 <in> Ada.Characters.Latin_1 [4934], page 574,
Append
 <in> Ada.Containers.Doubly_Linked_Lists [6099], page 813,
 <in> Ada.Containers.Vectors [6032], page 784, [6033], page 784,
 <in> Ada.Strings.Bounded [5226], page 612, [5227], page 612, [5228], page 612, [5229],
page 612, [5230], page 612, [5231], page 612, [5232], page 612, [5233], page 613,
 <in> Ada.Strings.Unbounded [5284], page 626, [5285], page 626, [5286], page 626,
applicable index constraint [2441], page 197,
application areas [1008], page 20,
apply
 to a callable construct by a return statement [3193], page 272,
 to a loop_statement by an exit_statement [3005], page 252,

to a program unit by a program unit pragma [4010], page 407,
arbitrary order [1069], page 27,

Arccos

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7244], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5426], page 649,

Arccosh

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7252], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5436], page 650,

Arccot

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7246], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5429], page 649,

Arccoth

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7254], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5438], page 650,

Arcsin

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7243], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5424], page 649,

Arcsinh

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7251], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5435], page 650,

Arctan

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7245], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5428], page 649,

Arctanh

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7253], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5437], page 650,

Argument

<in> Ada.Command_Line [5920], page 755,
<in> Ada.Numerics.Generic_Complex_Arrays [7346], page 1131, [7359], page 1133,
<in> Ada.Numerics.Generic_Complex_Types [7222], page 1085,

argument of a pragma [1266], page 45,

Argument_Count

<in> Ada.Command_Line [5919], page 755,

Argument_Error

<in> Ada.Numerics [5407], page 648,

Arithmetic

<child of> Ada.Calendar [3769], page 364,
array [1815], page 114,
array component expression [2436], page 197,
array indexing
 <See> indexed_component [2301], page 181,
array slice [2313], page 182,
array type [1816], page 114, [7675], page 1283,
array_aggregate [2415], page 196,
 <used> [2374], page 190, [4547], page 500, [8474], page 1310,
array_component_association [2430], page 196,
 <used> [2428], page 196, [8019], page 1296,
array_type_definition [1817], page 114,
 <used> [1366], page 54, [1444], page 61, [4367], page 467, [7834], page 1291,
ASCII
 package physically nested within the declaration of Standard [4846], page 562,
 <in> Standard [4845], page 562,
aspect of representation [4431], page 482,
 coding [4549], page 500,
 controlled [4736], page 534,
 convention, calling convention [6444], page 897,
 exported [6448], page 897,
 imported [6446], page 897,
 layout [4553], page 501,
 packing [4455], page 486,
 record layout [4555], page 501,
 specifiable attributes [4473], page 487,
 storage place [4557], page 501,
aspect_clause [4417], page 481,
 <used> [1963], page 130, [2248], page 175, [3523], page 329, [3581], page 338, [3593],
page 338, [8242], page 1302,
assembly language [6663], page 949,
Assert pragma [4164], page 427, [7505], page 1242,
assertion policy [4173], page 428,
Assertion_Policy pragma [4168], page 427, [7509], page 1242,
Assertions [4162], page 427,
 <child of> Ada [4172], page 428,

assign

<See> assignment operation [2941], page 242,

assigning back of parameters [3176], page 272,

assignment

user-defined [3291], page 295,

assignment operation [2940], page 242, [2952], page 243, [3310], page 297,

during elaboration of an object_declaration [1470], page 63,

during evaluation of a generic_association for a formal object of mode in [4322], page 460,

during evaluation of a parameter_association [3169], page 271,

during evaluation of an aggregate [2377], page 190,

during evaluation of an initialized allocator [2855], page 232,

during evaluation of an uninitialized allocator [2858], page 232,

during evaluation of concatenation [2672], page 212,

during execution of a for loop [2995], page 249,

during execution of an assignment_statement [2953], page 243,

during parameter copy back [3177], page 272,

assignment_statement [2937], page 242,

<used> [2910], page 240, [8076], page 1298,

associated components

of a record_component_association [2398], page 192,

associated discriminants

of a named discriminant_association [1937], page 128,

of a positional discriminant_association [1938], page 128,

associated object

of a value of a by-reference type [3086], page 261,

asterisk [1152], page 36,

<in> Ada.Characters.Latin_1 [4937], page 574,

asynchronous

remote procedure call [7152], page 1047,

Asynchronous pragma [7147], page 1047, [7512], page 1242,

asynchronous remote procedure call [7127], page 1044,

asynchronous_select [3850], page 384,

<used> [3809], page 378, [8300], page 1304,

Asynchronous_Task_Control

<child of> Ada [6972], page 1016,

at-most-once execution [7136], page 1045,

at_clause [7433], page 1171,
 <used> [4421], page 481, [8461], page 1310,
atomic [6732], page 963,
Atomic pragma [6721], page 963, [7515], page 1242,
Atomic_Components pragma [6727], page 963, [7518], page 1242,
Attach_Handler
 <in> Ada.Interrupts [6703], page 957,
Attach_Handler pragma [6681], page 954, [7521], page 1242,
attaching
 to an interrupt [6672], page 951,
attribute [2339], page 187, [7458], page 1179,
 representation [4460], page 486,
 specifiable [4472], page 487,
 specifying [4461], page 486,
attribute_definition_clause [4462], page 487,
 <used> [4418], page 481, [8458], page 1310,
attribute_designator [2343], page 187,
 <used> [2342], page 187, [4425], page 481, [4467], page 487, [8464], page 1310,
Attribute_Handle
 <in> Ada.Task_Attributes [6765], page 967,
attribute_reference [2340], page 187,
 <used> [2276], page 179, [7967], page 1295,
attributes
 Access [2222], page 168, [2232], page 173,
 Address [4495], page 488, [7437], page 1171,
 Adjacent [5534], page 673,
 Aft [1804], page 111,
 Alignment [4501], page 489, [4505], page 490,
 Base [1562], page 78,
 Bit_Order [4596], page 508,
 Body_Version [7119], page 1044,
 Callable [3891], page 388,
 Caller [6756], page 966,
 Ceiling [5518], page 670,
 Class [2041], page 139, [3278], page 288, [7453], page 1176,
 Component_Size [4534], page 497,

Compose [5506], page 667,
Constrained [1946], page 129, [7431], page 1169,
Copy_Sign [5539], page 674,
Count [3897], page 389,
Definite [4357], page 465,
Delta [1800], page 110,
Denorm [5488], page 664,
Digits [1754], page 105, [1806], page 111,
Exponent [5502], page 667,
External_Tag [4539], page 498,
First [1556], page 77, [1872], page 119,
First(N) [1874], page 119,
First_Bit [4583], page 506,
Floor [5516], page 669,
Fore [1802], page 110,
Fraction [5504], page 667,
Identity [4159], page 425, [6754], page 966,
Image [1590], page 84,
Input [4781], page 544, [4785], page 546,
Last [1558], page 77, [1876], page 120,
Last(N) [1878], page 120,
Last_Bit [4585], page 507,
Leading_Part [5544], page 674,
Length [1884], page 120,
Length(N) [1886], page 120,
Machine [5549], page 675,
Machine_Emax [5486], page 664,
Machine_Emin [5484], page 664,
Machine_Mantissa [5482], page 663,
Machine_Overflows [5496], page 665, [5574], page 680,
Machine_Radix [5479], page 663, [5570], page 679,
Machine_Rounding [5524], page 671,
Machine_Rounds [5494], page 665, [5572], page 679,
Max [1567], page 78,
Max_Size_In_Storage_Elements [4712], page 531,
Min [1565], page 78,

Mod [1685], page 96,
Model [5563], page 678, [7298], page 1108,
Model_Emin [5557], page 677, [7291], page 1106,
Model_Epsilon [5559], page 677,
Model_Mantissa [5555], page 676, [7289], page 1105,
Model_Small [5561], page 677,
Modulus [1687], page 97,
Output [4779], page 543, [4783], page 545,
Partition_Id [7065], page 1035,
Pos [1706], page 99,
Position [4581], page 506,
Pred [1576], page 79,
Priority [6894], page 999,
Range [1560], page 77, [1880], page 120,
Range(N) [1882], page 120,
Read [4773], page 541, [4777], page 543,
Remainder [5529], page 672,
Round [1812], page 113,
Rounding [5520], page 670,
Safe_First [5565], page 678, [7293], page 1107,
Safe_Last [5567], page 679, [7295], page 1107,
Scale [1809], page 112,
Scaling [5511], page 668,
Signed_Zeros [5498], page 666,
Size [4513], page 492, [4518], page 493,
Small [1796], page 109,
Storage_Pool [4695], page 527,
Storage_Size [4523], page 496, [4697], page 527, [7446], page 1174,
Stream_Size [4768], page 540,
Succ [1569], page 79,
Tag [2045], page 140, [2047], page 140,
Terminated [3895], page 388,
Truncation [5527], page 672,
Unbiased_Rounding [5522], page 671,
Unchecked_Access [4678], page 525, [7404], page 1160,
Val [1708], page 100,

Valid [4676], page 524, [7378], page 1153,
Value [1616], page 89,
Version [7117], page 1043,
Wide_Image [1587], page 83,
Wide_Value [1610], page 88,
Wide_Wide_Image [1583], page 80,
Wide_Wide_Value [1598], page 86,
Wide_Wide_Width [1592], page 85,
Wide_Width [1594], page 85,
Width [1596], page 85,
Write [4771], page 541, [4775], page 542,
available
stream attribute [4800], page 547,

30.3 B

Backus–Naur Form (BNF)
complete listing [7739], page 1289,
cross reference [8500], page 1311,
notation [1064], page 25,
under Syntax heading [1018], page 22,
base [1233], page 41, [1241], page 41,
<used> [1229], page 41, [7764], page 1289,
base 16 literal [1226], page 41,
base 2 literal [1220], page 41,
base 8 literal [1223], page 41,
Base attribute [1563], page 78,
base decimal precision
of a floating point type [1740], page 104,
of a floating point type [1742], page 104,
base priority [6804], page 976,
base range
of a decimal fixed point type [1786], page 108,

- of a fixed point type [1781], page 107,
- of a floating point type [1739], page 104, [1744], page 104,
- of a modular type [1673], page 96,
- of a scalar type [1547], page 77,
- of a signed integer type [1670], page 96,
- of an ordinary fixed point type [1782], page 107,

base subtype

- of a type [1564], page 78,

Base_Name

- <in> Ada.Directories [5941], page 759,

based_literal [1228], page 41,

- <used> [1204], page 40, [7755], page 1289,

based_numeral [1235], page 41,

- <used> [1231], page 41, [7766], page 1289,

basic letter

- a category of Character [4888], page 569,

basic_declaration [1282], page 49,

- <used> [2247], page 175, [7953], page 1295,

basic_declarative_item [2246], page 175,

- <used> [2244], page 175, [3234], page 279, [7951], page 1295,

Basic_Map

- <in> Ada.Strings.Maps.Constants [5338], page 636,

Basic_Set

- <in> Ada.Strings.Maps.Constants [5330], page 636,

become nonlimited [3275], page 287, [3287], page 294,

BEL

- <in> Ada.Characters.Latin_1 [4902], page 573,

belong

- to a range [1542], page 76,
- to a subtype [1342], page 52,

bibliography [1087], page 30,

big endian [4591], page 508,

binary

- literal [1221], page 41,
- <in> Interfaces.COBOLE [6584], page 932,

binary adding operator [2650], page 211,

binary literal [1219], page 41,
binary operator [2581], page 203,
binary_adding_operator [2575], page 203,
 <used> [2548], page 201, [8043], page 1297,
Binary_Format
 <in> Interfaces.COBOL [6605], page 933,
bit field
 <See> record_representation_clause [4559], page 501,
bit ordering [4589], page 508,
bit string
 <See> logical operators on boolean arrays [2600], page 205,
Bit_Order
 <in> System [4622], page 512,
Bit_Order attribute [4597], page 508,
Bit_Order clause [4482], page 487, [4599], page 508,
blank
 in text input for enumeration and numeric types [5823], page 721,
Blank_When_Zero
 <in> Ada.Text_IO.Editing [7188], page 1073,
block_statement [2996], page 251,
 <used> [2925], page 240, [8090], page 1298,
blocked
 [<partial>] [6822], page 979,
 a task state [3501], page 328,
 during an entry call [3710], page 354,
 execution of a selective_accept [3833], page 380,
 on a delay_statement [3759], page 361,
 on an accept_statement [3681], page 350,
 waiting for activations to complete [3549], page 334,
 waiting for dependents to terminate [3556], page 335,
blocked interrupt [6671], page 951,
blocking, potentially [3633], page 345,
 Abort_Task [6760], page 967,
 delay_statement [3762], page 363, [6960], page 1013,
 remote subprogram call [7139], page 1046,
 RPC operations [7168], page 1052,

Suspend_Until_True [6970], page 1016,
BMP [1639], page 93, [1647], page 94, [1644], page 94,
BNF (Backus–Naur Form)
 complete listing [7738], page 1289,
 cross reference [8499], page 1311,
 notation [1063], page 25,
 under Syntax heading [1017], page 22,
body [2250], page 176, [2266], page 177,
 <used> [2245], page 175, [7952], page 1295,
body_stub [3980], page 403,
 <used> [2252], page 176, [7957], page 1295,
Body_Version attribute [7120], page 1044,
Boolean [1649], page 95,
 <in> Standard [4837], page 557,
boolean type [1652], page 95,
Bounded
 <child of> Ada.Strings [5216], page 611,
bounded error [1039], page 22, [1078], page 29,
 cause [2873], page 232, [3091], page 261, [3336], page 301, [3613], page 341, [3631],
 page 345, [3885], page 387, [4046], page 412, [4671], page 523, [4724], page 533, [5983],
 page 777, [6069], page 808, [6070], page 808, [6073], page 808, [6131], page 829, [6761],
 page 967, [6772], page 969, [6858], page 990, [6875], page 992, [7067], page 1036, [7123],
 page 1044, [7443], page 1172,
Bounded_IO
 <child of> Ada.Text_IO [5830], page 739,
 <child of> Ada.Wide_Text_IO [5854], page 745,
 <child of> Ada.Wide_Wide_Text_IO [5855], page 745,
Bounded_Slice
 <in> Ada.Strings.Bounded [5237], page 614, [5238], page 614,
Bounded_String
 <in> Ada.Strings.Bounded [5219], page 611,
bounds
 of a discrete_range [1866], page 118,
 of an array [1844], page 115,
 of the index range of an array_aggregate [2445], page 199,
box

compound delimiter [1849], page 115,

BPH

<in> Ada.Characters.Latin_1 [4995], page 576,

broadcast signal

<See> protected object [3564], page 338,

<See> requeue [3715], page 356,

Broken_Bar

<in> Ada.Characters.Latin_1 [5032], page 577,

BS

<in> Ada.Characters.Latin_1 [4903], page 573,

budget [7035], page 1029,

Budget_Has_Expired

<in> Ada.Execution_Time.Group_Budgets [7029], page 1028,

Budget_Remaining

<in> Ada.Execution_Time.Group_Budgets [7030], page 1028,

Buffer_Size

<in> Ada.Storage_IO [5650], page 695,

Buffer_Type <subtype of> Storage_Array

<in> Ada.Storage_IO [5651], page 695,

by copy parameter passing [3079], page 260,

by reference parameter passing [3082], page 260,

by-copy type [3084], page 260,

by-reference type [3085], page 260,

atomic or volatile [6744], page 964,

Byte

<in> Interfaces.COBOL [6612], page 934,

<See> storage element [4492], page 487,

byte sex

<See> ordering of storage elements in a word [4600], page 508,

Byte_Array

<in> Interfaces.COBOL [6613], page 934,

30.4 C

C

<child of> Interfaces [6470], page 902,

C interface [6469], page 901,

C standard [1107], page 30,

C++ standard [1113], page 30,

C_float

<in> Interfaces.C [6486], page 903,

Calendar

<child of> Ada [3741], page 359,

call [3015], page 255,

call on a dispatching operation [2083], page 146,

callable [3894], page 388,

Callable attribute [3892], page 388,

callable construct [3016], page 255,

callable entity [3014], page 255,

called partition [7129], page 1044,

Caller attribute [6757], page 966,

calling convention [3108], page 263, [6432], page 895,

Ada [3110], page 263,

associated with a designated profile [2182], page 157,

entry [3116], page 264,

Intrinsic [3112], page 263,

protected [3114], page 264,

calling partition [7128], page 1044,

calling stub [7134], page 1045,

CAN

<in> Ada.Characters.Latin_1 [4919], page 574,

Cancel_Handler

<in> Ada.Execution_Time.Group_Budgets [7033], page 1028,

<in> Ada.Execution_Time.Timers [7009], page 1025,

<in> Ada.Real_Time.Timing_Events [7047], page 1031,

cancellation

of a delay_statement [3760], page 361,

of an entry call [3711], page 354,
cancellation of a remote subprogram call [7137], page 1045,
canonical form [5481], page 663,
canonical semantics [4218], page 448,
canonical–form representation [5493], page 665,
capacity
 of a hashed map [6185], page 843,
 of a hashed set [6310], page 873,
 of a vector [5994], page 779,
 <in> Ada.Containers.Hashed_Maps [6155], page 840,
 <in> Ada.Containers.Hashed_Sets [6265], page 868,
 <in> Ada.Containers.Vectors [6005], page 781,
case insensitive [1198], page 39,
case_statement [2965], page 246,
 <used> [2923], page 240, [8088], page 1298,
case_statement_alternative [2969], page 246,
 <used> [2967], page 246, [8105], page 1298,
cast
 <See> type conversion [2726], page 219,
 <See> unchecked type conversion [4657], page 520,
catch (an exception)
 <See> handle [4090], page 419,
categorization pragma [7069], page 1037,
 Remote_Call_Interface [7099], page 1041,
 Remote_Types [7090], page 1039,
 Shared_Passive [7078], page 1038,
categorized library unit [7073], page 1037,
category
 of types [1317], page 51, [1484], page 66,
category (of types) [7676], page 1283,
category determined for a formal type [4349], page 461,
catenation operator
 <See> concatenation operator [2502], page 201,
 <See> concatenation operator [2666], page 212,
Cause_Of_Termination
 <in> Ada.Task_Termination [6777], page 971,

CCH

<in> Ada.Characters.Latin_1 [5013], page 576,

cease to exist

object [3333], page 301, [4723], page 533,

type [3334], page 301,

Cedilla

<in> Ada.Characters.Latin_1 [5052], page 578,

Ceiling

<in> Ada.Containers.Ordered_Maps [6232], page 850,

<in> Ada.Containers.Ordered_Sets [6365], page 880, [6379], page 882,

Ceiling attribute [5519], page 670,

ceiling priority

of a protected object [6871], page 992,

Ceiling_Check

[<partial>] [6690], page 955, [6872], page 992,

Cent_Sign

<in> Ada.Characters.Latin_1 [5028], page 577,

change of representation [4602], page 509,

char

<in> Interfaces.C [6489], page 903,

char16_array

<in> Interfaces.C [6513], page 905,

char16_nul

<in> Interfaces.C [6510], page 905,

char16_t

<in> Interfaces.C [6509], page 905,

char32_array

<in> Interfaces.C [6523], page 906,

char32_nul

<in> Interfaces.C [6520], page 906,

char32_t

<in> Interfaces.C [6519], page 906,

char_array

<in> Interfaces.C [6493], page 903,

char_array_access

<in> Interfaces.C.Strings [6533], page 916,

CHAR_BIT

<in> Interfaces.C [6471], page 902,

Character [1641], page 93,

<used> [1252], page 43, [7776], page 1290,

<in> Standard [4842], page 560,

character plane [1123], page 32,

character set [1121], page 32,

character set standard

16 and 32-bit [1110], page 30,

7-bit [1090], page 30,

8-bit [1104], page 30,

control functions [1099], page 30,

character type [1637], page 93, [7677], page 1283,

character_literal [1242], page 42,

<used> [1630], page 92, [2279], page 179, [2329], page 184, [7970], page 1295,

Character_Mapping

<in> Ada.Strings.Maps [5170], page 587,

Character_Mapping_Function

<in> Ada.Strings.Maps [5176], page 587,

Character_Range

<in> Ada.Strings.Maps [5159], page 585,

Character_Ranges

<in> Ada.Strings.Maps [5160], page 585,

Character_Sequence <subtype of> String

<in> Ada.Strings.Maps [5166], page 586,

Character_Set

<in> Ada.Strings.Maps [5157], page 585,

<in> Ada.Strings.Wide_Maps [5368], page 641,

<in> Ada.Strings.Wide_Maps.Wide_Constants [5400], page 646,

<in> Interfaces.Fortran [6645], page 946,

characteristics [3271], page 285,

Characters

<child of> Ada [4857], page 565,

chars_ptr

<in> Interfaces.C.Strings [6534], page 916,

chars_ptr_array

<in> Interfaces.C.Strings [6535], page 916,
check
language–defined [4176], page 431, [4213], page 448,
check, language–defined
Access_Check [2299], page 180, [2810], page 227,
Accessibility_Check [2229], page 172, [2788], page 225, [2802], page 226, [2865],
page 232, [3204], page 274, [3207], page 274, [7141], page 1046,
Allocation_Check [2868], page 232, [2871], page 232,
Ceiling_Check [6691], page 955, [6873], page 992,
Discriminant_Check [2336], page 186, [2379], page 191, [2413], page 195, [2794],
page 226, [2796], page 226, [2806], page 227, [2817], page 227, [2836], page 230, [2862],
page 232,
Division_Check [1694], page 97, [2701], page 216, [5441], page 652, [5533], page 673,
[7230], page 1089, [7257], page 1094, [7486], page 1222,
Elaboration_Check [2262], page 176,
Index_Check [2311], page 181, [2320], page 183, [2449], page 199, [2451], page 199,
[2669], page 212, [2808], page 227, [2838], page 230, [2860], page 232,
Length_Check [2603], page 205, [2783], page 224, [2813], page 227,
Overflow_Check [1691], page 97, [2569], page 202, [2976], page 247, [7287], page 1105,
[7301], page 1108, [7305], page 1112, [7310], page 1113, [7317], page 1116,
Partition_Check [7144], page 1046,
Range_Check [1405], page 56, [1575], page 79, [1582], page 80, [1609], page 88, [1603],
page 86, [1606], page 87, [1615], page 89, [1621], page 90, [1713], page 100, [1794],
page 108, [2367], page 189, [2447], page 199, [2605], page 205, [2712], page 218, [2721],
page 219, [2773], page 223, [2785], page 225, [2798], page 226, [2804], page 227, [2834],
page 230, [4788], page 546, [5474], page 658, [5510], page 668, [5515], page 669, [5538],
page 673, [5543], page 674, [5548], page 675, [5553], page 676, [7461], page 1181, [7474],
page 1202, [7477], page 1204, [7483], page 1219, [7490], page 1226, [7493], page 1231,
[7466], page 1188, [7469], page 1189,
Reserved_Check [6687], page 955,
Storage_Check [4104], page 419, [4532], page 497, [4707], page 528, [6919], page 1006,
[6923], page 1006, [6927], page 1007,
Tag_Check [2097], page 148, [2792], page 225, [2815], page 227, [2949], page 243,
Checking pragmas [4174], page 431,
child
of a library unit [3914], page 395,

choice parameter [4121], page 420,
choice_parameter_specification [4116], page 420,
 <used> [4112], page 420, [8373], page 1307,
Circumflex
 <in> Ada.Characters.Latin_1 [4954], page 575,
class
 of types [1318], page 51, [1483], page 66,
 <See also> package [3228], page 279,
 <See also> tag [2024], page 137,
class (of types) [7678], page 1283,
Class attribute [2042], page 140, [3279], page 288, [7454], page 1176,
class factory [2058], page 143,
class-wide type [1516], page 72, [1925], page 126,
cleanup
 <See> finalization [3316], page 299,
clear
 execution timer object [7013], page 1025,
 group budget object [7039], page 1029,
 timing event object [7050], page 1032,
 <in> Ada.Containers.Doubly_Linked_Lists [6089], page 811,
 <in> Ada.Containers.Hashed_Maps [6159], page 840,
 <in> Ada.Containers.Hashed_Sets [6269], page 868,
 <in> Ada.Containers.Ordered_Maps [6202], page 847,
 <in> Ada.Containers.Ordered_Sets [6331], page 877,
 <in> Ada.Containers.Vectors [6010], page 782,
 <in> Ada.Environment_Variables [5980], page 775,
cleared
 termination handler [6790], page 972,
clock [3740], page 359,
 <in> Ada.Calendar [3747], page 360,
 <in> Ada.Execution_Time [6995], page 1022,
 <in> Ada.Real_Time [6942], page 1008,
clock jump [6957], page 1011,
clock tick [6956], page 1010,
Close
 <in> Ada.Direct_IO [5626], page 691,

- <in> Ada.Sequential_IO [5601], page 684,
- <in> Ada.Streams.Stream_IO [5869], page 747,
- <in> Ada.Text_IO [5679], page 699,
- close result set [7303], page 1110,
- closed entry [3693], page 353,
 - of a protected object [3698], page 353,
 - of a task [3696], page 353,
- closed under derivation [1509], page 70, [7679], page 1283,
- closure
 - downward [2240], page 174,
- COBOL
 - <child of> Interfaces [6581], page 932,
- COBOL interface [6580], page 931,
- COBOL standard [1096], page 30,
- COBOL_Character
 - <in> Interfaces.COBOL [6590], page 932,
- COBOL_To_Ada
 - <in> Interfaces.COBOL [6592], page 932,
- code_statement [4648], page 518,
 - <used> [2920], page 240, [8086], page 1298,
- coding
 - aspect of representation [4550], page 500,
- coextension
 - of an object [2217], page 167,
- Col
 - <in> Ada.Text_IO [5732], page 702,
- collection
 - finalization of [3331], page 301,
- colon [1163], page 36,
 - <in> Ada.Characters.Latin_1 [4944], page 575,
- column number [5662], page 697,
- comma [1155], page 36,
 - <in> Ada.Characters.Latin_1 [4939], page 574,
- Command_Line
 - <child of> Ada [5918], page 755,
- Command_Name

<in> Ada.Command_Line [5921], page 755,
comment [1251], page 43,
comments, instructions for submission [1004], page 13,
Commercial_At
<in> Ada.Characters.Latin_1 [4950], page 575,
Communication_Error
<in> System.RPC [7157], page 1050,
comparison operator
<See> relational operator [2609], page 206,
compatibility
 composite_constraint with an access subtype [2191], page 158,
 constraint with a subtype [1406], page 56,
 delta_constraint with an ordinary fixed point subtype [7428], page 1168,
 digits_constraint with a decimal fixed point subtype [1791], page 108,
 digits_constraint with a floating point subtype [7429], page 1168,
 discriminant constraint with a subtype [1942], page 128,
 index constraint with a subtype [1867], page 118,
 range with a scalar subtype [1551], page 77,
 range_constraint with a scalar subtype [1552], page 77,
compatible
 a type, with a convention [6433], page 895,
compilation [3915], page 395,
 separate [3909], page 394,
Compilation unit [3911], page 394, [3945], page 395, [7680], page 1283,
compilation units needed
 by a compilation unit [4037], page 410,
 remote call interface [7114], page 1043,
 shared passive library unit [7087], page 1039,
compilation_unit [3917], page 395,
 <used> [3916], page 395, [8331], page 1306,
compile-time error [1023], page 22, [1072], page 28,
compile-time semantics [1028], page 22,
complete context [3466], page 324,
completely defined [2269], page 177,
completion
 abnormal [3323], page 299,

compile–time concept [2265], page 177,
normal [3321], page 299,
run–time concept [3319], page 299,
completion and leaving (completed and left) [3318], page 299,
completion legality
 [<partial>] [2205], page 162,
 entry_body [3669], page 349,
Complex
 <in> Ada.Numerics.Generic_Complex_Types [7208], page 1084,
 <in> Interfaces.Fortran [6641], page 945,
Complex_Arrays
 <child of> Ada.Numerics [7371], page 1136,
Complex_Elementary_Functions
 <child of> Ada.Numerics [7255], page 1092,
Complex_IO
 <child of> Ada.Text_IO [7263], page 1097,
 <child of> Ada.Wide_Text_IO [7275], page 1103,
 <child of> Ada.Wide_Wide_Text_IO [7277], page 1103,
Complex_Matrix
 <in> Ada.Numerics.Generic_Complex_Arrays [7337], page 1130,
Complex_Text_IO
 <child of> Ada [7273], page 1098,
Complex_Types
 <child of> Ada.Numerics [7228], page 1087,
Complex_Vector
 <in> Ada.Numerics.Generic_Complex_Arrays [7336], page 1130,
component [1323], page 51,
component subtype [1839], page 115,
component_choice_list [2390], page 191,
 <used> [2387], page 191, [8002], page 1296,
component_clause [4564], page 502,
 <used> [4563], page 502, [8477], page 1310,
component_declaration [1964], page 130,
 <used> [1962], page 130, [3584], page 338, [8261], page 1303,
component_definition [1833], page 114,
 <used> [1823], page 114, [1829], page 114, [1966], page 130, [7882], page 1292,

component_item [1961], page 130,
 <used> [1958], page 130, [7911], page 1293,
component_list [1956], page 130,
 <used> [1955], page 130, [1991], page 134, [7909], page 1293,
Component_Size attribute [4535], page 497,
Component_Size clause [4479], page 487, [4537], page 497,
components
 of a record type [1969], page 130,
Compose
 <in> Ada.Directories [5942], page 759,
Compose attribute [5507], page 667,
Compose_From_Cartesian
 <in> Ada.Numerics.Generic_Complex_Arrays [7343], page 1131, [7355], page 1133,
 <in> Ada.Numerics.Generic_Complex_Types [7220], page 1085,
Compose_From_Polar
 <in> Ada.Numerics.Generic_Complex_Arrays [7347], page 1131, [7361], page 1133,
 <in> Ada.Numerics.Generic_Complex_Types [7225], page 1085,
composite type [1322], page 51, [7681], page 1283,
composite_constraint [1398], page 56,
 <used> [1393], page 56, [7821], page 1291,
compound delimiter [1180], page 37,
compound_statement [2921], page 240,
 <used> [2907], page 240, [8074], page 1297,
concatenation operator [2500], page 201, [2664], page 212,
concrete subprogram
 <See> nonabstract subprogram [2105], page 150,
concrete type
 <See> nonabstract type [2103], page 149,
concurrent processing
 <See> task [3494], page 328,
condition [2961], page 245,
 <used> [2956], page 245, [2984], page 249, [3004], page 252, [3659], page 348, [3818],
page 378, [8113], page 1299,
 <See also> exception [4087], page 419,
conditional_entry_call [3846], page 383,
 <used> [3808], page 378, [8299], page 1304,

configuration

of the partitions of a program [7056], page 1034,

configuration pragma [4015], page 408,

Assertion_Policy [4170], page 427,

Detect_Blocking [7415], page 1163,

Discard_Names [6716], page 962,

Locking_Policy [6865], page 991,

Normalize_Scalars [7382], page 1154,

Partition_Elaboration_Policy [7420], page 1164,

Priority_Specific_Dispatching [6833], page 981,

Profile [6986], page 1020,

Queuing_Policy [6884], page 995,

Restrictions [4751], page 535,

Reviewable [7387], page 1155,

Suppress [4187], page 431,

Task_Dispatching_Policy [6831], page 981,

Unsuppress [4189], page 431,

confirming

representation item [4446], page 484,

conformance [3102], page 263,

of an implementation with the Standard [1050], page 23,

<See also> full conformance, mode conformance, subtype conformance, type conformance

Conjugate

<in> Ada.Numerics.Generic_Complex_Arrays [7349], page 1131, [7362], page 1134,

<in> Ada.Numerics.Generic_Complex_Types [7226], page 1085, [7227], page 1086,

consistency

among compilation units [4005], page 406,

constant [1416], page 59,

result of a function_call [3162], page 268,

<See also> literal [2357], page 189,

<See also> static [2876], page 234,

constant object [1418], page 59,

constant view [1420], page 59,

Constants

<child of> Ada.Strings.Maps [5324], page 636,

constituent

of a construct [1068], page 27,
constrained [1345], page 52,
 object [1462], page 62,
 object [3171], page 272,
 subtype [1347], page 52, [1492], page 67, [1548], page 77, [1635], page 92, [1671],
page 96, [1674], page 96, [1745], page 104, [1783], page 108, [1787], page 108, [1847],
page 115, [1850], page 115, [1921], page 125, [2044], page 140,
 subtype [2189], page 158,
 subtype [7463], page 1186,
Constrained attribute [1947], page 129, [7432], page 1169,
constrained by its initial value [1459], page 62,
 [<partial>] [2852], page 232,
constrained_array_definition [1826], page 114,
 <used> [1819], page 114, [7875], page 1292,
constraint [1391], page 56,
 [<partial>] [1335], page 51,
 of a first array subtype [1852], page 115,
 of a subtype [1340], page 52,
 of an object [1458], page 62,
 <used> [1388], page 56, [7818], page 1290,
Constraint_Error
 raised by failure of run-time check [1407], page 56, [1571], page 79, [1578], page 80,
[1607], page 88, [1601], page 86, [1604], page 87, [1613], page 89, [1619], page 90, [1692],
page 97, [1711], page 100, [1795], page 108, [2098], page 148, [2300], page 180, [2312],
page 181, [2322], page 183, [2337], page 186, [2368], page 190, [2380], page 191, [2414],
page 195, [2452], page 199, [2570], page 202, [2585], page 203, [2586], page 204, [2587],
page 204, [2606], page 205, [2670], page 212, [2702], page 216, [2713], page 218, [2719],
page 218, [2722], page 219, [2774], page 223, [2821], page 228, [2824], page 228, [2840],
page 230, [2863], page 232, [2950], page 243, [2977], page 247, [4098], page 419, [4161],
page 426, [4192], page 432, [4673], page 523, [4789], page 547, [5215], page 610, [5214],
page 604, [5369], page 641, [5402], page 646, [5442], page 652, [5443], page 652, [5475],
page 658, [5477], page 659, [5508], page 668, [5513], page 668, [5531], page 673, [5536],
page 673, [5541], page 674, [5546], page 675, [5551], page 676, [5926], page 756, [6530],
page 909, [6531], page 909, [6632], page 936, [7145], page 1046, [7231], page 1089, [7258],
page 1094, [7288], page 1105, [7297], page 1108, [7306], page 1112, [7311], page 1113,
[7318], page 1116, [7459], page 1180, [7472], page 1202, [7475], page 1203, [7479],

page 1218, [7484], page 1222, [7488], page 1226, [7491], page 1231, [7497], page 1235, [7464], page 1187, [7467], page 1188,

<in> Standard [4851], page 563,

Construct [1067], page 27, [7682], page 1283,

constructor

<See> initialization [1471], page 63,

<See> initialization [3292], page 295,

<See> initialization expression [1453], page 61,

<See> Initialize [3293], page 295,

<See> initialized allocator [2850], page 231,

container

cursor [5986], page 778,

list [6079], page 810,

map [6137], page 830,

set [6244], page 855,

vector [5992], page 779,

Containers

<child of> Ada [5989], page 779,

Containing_Directory

<in> Ada.Directories [5939], page 758,

Contains

<in> Ada.Containers.Doubly_Linked_Lists [6119], page 814,

<in> Ada.Containers.Hashed_Maps [6179], page 843,

<in> Ada.Containers.Hashed_Sets [6295], page 871, [6308], page 872,

<in> Ada.Containers.Ordered_Maps [6233], page 850,

<in> Ada.Containers.Ordered_Sets [6366], page 880, [6380], page 882,

<in> Ada.Containers.Vectors [6057], page 787,

context free grammar

complete listing [7737], page 1289,

cross reference [8498], page 1311,

notation [1062], page 25,

under Syntax heading [1016], page 22,

context_clause [3961], page 399,

<used> [3920], page 395, [8334], page 1306,

context_item [3963], page 400,

<used> [3962], page 399, [8349], page 1306,

contiguous representation

[<partial>] [4587], page 508, [4640], page 517, [4659], page 521, [4662], page 521, [4703], page 528,

Continue

<in> Ada.Asynchronous_Task_Control [6974], page 1016,

control character

a category of Character [4883], page 568,

a category of Character [4894], page 573, [4988], page 576,

<See also> format_effector [1140], page 35,

Control_Set

<in> Ada.Strings.Maps.Constants [5325], page 636,

controlled

aspect of representation [4737], page 534,

<in> Ada.Finalization [3300], page 296,

Controlled pragma [4732], page 534, [7525], page 1242,

controlled type [3295], page 295, [3307], page 296, [7683], page 1284,

controlling formal parameter [2086], page 146,

controlling operand [2085], page 146,

controlling result [2087], page 146,

controlling tag

for a call on a dispatching operation [2077], page 145,

controlling tag value [2094], page 147,

for the expression in an assignment_statement [2947], page 243,

controlling type

of a formal_abstract_subprogram_declaration [4396], page 472,

convention [3107], page 263, [6431], page 895,

aspect of representation [6445], page 897,

Convention pragma [6423], page 895, [7528], page 1242,

conversion [2725], page 219, [2771], page 223,

access [2756], page 221, [2762], page 222, [2764], page 222, [2800], page 226,

arbitrary order [1071], page 27,

array [2751], page 220, [2781], page 224,

composite (non-array) [2745], page 220, [2790], page 225,

enumeration [2747], page 220, [2779], page 224,

numeric [2749], page 220, [2776], page 223,

unchecked [4655], page 520,

- value [2741], page 219,
- view [2739], page 219,

Conversion_Error

- <in> Interfaces.COBOL [6614], page 934,

Conversions

- <child of> Ada.Characters [5124], page 580,

convertible [2737], page 219,

- required [1914], page 124, [1941], page 128, [2757], page 221, [2752], page 220, [3166], page 270,

copy back of parameters [3174], page 272,

copy parameter passing [3080], page 260,

Copy_Array

- <in> Interfaces.C.Pointers [6563], page 924,

Copy_File

- <in> Ada.Directories [5936], page 758,

Copy_Sign attribute [5540], page 674,

Copy_Terminated_Array

- <in> Interfaces.C.Pointers [6562], page 924,

Copyright_Sign

- <in> Ada.Characters.Latin_1 [5035], page 577,

core language [1005], page 20,

corresponding constraint [1494], page 67,

corresponding discriminants [1915], page 125,

corresponding index

- for an array_aggregate [2440], page 197,

corresponding subtype [1503], page 69,

corresponding value

- of the target type of a conversion [2770], page 223,

Cos

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7240], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5417], page 649,

Cosh

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7248], page 1092,
- <in> Ada.Numerics.Generic_Elementary_Functions [5432], page 650,

Cot

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7242], page 1092,

<in> Ada.Numerics.Generic_Elementary_Functions [5421], page 649,

Coth

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7250], page 1092,

<in> Ada.Numerics.Generic_Elementary_Functions [5434], page 650,

Count

<in> Ada.Direct_IO [5622], page 691,

<in> Ada.Streams.Stream_IO [5865], page 746,

<in> Ada.Strings.Bounded [5247], page 617, [5248], page 617, [5249], page 617,

<in> Ada.Strings.Fixed [5190], page 594, [5191], page 594, [5192], page 594,

<in> Ada.Strings.Unbounded [5300], page 630, [5301], page 630, [5302], page 630,

<in> Ada.Text_IO [5671], page 698,

Count attribute [3898], page 389,

cover

 a type [1521], page 75,

 of a choice and an exception [4120], page 420,

cover a value

 by a discrete_choice [2000], page 134,

 by a discrete_choice_list [2001], page 135,

CPU clock tick [7000], page 1023,

CPU time

 of a task [6999], page 1022,

CPU_Tick

<in> Ada.Execution_Time [6994], page 1022,

CPU_Time

<in> Ada.Execution_Time [6990], page 1021,

CPU_Time_First

<in> Ada.Execution_Time [6991], page 1022,

CPU_Time_Last

<in> Ada.Execution_Time [6992], page 1022,

CPU_Time_Unit

<in> Ada.Execution_Time [6993], page 1022,

CR

<in> Ada.Characters.Latin_1 [4908], page 573,

create [1313], page 50,

<in> Ada.Direct_IO [5624], page 691,

<in> Ada.Sequential_IO [5599], page 683,

- <in> Ada.Streams.Stream_IO [5867], page 746,
- <in> Ada.Text_IO [5677], page 699,
- Create_Directory
 - <in> Ada.Directories [5930], page 758,
- Create_Path
 - <in> Ada.Directories [5932], page 758,
- creation
 - of a protected object [6684], page 955,
 - of a return object [3198], page 274,
 - of a tag [4830], page 552,
 - of a task object [6806], page 976,
 - of an object [1414], page 58,
- critical section
 - <See> intertask communication [3615], page 343,
- CSI
 - <in> Ada.Characters.Latin_1 [5020], page 577,
- Currency_Sign
 - <in> Ada.Characters.Latin_1 [5030], page 577,
- current column number [5663], page 697,
- current index
 - of an open direct file [5595], page 683,
 - of an open stream file [5859], page 746,
- current instance
 - of a generic unit [3476], page 325,
 - of a type [3475], page 325,
- current line number [5664], page 697,
- current mode
 - of an open file [5585], page 681,
- current page number [5665], page 697,
- Current size
 - of a stream file [5860], page 746,
 - of an external file [5594], page 683,
- Current_Directory
 - <in> Ada.Directories [5928], page 757,
- Current_Error
 - <in> Ada.Text_IO [5695], page 700, [5702], page 700,

Current_Handler

- <in> Ada.Execution_Time.Group_Budgets [7032], page 1028,
- <in> Ada.Execution_Time.Timers [7008], page 1025,
- <in> Ada.Interrupts [6702], page 957,
- <in> Ada.Real_Time.Timing_Events [7046], page 1031,

Current_Input

- <in> Ada.Text_IO [5693], page 700, [5700], page 700,

Current_Output

- <in> Ada.Text_IO [5694], page 700, [5701], page 700,

Current_State

- <in> Ada.Synchronous_Task_Control [6967], page 1015,

Current_Task

- <in> Ada.Task_Identification [6750], page 965,

Current_Task_Fallback_Handler

- <in> Ada.Task_Termination [6780], page 972,

cursor

- ambiguous [6072], page 808,
- for a container [5985], page 778,
- invalid [6075], page 809, [6133], page 829, [6147], page 839, [6255], page 866,
- <in> Ada.Containers.Doubly_Linked_Lists [6084], page 811,
- <in> Ada.Containers.Hashed_Maps [6152], page 840,
- <in> Ada.Containers.Hashed_Sets [6260], page 867,
- <in> Ada.Containers.Ordered_Maps [6197], page 846,
- <in> Ada.Containers.Ordered_Sets [6324], page 876,
- <in> Ada.Containers.Vectors [6000], page 780,

30.5 D

dangling references

- prevention via accessibility rules [2213], page 164,

Data_Error

- <in> Ada.Direct_IO [5648], page 692,
- <in> Ada.IO_Exceptions [5912], page 752,

<in> Ada.Sequential_IO [5618], page 684,
<in> Ada.Storage_IO [5654], page 696,
<in> Ada.Streams.Stream_IO [5894], page 748,
<in> Ada.Text_IO [5820], page 708,
date and time formatting standard [1101], page 30,
Day
<in> Ada.Calendar [3750], page 360,
<in> Ada.Calendar.Formatting [3789], page 366,
Day_Count
<in> Ada.Calendar.Arithmetic [3770], page 364,
Day_Duration <subtype of> Duration
<in> Ada.Calendar [3746], page 360,
Day_Name
<in> Ada.Calendar.Formatting [3774], page 365,
Day_Number <subtype of> Integer
<in> Ada.Calendar [3745], page 360,
Day_of_Week
<in> Ada.Calendar.Formatting [3782], page 366,
DC1
<in> Ada.Characters.Latin_1 [4912], page 574,
DC2
<in> Ada.Characters.Latin_1 [4913], page 574,
DC3
<in> Ada.Characters.Latin_1 [4914], page 574,
DC4
<in> Ada.Characters.Latin_1 [4915], page 574,
DCS
<in> Ada.Characters.Latin_1 [5009], page 576,
Deadline <subtype of> Time
<in> Ada.Dispatching.EDF [6853], page 988,
Deallocate
<in> System.Storage_Pools [4688], page 527,
deallocation of storage [4716], page 532,
Decimal
<child of> Ada [7172], page 1055,
decimal digit

- a category of Character [4889], page 569,
- decimal fixed point type [1759], page 106, [1777], page 107,
- Decimal_Conversions
 - <in> Interfaces.COBOL [6615], page 934,
- Decimal_Digit_Set
 - <in> Ada.Strings.Maps.Constants [5331], page 636,
- Decimal_Element
 - <in> Interfaces.COBOL [6588], page 932,
- decimal_fixed_point_definition [1767], page 107,
 - <used> [1763], page 106, [7866], page 1292,
- Decimal_IO
 - <in> Ada.Text_IO [5795], page 707,
- decimal_literal [1206], page 40,
 - <used> [1203], page 40, [7754], page 1289,
- Decimal_Output
 - <in> Ada.Text_IO.Editing [7195], page 1074,
- Declaration [1297], page 49, [1300], page 49, [7684], page 1284,
- declaration list
 - declarative_part [2258], page 176,
 - package_specification [3241], page 279,
- declarative region
 - of a construct [3343], page 304,
- declarative_item [2243], page 175,
 - <used> [2242], page 175, [7950], page 1295,
- declarative_part [2241], page 175,
 - <used> [2998], page 251, [3096], page 261, [3245], page 281, [3526], page 330, [3652], page 348, [8190], page 1301,
- declare [1306], page 50, [1312], page 50,
- declared pure [4071], page 417,
- Decrement
 - <in> Interfaces.C.Pointers [6560], page 923,
- deeper
 - accessibility level [2211], page 164,
 - statically [2216], page 164, [2219], page 168,
- default directory [5972], page 761,
- default entry queuing policy [3708], page 354,

default treatment [6676], page 951,

Default_Aft

<in> Ada.Text_IO [5777], page 705, [5787], page 706, [5797], page 707,

<in> Ada.Text_IO.Complex_IO [7265], page 1098,

Default_Base

<in> Ada.Text_IO [5759], page 703, [5768], page 704,

Default_Bit_Order

<in> System [4625], page 512,

Default_Currency

<in> Ada.Text_IO.Editing [7191], page 1073,

Default_Deadline

<in> Ada.Dispatching.EDF [6854], page 988,

Default_Exp

<in> Ada.Text_IO [5778], page 705, [5788], page 706, [5798], page 707,

<in> Ada.Text_IO.Complex_IO [7266], page 1098,

default_expression [1908], page 123,

<used> [1907], page 123, [1967], page 130, [3062], page 256, [4309], page 458, [7901],
page 1293,

Default_Fill

<in> Ada.Text_IO.Editing [7192], page 1073,

Default_Fore

<in> Ada.Text_IO [5776], page 705, [5786], page 706, [5796], page 707,

<in> Ada.Text_IO.Complex_IO [7264], page 1098,

default_name [4389], page 471,

<used> [4388], page 471, [8448], page 1309,

Default_Priority

<in> System [4629], page 512,

Default_Quantum

<in> Ada.Dispatching.Round_Robin [6844], page 986,

Default_Radix_Mark

<in> Ada.Text_IO.Editing [7194], page 1073,

Default_Separator

<in> Ada.Text_IO.Editing [7193], page 1073,

Default_Setting

<in> Ada.Text_IO [5807], page 708,

Default_Width

<in> Ada.Text_IO [5758], page 703, [5767], page 704, [5806], page 708,
deferred constant [3281], page 290,
deferred constant declaration [1455], page 61, [3280], page 290,
defining name [1307], page 50,
defining_character_literal [1629], page 92,
 <used> [1628], page 92, [7852], page 1291,
defining_designator [3033], page 255,
 <used> [3027], page 255, [4266], page 455, [8401], page 1308,
defining_identifier [1295], page 49,
 <used> [1357], page 53, [1383], page 55, [1449], page 61, [1627], page 92, [2196], page 160,
[2987], page 249, [3038], page 256, [3183], page 272, [3257], page 283, [3260], page 283,
[3430], page 317, [3437], page 318, [3509], page 329, [3514], page 329, [3525], page 330,
[3566], page 338, [3571], page 338, [3586], page 338, [3637], page 347, [3649], page 348,
[3661], page 348, [3989], page 403, [3991], page 403, [3993], page 403, [4117], page 420,
[4325], page 460, [4400], page 475, [8450], page 1309,
defining_identifier_list [1448], page 61,
 <used> [1440], page 61, [1476], page 65, [1905], page 123, [1965], page 130, [3055],
page 256, [4096], page 419, [4306], page 458, [7916], page 1293,
defining_operator_symbol [3041], page 256,
 <used> [3035], page 255, [8136], page 1299,
defining_program_unit_name [3036], page 256,
 <used> [3024], page 255, [3034], page 255, [3232], page 279, [3244], page 281, [3440],
page 319, [3459], page 323, [4262], page 455, [8220], page 1302,
Definite attribute [4358], page 465,
definite subtype [1428], page 60,
definition [1302], page 50,
Degree_Sign
 <in> Ada.Characters.Latin_1 [5042], page 577,
DEL
 <in> Ada.Characters.Latin_1 [4987], page 576,
delay_alternative [3826], page 379,
 <used> [3821], page 378, [3838], page 381, [8308], page 1305,
delay_relative_statement [3734], page 359,
 <used> [3731], page 359, [8294], page 1304,
delay_statement [3729], page 359,
 <used> [2917], page 240, [3827], page 379, [3858], page 384, [8327], page 1306,

Delay_Until_And_Set_Deadline

<in> Ada.Dispatching.EDF [6856], page 988,

delay_until_statement [3732], page 359,

<used> [3730], page 359, [8293], page 1304,

Delete

<in> Ada.Containers.Doubly_Linked_Lists [6100], page 813,

<in> Ada.Containers.Hashed_Maps [6172], page 842, [6173], page 842,

<in> Ada.Containers.Hashed_Sets [6279], page 869, [6280], page 869, [6306], page 872,

<in> Ada.Containers.Ordered_Maps [6215], page 848, [6216], page 848,

<in> Ada.Containers.Ordered_Sets [6341], page 878, [6342], page 878, [6376], page 882,

<in> Ada.Containers.Vectors [6036], page 785, [6037], page 785,

<in> Ada.Direct_IO [5627], page 691,

<in> Ada.Sequential_IO [5602], page 684,

<in> Ada.Streams.Stream_IO [5870], page 747,

<in> Ada.Strings.Bounded [5261], page 619, [5262], page 619,

<in> Ada.Strings.Fixed [5204], page 596, [5205], page 596,

<in> Ada.Strings.Unbounded [5314], page 632, [5315], page 632,

<in> Ada.Text_IO [5680], page 699,

Delete_Directory

<in> Ada.Directories [5931], page 758,

Delete_File

<in> Ada.Directories [5934], page 758,

Delete_First

<in> Ada.Containers.Doubly_Linked_Lists [6101], page 813,

<in> Ada.Containers.Ordered_Maps [6217], page 849,

<in> Ada.Containers.Ordered_Sets [6343], page 878,

<in> Ada.Containers.Vectors [6038], page 785,

Delete_Last

<in> Ada.Containers.Doubly_Linked_Lists [6102], page 813,

<in> Ada.Containers.Ordered_Maps [6218], page 849,

<in> Ada.Containers.Ordered_Sets [6344], page 878,

<in> Ada.Containers.Vectors [6039], page 785,

Delete_Tree

<in> Ada.Directories [5933], page 758,

delimiter [1179], page 37,

delivery

- of an interrupt [6669], page 951,
- delta
 - of a fixed point type [1760], page 106,
- Delta attribute [1801], page 110,
- delta_constraint [7423], page 1168,
 - <used> [1397], page 56, [7824], page 1291,
- Denorm attribute [5489], page 664,
- denormalized number [5490], page 665,
- denote [3474], page 325,
 - informal definition [1305], page 50,
 - name used as a pragma argument [3488], page 327,
- depend on a discriminant
 - for a component [1918], page 125,
 - for a constraint or component_definition [1917], page 125,
- dependence
 - elaboration [4042], page 410,
 - of a task on a master [3551], page 335,
 - of a task on another task [3555], page 335,
 - semantic [3960], page 398,
- depth
 - accessibility level [2212], page 164,
- dereference [2290], page 179,
- Dereference_Error
 - <in> Interfaces.C.Strings [6541], page 916,
- derivation class
 - for a type [1512], page 72,
- derived from
 - directly or indirectly [1511], page 72,
- derived type [1481], page 66, [7687], page 1284,
 - [<partial>] [1505], page 70,
- derived_type_definition [1485], page 66,
 - <used> [1369], page 54, [7812], page 1290,
- descendant [3951], page 396, [7688], page 1284,
 - of a type [1522], page 75,
 - relationship with scope [3352], page 306,
- Descendant_Tag

<in> Ada.Tags [2034], page 138,
designate [2144], page 156,
designated profile
 of an access-to-subprogram type [2181], page 157,
 of an anonymous access type [2186], page 158,
designated subtype
 of a named access type [2176], page 157,
 of an anonymous access type [2184], page 157,
designated type
 of a named access type [2177], page 157,
 of an anonymous access type [2185], page 157,
designator [3029], page 255,
 <used> [3098], page 261, [8161], page 1300,
destructor
 <See> finalization [3294], page 295,
 <See> finalization [3317], page 299,
Detach_Handler
 <in> Ada.Interrupts [6705], page 958,
Detect_Blocking pragma [7414], page 1163, [7532], page 1242,
Determinant
 <in> Ada.Numerics.Generic_Complex_Arrays [7367], page 1135,
 <in> Ada.Numerics.Generic_Real_Arrays [7327], page 1121,
determined category for a formal type [4348], page 461,
determines
 a type by a subtype_mark [1401], page 56,
Device_Error
 <in> Ada.Direct_IO [5646], page 692,
 <in> Ada.Directories [5964], page 761,
 <in> Ada.IO_Exceptions [5910], page 752,
 <in> Ada.Sequential_IO [5616], page 684,
 <in> Ada.Streams.Stream_IO [5892], page 748,
 <in> Ada.Text_IO [5818], page 708,
Diaeresis
 <in> Ada.Characters.Latin_1 [5034], page 577,
Difference
 <in> Ada.Calendar.Arithmetic [3772], page 365,

<in> Ada.Containers.Hashed_Sets [6285], page 870, [6286], page 870,
<in> Ada.Containers.Ordered_Sets [6349], page 879, [6350], page 879,
digit [1217], page 40,
<used> [1211], page 40, [1240], page 41, [7761], page 1289,
digits
of a decimal fixed point subtype [1776], page 107, [1808], page 111,
Digits attribute [1755], page 105, [1807], page 111,
digits_constraint [1771], page 107,
<used> [1396], page 56, [7823], page 1291,
dimensionality
of an array [1840], page 115,
direct access [5592], page 683,
direct file [5589], page 682,
Direct_IO
<child of> Ada [5619], page 691,
direct_name [2280], page 179,
<used> [1986], page 134, [2271], page 179, [2933], page 241, [3641], page 347, [4058],
page 414, [4424], page 481, [7434], page 1171, [7601], page 1243, [7919], page 1293,
Direction
<in> Ada.Strings [5154], page 585,
directly specified
of an aspect of representation of an entity [4433], page 482,
of an operational aspect of an entity [4439], page 482,
directly visible [3364], page 308, [3387], page 311,
within a pragma in a context_clause [4023], page 409,
within a pragma that appears at the place of a compilation unit [4027], page 409,
within a use_clause in a context_clause [4021], page 409,
within a with_clause [4019], page 409,
within the parent_unit_name of a library unit [4017], page 409,
within the parent_unit_name of a subunit [4025], page 409,
Directories
<child of> Ada [5927], page 757,
directory [5965], page 761,
directory entry [5973], page 761,
directory name [5968], page 761,
Directory_Entry_Type

<in> Ada.Directories [5949], page 759,
Discard_Names pragma [6714], page 962, [7534], page 1242,
discontiguous representation
 [<partial>] [4588], page 508, [4641], page 517, [4660], page 521, [4663], page 521, [4704],
 page 528,
discrete array type [2638], page 206,
discrete type [1326], page 51, [1528], page 76, [7689], page 1284,
discrete_choice [1995], page 134,
 <used> [1993], page 134, [7925], page 1294,
discrete_choice_list [1992], page 134,
 <used> [1990], page 134, [2433], page 196, [2970], page 246, [8107], page 1298,
Discrete_Random
 <child of> Ada.Numerics [5459], page 656,
discrete_range [1861], page 117,
 <used> [1860], page 117, [1997], page 134, [2316], page 182, [7927], page 1294,
discrete_subtype_definition [1830], page 114,
 <used> [1828], page 114, [2988], page 249, [3638], page 347, [3662], page 348, [7881],
 page 1292,
discriminant [1332], page 51, [1890], page 123, [7690], page 1284,
 of a variant_part [1998], page 134,
 use in a record definition [1970], page 131,
discriminant_association [1931], page 127,
 <used> [1930], page 127, [7904], page 1293,
Discriminant_Check [4194], page 433,
 [<partial>] [2335], page 186, [2378], page 191, [2412], page 195, [2793], page 226, [2795],
 page 226, [2805], page 227, [2816], page 227, [2835], page 230, [2861], page 232,
discriminant_constraint [1928], page 127,
 <used> [1400], page 56, [7826], page 1291,
discriminant_part [1893], page 123,
 <used> [2197], page 160, [3258], page 283, [3261], page 283, [4326], page 460, [7949],
 page 1294,
discriminant_specification [1900], page 123,
 <used> [1898], page 123, [7894], page 1293,
discriminants
 known [1920], page 125,
 unknown [1924], page 125,

discriminated type [1911], page 124,
dispatching [2020], page 137,
 <child of> Ada [6809], page 978,
dispatching call
 on a dispatching operation [2071], page 145,
dispatching operation [2070], page 145, [2084], page 146,
 [<partial>] [2005], page 136,
dispatching point [6814], page 978,
 [<partial>] [6838], page 983, [6842], page 985,
dispatching policy for tasks
 [<partial>] [6820], page 979,
dispatching, task [6812], page 978,
Dispatching_Policy_Error
 <in> Ada.Dispatching [6810], page 978,
Display_Format
 <in> Interfaces.COBOL [6599], page 933,
displayed magnitude (of a decimal value) [7182], page 1066,
disruption of an assignment [3889], page 387, [4667], page 522,
 [<partial>] [4225], page 449,
distinct access paths [3088], page 261,
distributed program [7055], page 1034,
distributed system [7054], page 1034,
distributed systems [6657], page 949,
divide [1162], page 36,
 <in> Ada.Decimal [7178], page 1055,
divide operator [2511], page 201, [2693], page 213,
Division_Check [4195], page 434,
 [<partial>] [1693], page 97, [2700], page 216, [5440], page 652, [5532], page 673, [7229],
page 1089, [7256], page 1094, [7485], page 1222,
Division_Sign
 <in> Ada.Characters.Latin_1 [5115], page 579,
DLE
 <in> Ada.Characters.Latin_1 [4911], page 574,
Do_APC
 <in> System.RPC [7162], page 1051,
Do_RPC

<in> System.RPC [7161], page 1051,
documentation (required of an implementation) [1058], page 25, [7664], page 1245, [7666],
page 1250, [7669], page 1267,
documentation requirements [1044], page 22, [7662], page 1245,
summary of requirements [7663], page 1245,
Dollar_Sign
<in> Ada.Characters.Latin_1 [4931], page 574,
dot [1159], page 36,
dot selection
<See> selected_component [2323], page 183,
double
<in> Interfaces.C [6487], page 903,
Double_Precision
<in> Interfaces.Fortran [6638], page 945,
Doubly_Linked_Lists
<child of> Ada.Containers [6082], page 811,
downward closure [2239], page 174,
drift rate [6958], page 1012,
Duration
<in> Standard [4850], page 563,
dynamic binding
<See> dispatching operation [2007], page 136,
dynamic semantics [1035], page 22,
Dynamic_Priorities
<child of> Ada [6888], page 996,
dynamically determined tag [2074], page 145,
dynamically enclosing
of one execution by another [4132], page 422,
dynamically tagged [2089], page 146,

30.6 E

- <in> Ada.Numerics [5409], page 648,

EDF

- <child of> Ada.Dispatching [6852], page 988,

edited output [7179], page 1057,

Editing

- <child of> Ada.Text_IO [7183], page 1073,
- <child of> Ada.Wide_Text_IO [7203], page 1081,
- <child of> Ada.Wide_Wide_Text_IO [7205], page 1081,

effect

- external [1052], page 24,

efficiency [4211], page 448, [4217], page 448,

Eigensystem

- <in> Ada.Numerics.Generic_Complex_Arrays [7369], page 1136,
- <in> Ada.Numerics.Generic_Real_Arrays [7329], page 1122,

Eigenvalues

- <in> Ada.Numerics.Generic_Complex_Arrays [7368], page 1136,
- <in> Ada.Numerics.Generic_Real_Arrays [7328], page 1122,

Elaborate pragma [4073], page 418, [7537], page 1242,

Elaborate_All pragma [4077], page 418, [7541], page 1242,

Elaborate_Body pragma [4081], page 418, [7545], page 1242,

elaborated [2260], page 176,

elaboration [1310], page 50, [7691], page 1284, [7699], page 1285,

- abstract_subprogram_declaration [2114], page 151,
- access_definition [2194], page 159,
- access_type_definition [2193], page 158,
- array_type_definition [1855], page 116,
- aspect_clause [4447], page 484,
- choice_parameter_specification [4139], page 423,
- component_declaration [1978], page 131,
- component_definition [1857], page 116, [1982], page 132,
- component_list [1977], page 131,
- declaration named by a pragma Import [6458], page 898,
- declarative_part [2259], page 176,
- deferred constant declaration [3284], page 291,
- delta_constraint [7430], page 1168,
- derived_type_definition [1506], page 70,

digits_constraint [1792], page 108,
discrete_subtype_definition [1856], page 116,
discriminant_constraint [1944], page 128,
entry_declaration [3678], page 350,
enumeration_type_definition [1634], page 92,
exception_declaration [4102], page 419,
fixed_point_definition [1790], page 108,
floating_point_definition [1748], page 104,
full type definition [1381], page 55,
full_type_declaration [1380], page 55,
generic body [4255], page 453,
generic_declaration [4253], page 451,
generic_instantiation [4296], page 457,
incomplete_type_declaration [2204], page 162,
index_constraint [1870], page 118,
integer_type_definition [1689], page 97,
loop_parameter_specification [2994], page 249,
non-generic subprogram_body [3100], page 262,
nongeneric package_body [3250], page 282,
null_procedure_declaration [3223], page 277,
number_declaration [1480], page 66,
object_declaration [1466], page 62,
of library units for a foreign language main subprogram [6463], page 898,
package_body of Standard [4855], page 564,
package_declaration [3242], page 280,
partition [7060], page 1035,
partition [7165], page 1052,
per-object constraint [1983], page 132,
pragma [1270], page 45,
private_extension_declaration [3273], page 286,
private_type_declaration [3272], page 286,
protected declaration [3605], page 340,
protected_body [3609], page 341,
protected_definition [3607], page 340,
range_constraint [1553], page 77,
real_type_definition [1724], page 102,

record_definition [1976], page 131,
record_extension_part [2069], page 144,
record_type_definition [1975], page 131,
renaming_declaration [3422], page 316,
single_protected_declaration [3606], page 340,
single_task_declaration [3538], page 331,
Storage_Size pragma [4530], page 497,
subprogram_declaration [3077], page 258,
subtype_declaration [1402], page 56,
subtype_indication [1403], page 56,
task_declaration [3537], page 331,
task_body [3541], page 331,
task_definition [3539], page 331,
use_clause [3415], page 315,
variant_part [2004], page 136,
elaboration control [4050], page 413,
elaboration dependence
 library_item on another [4041], page 410,
Elaboration_Check [4204], page 444,
 [<partial>] [2261], page 176,
element
 of a storage pool [4694], page 527,
 <in> Ada.Containers.Doubly_Linked_Lists [6090], page 811,
 <in> Ada.Containers.Hashed_Maps [6161], page 841, [6178], page 842,
 <in> Ada.Containers.Hashed_Sets [6270], page 868, [6303], page 872,
 <in> Ada.Containers.Ordered_Maps [6204], page 847, [6230], page 850,
 <in> Ada.Containers.Ordered_Sets [6332], page 877, [6373], page 882,
 <in> Ada.Containers.Vectors [6013], page 782, [6014], page 782,
 <in> Ada.Strings.Bounded [5234], page 613,
 <in> Ada.Strings.Unbounded [5287], page 627,
elementary type [1321], page 51, [7692], page 1284,
Elementary_Functions
 <child of> Ada.Numerics [5439], page 650,
eligible
 a type, for a convention [6434], page 895,
else part

of a selective_accept [3830], page 379,

EM

<in> Ada.Characters.Latin_1 [4920], page 574,

embedded systems [6656], page 949, [6792], page 974,

empty element

of a vector [5995], page 780,

Empty_List

<in> Ada.Containers.Doubly_Linked_Lists [6085], page 811,

Empty_Map

<in> Ada.Containers.Hashed_Maps [6153], page 840,

<in> Ada.Containers.Ordered_Maps [6198], page 847,

Empty_Set

<in> Ada.Containers.Hashed_Sets [6261], page 867,

<in> Ada.Containers.Ordered_Sets [6325], page 876,

Empty_Vector

<in> Ada.Containers.Vectors [6001], page 780,

encapsulation

<See> package [3226], page 279,

enclosing

immediately [3348], page 305,

end of a line [1177], page 36,

End_Error

raised by failure of run-time check [4791], page 547,

<in> Ada.Direct_IO [5647], page 692,

<in> Ada.IO_Exceptions [5911], page 752,

<in> Ada.Sequential_IO [5617], page 684,

<in> Ada.Streams.Stream_IO [5893], page 748,

<in> Ada.Text_IO [5819], page 708,

End_Of_File

<in> Ada.Direct_IO [5641], page 692,

<in> Ada.Sequential_IO [5611], page 684,

<in> Ada.Streams.Stream_IO [5877], page 747,

<in> Ada.Text_IO [5725], page 701,

End_Of_Line

<in> Ada.Text_IO [5717], page 701,

End_Of_Page

<in> Ada.Text_IO [5724], page 701,
End_Search
<in> Ada.Directories [5953], page 760,
endian
 big [4592], page 508,
 little [4595], page 508,
ENQ
<in> Ada.Characters.Latin_1 [4900], page 573,
entity
 [<partial>] [1280], page 49,
entry
 closed [3694], page 353,
 open [3692], page 353,
 single [3676], page 350,
entry call [3685], page 352,
 simple [3687], page 352,
entry calling convention [3115], page 264,
entry family [3673], page 350,
entry index subtype [1981], page 131, [3674], page 350,
entry queue [3703], page 354,
entry queuing policy [3707], page 354,
 default policy [3709], page 354,
entry_barrier [3658], page 348,
 <used> [3651], page 348, [8281], page 1304,
entry_body [3648], page 348,
 <used> [3592], page 338, [8267], page 1303,
entry_body_formal_part [3655], page 348,
 <used> [3650], page 348, [8280], page 1304,
entry_call_alternative [3839], page 381,
 <used> [3837], page 381, [3847], page 383, [8314], page 1305,
entry_call_statement [3688], page 352,
 <used> [2915], page 240, [3844], page 381, [8081], page 1298,
entry_declaration [3635], page 347,
 <used> [3522], page 329, [3580], page 338, [8258], page 1303,
entry_index [3646], page 348,
 <used> [3642], page 347, [8274], page 1303,

entry_index_specification [3660], page 348,
 <used> [3656], page 348, [8285], page 1304,
enumeration_literal [1631], page 92,
enumeration_type [1327], page 51, [1622], page 92, [7693], page 1284,
enumeration_aggregate [4546], page 500,
 <used> [4545], page 500, [8473], page 1310,

Enumeration_IO

 <in> Ada.Text_IO [5805], page 708,
enumeration_literal_specification [1626], page 92,
 <used> [1624], page 92, [7850], page 1291,
enumeration_representation_clause [4543], page 500,
 <used> [4419], page 481, [8459], page 1310,
enumeration_type_definition [1623], page 92,
 <used> [1363], page 54, [7806], page 1290,
environment [4002], page 406,
environment_declarative_part [4003], page 406,
 for the environment task of a partition [4043], page 411,
environment_task [4040], page 410,
environment_variable [5975], page 775,

Environment_Variables

 <child of> Ada [5976], page 775,

EOT

 <in> Ada.Characters.Latin_1 [4899], page 573,

EPA

 <in> Ada.Characters.Latin_1 [5016], page 576,

epoch [6954], page 1010,
equal_operator [2464], page 201, [2616], page 206,
equality_operator [2610], page 206,
 special inheritance rule for tagged types [1501], page 68, [2645], page 208,
equals_sign [1166], page 36,

Equals_Sign

 <in> Ada.Characters.Latin_1 [4947], page 575,

equivalent_element

 of a hashed set [6311], page 873,
 of an ordered set [6382], page 883,

equivalent_key

of a hashed map [6186], page 843,
of an ordered map [6237], page 851,

Equivalent_Elements

<in> Ada.Containers.Hashed_Sets [6297], page 871, [6298], page 871, [6299], page 871,
<in> Ada.Containers.Ordered_Sets [6322], page 876,

Equivalent_Keys

<in> Ada.Containers.Hashed_Maps [6181], page 843, [6182], page 843, [6183], page 843,
<in> Ada.Containers.Ordered_Maps [6195], page 846,
<in> Ada.Containers.Ordered_Sets [6371], page 881,

Equivalent_Sets

<in> Ada.Containers.Hashed_Sets [6263], page 868,
<in> Ada.Containers.Ordered_Sets [6327], page 877,

erroneous execution [1041], page 22, [1080], page 29,

cause [1948], page 129, [2056], page 142, [3890], page 387, [3902], page 391, [4208],
page 447, [4499], page 488, [4509], page 491, [4510], page 491, [4669], page 523, [4674],
page 523, [4675], page 523, [4711], page 528, [4730], page 534, [4803], page 548, [5822],
page 712, [5895], page 750, [5914], page 754, [5984], page 778, [6077], page 809, [6135],
page 829, [6149], page 839, [6257], page 866, [6460], page 898, [6550], page 921, [6551],
page 922, [6552], page 922, [6553], page 922, [6564], page 926, [6565], page 926, [6566],
page 926, [6567], page 926, [6568], page 926, [6569], page 927, [6694], page 955, [6695],
page 955, [6763], page 967, [6773], page 969, [6774], page 969, [6775], page 969, [6859],
page 990, [6893], page 997, [6981], page 1017, [7001], page 1023, [7016], page 1027, [7040],
page 1030, [7411], page 1162, [7412], page 1162,

error

compile-time [1024], page 22, [1073], page 28,
link-time [1032], page 22, [1075], page 28,
run-time [1038], page 22, [1077], page 28, [4179], page 431, [4215], page 448,
<See also> bounded error, erroneous execution

ESA

<in> Ada.Characters.Latin_1 [5000], page 576,

ESC

<in> Ada.Characters.Latin_1 [4922], page 574,

Establish_RPC_Receiver

<in> System.RPC [7164], page 1051,

ETB

<in> Ada.Characters.Latin_1 [4918], page 574,

ETX

<in> Ada.Characters.Latin_1 [4898], page 573,
evaluation [1311], page 50, [7694], page 1284, [7700], page 1285,
aggregate [2376], page 190,
allocator [2853], page 232,
array_aggregate [2442], page 198,
attribute_reference [2353], page 188,
concatenation [2667], page 212,
dereference [2297], page 180,
discrete_range [1871], page 118,
extension_aggregate [2411], page 195,
generic_association [4297], page 457,
generic_association for a formal object of mode in [4321], page 460,
indexed_component [2308], page 181,
initialized_allocator [2854], page 232,
membership test [2649], page 210,
name [2294], page 180,
name that has a prefix [2295], page 180,
null literal [2362], page 189,
numeric literal [2361], page 189,
parameter_association [3167], page 271,
prefix [2296], page 180,
primary that is a name [2567], page 202,
qualified_expression [2832], page 230,
range [1554], page 77,
range_attribute_reference [2354], page 188,
record_aggregate [2399], page 193,
record_component_association_list [2400], page 193,
selected_component [2334], page 185,
short-circuit control form [2601], page 205,
slice [2318], page 183,
string_literal [2365], page 189,
uninitialized_allocator [2857], page 232,
Val [1710], page 100, [7496], page 1235,
Value [1618], page 90,
value conversion [2769], page 223,

view conversion [2811], page 227,
Wide_Value [1612], page 89,
Wide_Wide_Value [1600], page 86,
Exception [4091], page 419, [4094], page 419, [7695], page 1284,
exception occurrence [4086], page 419,
exception_choice [4118], page 420,
 <used> [4114], page 420, [8375], page 1307,
exception_declaration [4095], page 419,
 <used> [1292], page 49, [7793], page 1290,
exception_handler [4111], page 420,
 <used> [4109], page 420, [8371], page 1307,
Exception_Id
 <in> Ada.Exceptions [4141], page 423,
Exception_Identity
 <in> Ada.Exceptions [4152], page 424,
Exception_Information
 <in> Ada.Exceptions [4156], page 424,
Exception_Message
 <in> Ada.Exceptions [4150], page 424,
Exception_Name
 <in> Ada.Exceptions [4143], page 423, [4153], page 424,
Exception_Occurrence
 <in> Ada.Exceptions [4146], page 423,
Exception_Occurrence_Access
 <in> Ada.Exceptions [4147], page 424,
exception_renaming_declaration [3436], page 318,
 <used> [3418], page 316, [8207], page 1302,
Exceptions
 <child of> Ada [4140], page 423,
Exchange_Handler
 <in> Ada.Interrupts [6704], page 957,
Exclamation
 <in> Ada.Characters.Latin_1 [4928], page 574,
exclamation point [1171], page 36,
Exclude
 <in> Ada.Containers.Hashed_Maps [6171], page 842,

<in> Ada.Containers.Hashtable_Sets [6278], page 869, [6305], page 872,
<in> Ada.Containers.Ordered_Maps [6214], page 848,
<in> Ada.Containers.Ordered_Sets [6340], page 878, [6375], page 882,
excludes null
 subtype [2188], page 158,
execution [1309], page 50, [7698], page 1285,
 abort_statement [3876], page 386,
 aborting the execution of a construct [3880], page 386,
 accept_statement [3679], page 350,
 Ada program [3489], page 328,
 assignment_statement [2946], page 243, [3315], page 298, [3335], page 301,
 asynchronous_select with a delay_statement trigger [3863], page 384,
 asynchronous_select with a procedure call trigger [3862], page 384,
 asynchronous_select with an entry call trigger [3861], page 384,
 block_statement [3001], page 251,
 call on a dispatching operation [2093], page 147,
 call on an inherited subprogram [1507], page 70,
 case_statement [2974], page 247,
 conditional_entry_call [3849], page 383,
 delay_statement [3755], page 361,
 dynamically enclosing [4133], page 422,
 entry_body [3684], page 351,
 entry_call_statement [3700], page 353,
 exit_statement [3006], page 252,
 extended_return_statement [3196], page 273,
 goto_statement [3010], page 253,
 handled_sequence_of_statements [4122], page 421,
 handler [4138], page 423,
 if_statement [2964], page 245,
 instance of Unchecked_Deallocation [3329], page 301,
 loop_statement [2991], page 249,
 loop_statement with a for_iteration_scheme [2993], page 249,
 loop_statement with a while_iteration_scheme [2992], page 249,
 null_statement [2934], page 241,
 partition [4045], page 412,
 pragma [1269], page 45,

- program [4044], page 412,
- protected subprogram call [3625], page 345,
- raise_statement with an exception_name [4128], page 422,
- re-raise statement [4129], page 422,
- remote subprogram call [7133], page 1045,
- requeue protected entry [3724], page 357,
- requeue task entry [3723], page 357,
- requeue_statement [3722], page 357,
- selective_accept [3832], page 379,
- sequence_of_statements [2936], page 241,
- simple_return_statement [3199], page 274,
- subprogram call [3159], page 268,
- subprogram_body [3101], page 262,
- task [3542], page 333,
- task_body [3543], page 333,
- timed_entry_call [3845], page 382,

execution resource

- associated with a protected object [3610], page 341,
- required for a task to run [3506], page 329,

execution time

- of a task [6998], page 1022,

Execution_Time

- <child of> Ada [6989], page 1021,

exhaust

- a budget [7036], page 1029,

exist

- cease to [3332], page 301, [4722], page 533,

Exists

- <in> Ada.Directories [5945], page 759,
- <in> Ada.Environment_Variables [5978], page 775,

exit_statement [3002], page 252,

- <used> [2911], page 240, [8077], page 1298,

Exit_Status

- <in> Ada.Command_Line [5922], page 755,

Exp

- <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7238], page 1092,

<in> Ada.Numerics.Generic_Elementary_Functions [5414], page 649,
expanded name [2331], page 184,
Expanded_Name
<in> Ada.Tags [2029], page 138,
expected profile [3482], page 326,
accept_statement entry_direct_name [3663], page 348,
Access attribute_reference prefix [2207], page 164,
attribute_definition_clause name [4470], page 487,
character_literal [2359], page 189,
formal subprogram actual [4392], page 471,
formal subprogram default_name [4391], page 471,
subprogram_renaming_declaration [3448], page 320,
expected type [3477], page 326,
abort_statement task_name [3875], page 386,
access attribute_reference [2206], page 164,
Access attribute_reference prefix [2208], page 164,
actual parameter [3165], page 270,
aggregate [2375], page 190,
allocator [2847], page 231,
array_aggregate [2437], page 197,
array_aggregate component_expression [2438], page 197,
array_aggregate discrete_choice [2439], page 197,
assignment_statement expression [2945], page 242,
assignment_statement variable_name [2944], page 242,
attribute_definition_clause expression or name [4469], page 487,
attribute_designator expression [2351], page 187,
case expression [2972], page 247,
case_statement_alternative discrete_choice [2973], page 247,
character_literal [2358], page 189,
code_statement [4650], page 519,
component_clause expressions [4575], page 502,
component_declaration default_expression [1968], page 130,
condition [2963], page 245,
decimal fixed point type digits [1775], page 107,
delay_relative_statement expression [3736], page 359,
delay_until_statement expression [3737], page 359,

delta_constraint expression [7426], page 1168,
dereference name [2291], page 179,
discrete_subtype_definition range [1836], page 114,
discriminant default_expression [1910], page 123,
discriminant_association expression [1939], page 128,
entry_index [3664], page 348,
enumeration_representation_clause expressions [4548], page 500,
expression of extended_return_statement [3192], page 272,
expression of simple_return_statement [3191], page 272,
extension_aggregate [2408], page 195,
extension_aggregate_ancestor expression [2409], page 195,
first_bit [4577], page 502,
fixed point type delta [1774], page 107,
generic formal in object actual [4311], page 458,
generic formal object default_expression [4310], page 458,
index_constraint discrete_range [1865], page 118,
indexed_component expression [2306], page 181,
Interrupt_Priority pragma argument [6800], page 975,
last_bit [4578], page 502,
link name [6429], page 895,
membership test simple_expression [2642], page 206,
modular_type_definition expression [1666], page 95,
number_declaration expression [1478], page 65,
object_declaration initialization expression [1451], page 61,
parameter default_expression [3065], page 257,
position [4576], page 502,
Priority pragma argument [6799], page 975,
range simple_expressions [1546], page 77,
range_attribute_designator expression [2352], page 187,
range_constraint range [1545], page 76,
real_range_specification bounds [1735], page 103,
record_aggregate [2395], page 191,
record_component_association expression [2397], page 192,
requested decimal precision [1734], page 103,
restriction parameter expression [4750], page 535,
short-circuit control form relation [2591], page 204,

signed_integer_type_definition simple_expression [1665], page 95,
slice discrete_range [2317], page 182,
Storage_Size pragma argument [4529], page 497,
string_literal [2360], page 189,
type_conversion operand [2743], page 219,
variant_part discrete_choice [1999], page 134,
expiration time
 [<partial>] [3727], page 358,
 for a delay_relative_statement [3757], page 361,
 for a delay_until_statement [3756], page 361,
expires
 execution timer [7015], page 1026,
explicit declaration [1298], page 49, [7685], page 1284,
explicit initial value [1431], page 61,
explicit_actual_parameter [3154], page 267,
 <used> [3153], page 267, [8171], page 1300,
explicit_dereference [2286], page 179,
 <used> [2272], page 179, [7963], page 1295,
explicit_generic_actual_parameter [4275], page 455,
 <used> [4274], page 455, [8407], page 1308,
explicitly assign [4035], page 410,
explicitly limited record [1971], page 131,
exponent [1214], page 40, [2718], page 218,
 <used> [1209], page 40, [1232], page 41, [7758], page 1289,
Exponent attribute [5503], page 667,
exponentiation operator [2519], page 201, [2714], page 218,
Export pragma [6417], page 894, [7548], page 1242,
exported
 aspect of representation [6449], page 897,
exported entity [6437], page 896,
expression [2453], page 201, [2526], page 201,
 <used> [1262], page 44, [1445], page 61, [1477], page 65, [1664], page 95, [1728], page 103,
 [1765], page 106, [1768], page 107, [1772], page 107, [1909], page 123, [1934], page 127,
 [1996], page 134, [2305], page 181, [2345], page 187, [2350], page 187, [2388], page 191,
 [2406], page 194, [2421], page 196, [2432], page 196, [2566], page 201, [2731], page 219,
 [2828], page 229, [2939], page 242, [2962], page 245, [2966], page 246, [3155], page 267,

[3185], page 272, [3181], page 272, [3647], page 348, [3733], page 359, [3735], page 359, [4125], page 421, [4165], page 427, [4276], page 455, [4465], page 487, [4528], page 496, [4570], page 502, [4749], page 535, [6414], page 894, [6421], page 894, [6428], page 895, [6430], page 895, [6683], page 954, [6795], page 975, [6798], page 975, [6829], page 981, [6851], page 988, [7424], page 1168, [7435], page 1171, [7445], page 1174, [7507], page 1242, [7523], page 1242, [7552], page 1242, [7558], page 1243, [7572], page 1243, [7575], page 1243, [7607], page 1243, [7612], page 1243, [7625], page 1244, [7643], page 1244, [7867], page 1292, extended_digit [1239], page 41,

<used> [1238], page 41, [7769], page 1289,

Extended_Index <subtype of> Index_Type'Base

<in> Ada.Containers.Vectors [5997], page 780,

extended_return_statement [3182], page 272,

<used> [2926], page 241, [8091], page 1298,

extension

of a private type [2018], page 137, [2064], page 143,

of a record type [2016], page 137, [2062], page 143,

of a type [2015], page 137, [2060], page 143,

<in> Ada.Directories [5940], page 759,

extension_aggregate [2402], page 194,

<used> [2373], page 190, [7995], page 1296,

external_call [3618], page 344,

external_effect

of the execution of an Ada program [1051], page 24,

volatile/atomic objects [6745], page 964,

external_file [5578], page 680,

external_interaction [1053], page 24,

external_name [6456], page 897,

external_requeue [3621], page 344,

external_streaming

type supports [4802], page 548,

External_Tag

<in> Ada.Tags [2032], page 138,

External_Tag_attribute [4540], page 498,

External_Tag_clause [4480], page 487, [4541], page 498, [7470], page 1192,

extra_permission_to_avoid_raising_exceptions [4219], page 448,

extra_permission_to_reorder_actions [4221], page 449,

30.7 F

factor [2554], page 201,

<used> [2551], page 201, [8045], page 1297,

factory [2057], page 143,

failure

of a language-defined check [4180], page 431,

<in> Ada.Command_Line [5924], page 755,

fall-back handler [6785], page 972,

False [1650], page 95,

family

entry [3672], page 350,

Feminine_Ordinal_Indicator

<in> Ada.Characters.Latin_1 [5036], page 577,

FF

<in> Ada.Characters.Latin_1 [4907], page 573,

Field <subtype of> Integer

<in> Ada.Text_IO [5674], page 698,

file

as file object [5581], page 681,

file name [5969], page 761,

file terminator [5660], page 697,

File_Access

<in> Ada.Text_IO [5696], page 700,

File_Kind

<in> Ada.Directories [5943], page 759,

File_Mode

<in> Ada.Direct_IO [5621], page 691,

<in> Ada.Sequential_IO [5598], page 683,

<in> Ada.Streams.Stream_IO [5864], page 746,

<in> Ada.Text_IO [5670], page 698,

File_Size

<in> Ada.Directories [5944], page 759,

File_Type

- <in> Ada.Direct_IO [5620], page 691,
- <in> Ada.Sequential_IO [5597], page 683,
- <in> Ada.Streams.Stream_IO [5863], page 746,
- <in> Ada.Text_IO [5669], page 698,

Filter_Type

- <in> Ada.Directories [5950], page 759,

finalization

- of a master [3327], page 300,
- of a protected object [3611], page 341,
- of a protected object [6693], page 955,
- of a task object [7442], page 1172,
- of an object [3328], page 300,
- of environment task for a foreign language main subprogram [6464], page 898,
- <child of> Ada [3299], page 296,

finalization of the collection [3330], page 301,

Finalize [3297], page 295,

- <in> Ada.Finalization [3303], page 296, [3306], page 296,

Find

- <in> Ada.Containers.Doubly_Linked_Lists [6117], page 814,
- <in> Ada.Containers.Hashing_Maps [6177], page 842,
- <in> Ada.Containers.Hashing_Sets [6294], page 871, [6307], page 872,
- <in> Ada.Containers.Ordered_Maps [6229], page 849,
- <in> Ada.Containers.Ordered_Sets [6363], page 880, [6377], page 882,
- <in> Ada.Containers.Vectors [6054], page 786,

Find_Index

- <in> Ada.Containers.Vectors [6053], page 786,

Find-Token

- <in> Ada.Strings.Bounded [5250], page 617,
- <in> Ada.Strings.Fixed [5193], page 594,
- <in> Ada.Strings.Unbounded [5303], page 630,

Fine_Delta

- <in> System [4614], page 511,

First

- <in> Ada.Containers.Doubly_Linked_Lists [6109], page 814,

- <in> Ada.Containers.Hashtable [6174], page 842,
- <in> Ada.Containers.Hashtable_Sets [6291], page 870,
- <in> Ada.Containers.Ordered_Maps [6219], page 849,
- <in> Ada.Containers.Ordered_Sets [6355], page 879,
- <in> Ada.Containers.Vectors [6044], page 785,

First attribute [1557], page 77, [1873], page 119,

first element

- of a hashed set [6315], page 873,
- of an ordered set [6385], page 883,
- of a set [6247], page 855,

first node

- of a hashed map [6190], page 844,
- of a map [6141], page 830,
- of an ordered map [6240], page 851,

first subtype [1371], page 54, [1517], page 73,

First(N) attribute [1875], page 119,

first_bit [4571], page 502,

- <used> [4567], page 502, [8480], page 1310,

First_Bit attribute [4584], page 507,

First_Element

- <in> Ada.Containers.Doubly_Linked_Lists [6110], page 814,
- <in> Ada.Containers.Ordered_Maps [6220], page 849,
- <in> Ada.Containers.Ordered_Sets [6356], page 879,
- <in> Ada.Containers.Vectors [6045], page 785,

First_Index

- <in> Ada.Containers.Vectors [6043], page 785,

First_Key

- <in> Ada.Containers.Ordered_Maps [6221], page 849,

Fixed

- <child of> Ada.Strings [5180], page 592,

fixed point type [1757], page 106,

Fixed_IO

- <in> Ada.Text_IO [5785], page 706,

fixed_point_definition [1761], page 106,

- <used> [1719], page 102, [7860], page 1292,

Float [1747], page 104, [1749], page 104,

- <in> Standard [4841], page 558,
- Float_IO
 - <in> Ada.Text_IO [5775], page 705,
- Float_Random
 - <child of> Ada.Numerics [5447], page 655,
- Float_Text_IO
 - <child of> Ada [5829], page 735,
- Float_Wide_Text_IO
 - <child of> Ada [5850], page 745,
- Float_Wide_Wide_Text_IO
 - <child of> Ada [5853], page 745,
- Floating
 - <in> Interfaces.COBOL [6582], page 932,
- floating point type [1726], page 103,
- floating_point_definition [1727], page 103,
 - <used> [1718], page 102, [7859], page 1292,
- Floor
 - <in> Ada.Containers.Ordered_Maps [6231], page 850,
 - <in> Ada.Containers.Ordered_Sets [6364], page 880, [6378], page 882,
- Floor attribute [5517], page 669,
- Flush
 - <in> Ada.Streams.Stream_IO [5887], page 748,
 - <in> Ada.Text_IO [5703], page 700,
- Fore attribute [1803], page 110,
- form
 - of an external file [5580], page 680,
 - <in> Ada.Direct_IO [5632], page 692,
 - <in> Ada.Sequential_IO [5607], page 684,
 - <in> Ada.Streams.Stream_IO [5875], page 747,
 - <in> Ada.Text_IO [5685], page 699,
- formal object, generic [4299], page 458,
- formal package, generic [4398], page 474,
- formal parameter
 - of a subprogram [3064], page 257,
- formal subprogram, generic [4377], page 470,
- formal subtype [4347], page 461,

formal_type [4345], page 461,
formal_abstract_subprogram_declaration [4384], page 471,
 <used> [4380], page 470, [8443], page 1309,
formal_access_type_definition [4370], page 468,
 <used> [4338], page 461, [8435], page 1308,
formal_array_type_definition [4366], page 467,
 <used> [4337], page 461, [8434], page 1308,
formal_concrete_subprogram_declaration [4381], page 471,
 <used> [4379], page 470, [8442], page 1309,
formal_decimal_fixed_point_definition [4365], page 467,
 <used> [4336], page 461, [8433], page 1308,
formal_derived_type_definition [4351], page 463,
 <used> [4330], page 460, [8427], page 1308,
formal_discrete_type_definition [4360], page 466,
 <used> [4331], page 460, [8428], page 1308,
formal_floating_point_definition [4363], page 466,
 <used> [4334], page 461, [8431], page 1308,
formal_interface_type_definition [4374], page 470,
 <used> [4339], page 461, [8436], page 1308,
formal_modular_type_definition [4362], page 466,
 <used> [4333], page 461, [8430], page 1308,
formal_object_declaration [4300], page 458,
 <used> [4244], page 451, [8389], page 1307,
formal_ordinary_fixed_point_definition [4364], page 466,
 <used> [4335], page 461, [8432], page 1308,
formal_package_actual_part [4403], page 475,
 <used> [4402], page 475, [8452], page 1309,
formal_package_association [4407], page 475,
 <used> [4406], page 475, [8455], page 1310,
formal_package_declaration [4399], page 475,
 <used> [4247], page 451, [8392], page 1308,
formal_part [3051], page 256,
 <used> [3044], page 256, [3046], page 256, [8142], page 1300,
formal_private_type_definition [4350], page 462,
 <used> [4329], page 460, [8426], page 1308,
formal_signed_integer_type_definition [4361], page 466,

<used> [4332], page 461, [8429], page 1308,
formal_subprogram_declaration [4378], page 470,

<used> [4246], page 451, [8391], page 1308,
formal_type_declaration [4324], page 460,

<used> [4245], page 451, [8390], page 1308,
formal_type_definition [4328], page 460,

<used> [4327], page 460, [8425], page 1308,
format_effector [1139], page 34,

Formatting

<child of> Ada.Calendar [3773], page 365,

Fortran

<child of> Interfaces [6635], page 945,

Fortran interface [6634], page 945,

Fortran standard [1093], page 30,

Fortran.Character

<in> Interfaces.Fortran [6646], page 946,

Fortran.Integer

<in> Interfaces.Fortran [6636], page 945,

Fraction attribute [5505], page 667,

Fraction.One.Half

<in> Ada.Characters.Latin_1 [5057], page 578,

Fraction.One.Quarter

<in> Ada.Characters.Latin_1 [5056], page 578,

Fraction.Three.Quarters

<in> Ada.Characters.Latin_1 [5058], page 578,

Free

<in> Ada.Strings.Unbounded [5279], page 625,

<in> Interfaces.C.Strings [6540], page 916,

freed

<See> nonexistent [4720], page 533,

freeing storage [4718], page 532,

freezing

by a constituent of a construct [4809], page 550,

by an expression [4813], page 550,

by an implicit call [4815], page 551,

by an object name [4814], page 550,

class-wide type caused by the freezing of the specific type [4828], page 552,
constituents of a full type definition [4826], page 551,
designated subtype caused by an allocator [4822], page 551,
entity [4804], page 550,
entity caused by a body [4807], page 550,
entity caused by a construct [4808], page 550,
entity caused by a name [4818], page 551,
entity caused by the end of an enclosing construct [4806], page 550,
first subtype caused by the freezing of the type [4827], page 551,
function call [4824], page 551,
generic_instantiation [4810], page 550,
nominal subtype caused by a name [4819], page 551,
object_declaration [4811], page 550,
specific type caused by the freezing of the class-wide type [4829], page 552,
subtype caused by a record extension [4812], page 550,
subtype caused by an implicit conversion [4816], page 551,
subtype caused by an implicit dereference [4820], page 551,
subtypes of the profile of a callable entity [4823], page 551,
type caused by a range [4821], page 551,
type caused by an expression [4817], page 551,
type caused by the freezing of a subtype [4825], page 551,

freezing points

entity [4805], page 550,

Friday

<in> Ada.Calendar.Formatting [3779], page 365,

FS

<in> Ada.Characters.Latin_1 [4923], page 574,

full conformance

for discrete_subtype_definitions [3131], page 265,

for expressions [3128], page 265,

for known_discriminant_parts [3129], page 265,

for profiles [3126], page 265,

required [2203], page 161, [3099], page 262, [3267], page 285, [3380], page 309, [3451],
page 320, [3666], page 349, [3670], page 349, [3671], page 350, [4000], page 404, [4001],
page 404,

full constant declaration [1454], page 61,

corresponding to a formal object of mode in [4318], page 460,
full declaration [3283], page 290,
full name
 of a file [5970], page 761,
full stop [1158], page 36,
full type [1375], page 54,
full type definition [1376], page 54,
full view
 of a type [1377], page 54,
Full_Name
 <in> Ada.Directories [5937], page 758, [5957], page 760,
Full_Stop
 <in> Ada.Characters.Latin_1 [4942], page 574,
full_type_declaration [1356], page 53,
 <used> [1352], page 53, [7797], page 1290,
function [3013], page 255, [7701], page 1285,
function instance [4290], page 456,
function_call [3144], page 267,
 <used> [2278], page 179, [7969], page 1295,
function_specification [3026], page 255,
 <used> [3022], page 255, [8127], page 1299,

30.8 G

garbage collection [4738], page 534,
general access type [2170], page 157, [2174], page 157,
general_access_modifier [2155], page 156,
 <used> [2153], page 156, [7938], page 1294,
generation
 of an interrupt [6668], page 951,
Generator
 <in> Ada.Numerics.Discrete_Random [5460], page 656,
 <in> Ada.Numerics.Float_Random [5448], page 655,

generic actual [4285], page 455,
generic actual parameter [4284], page 455,
generic actual subtype [4340], page 461,
generic actual type [4342], page 461,
generic body [4254], page 453,
generic contract issue [4060], page 414,
[<partial>] [1913], page 124, [1940], page 128, [2067], page 143, [2141], page 153, [2227],
page 172, [2235], page 173, [2760], page 222, [2767], page 223, [2851], page 231, [2893],
page 237, [3213], page 275, [3397], page 312, [3400], page 313, [3435], page 318, [3450],
page 320, [3536], page 331, [3604], page 340, [3603], page 340, [3665], page 349, [4063],
page 415, [4315], page 459, [4395], page 471,
generic formal [4252], page 451,
generic formal object [4298], page 458,
generic formal package [4397], page 474,
generic formal subprogram [4376], page 470,
generic formal subtype [4346], page 461,
generic formal type [4344], page 461,
generic function [4251], page 451,
generic package [4248], page 451,
generic procedure [4250], page 451,
generic subprogram [4249], page 451,
generic unit [4226], page 450, [7702], page 1285,
 <See also> dispatching operation [2008], page 136,
generic_actual_part [4269], page 455,
 <used> [4260], page 455, [4404], page 475, [8399], page 1308,
Generic_Array_Sort
 <child of> Ada.Containers [6395], page 891,
generic_association [4272], page 455,
 <used> [4271], page 455, [4408], page 475, [8405], page 1308,
Generic_Bounded_Length
 <in> Ada.Strings.Bounded [5217], page 611,
Generic_Complex_Arrays
 <child of> Ada.Numerics [7335], page 1130,
Generic_Complex_Elementary_Functions
 <child of> Ada.Numerics [7234], page 1091,
Generic_Complex_Types

<child of> Ada.Numerics [7207], page 1084,
Generic_Constrained_Array_Sort
<child of> Ada.Containers [6397], page 892,
generic_declaration [4231], page 450,
<used> [1293], page 49, [3929], page 395, [7794], page 1290,
Generic_Dispatching_Constructor
<child of> Ada.Tags [2049], page 141,
Generic_Elementary_Functions
<child of> Ada.Numerics [5410], page 649,
generic_formal_parameter_declaration [4243], page 451,
<used> [4241], page 450, [8387], page 1307,
generic_formal_part [4240], page 450,
<used> [4235], page 450, [4238], page 450, [8383], page 1307,
generic_instantiation [4257], page 455,
<used> [1294], page 49, [3930], page 395, [7795], page 1290,
Generic_Keys
<in> Ada.Containers.Hashed_Sets [6301], page 871,
<in> Ada.Containers.Ordered_Sets [6370], page 881,
generic_package_declaration [4237], page 450,
<used> [4233], page 450, [8382], page 1307,
Generic_Real_Arrays
<child of> Ada.Numerics [7319], page 1119,
generic_renaming_declaration [3456], page 323,
<used> [3421], page 316, [3933], page 395, [8210], page 1302,
Generic_Sorting
<in> Ada.Containers.Doubly_Linked_Lists [6123], page 815,
<in> Ada.Containers.Vectors [6061], page 787,
generic_subprogram_declaration [4234], page 450,
<used> [4232], page 450, [8381], page 1307,
Get
<in> Ada.Text_IO [5737], page 702, [5748], page 703, [5760], page 703, [5764], page 704,
[5770], page 704, [5773], page 705, [5780], page 705, [5783], page 705, [5789], page 706,
[5793], page 706, [5800], page 707, [5803], page 707, [5808], page 708, [5812], page 708,
<in> Ada.Text_IO.Complex_IO [7268], page 1098, [7271], page 1098,
Get_Deadline
<in> Ada.Dispatching.EDF [6857], page 988,

Get_Immediate

<in> Ada.Text_IO [5744], page 702, [5745], page 702,

Get_Line

<in> Ada.Text_IO [5751], page 703, [5753], page 703,

<in> Ada.Text_IO.Bounded_IO [5835], page 740, [5836], page 740, [5837], page 740, [5838], page 740,

<in> Ada.Text_IO.Unbounded_IO [5844], page 742, [5845], page 743, [5846], page 743, [5847], page 743,

Get_Next_Entry

<in> Ada.Directories [5955], page 760,

Get_Priority

<in> Ada.Dynamic_Priorities [6890], page 997,

global to [3350], page 305,

Glossary [7670], page 1283,

goto_statement [3007], page 253,

<used> [2912], page 240, [8078], page 1298,

govern a variant [2003], page 135,

govern a variant_part [2002], page 135,

grammar

complete listing [7736], page 1289,

cross reference [8497], page 1311,

notation [1061], page 25,

resolution of ambiguity [3465], page 324,

under Syntax heading [1015], page 22,

graphic character

a category of Character [4884], page 568,

graphic_character [1144], page 35,

<used> [1243], page 42, [1248], page 42, [7775], page 1290,

Graphic_Set

<in> Ada.Strings.Maps.Constants [5326], page 636,

Grave

<in> Ada.Characters.Latin_1 [4956], page 575,

greater than operator [2480], page 201, [2632], page 206,

greater than or equal operator [2484], page 201, [2636], page 206,

greater-than sign [1167], page 36,

Greater_Than_Sign

<in> Ada.Characters.Latin_1 [4948], page 575,
Group_Budget
<in> Ada.Execution_Time.Group_Budgets [7018], page 1027,
Group_Budget_Error
<in> Ada.Execution_Time.Group_Budgets [7034], page 1028,
Group_Budget_Handler
<in> Ada.Execution_Time.Group_Budgets [7019], page 1027,
Group_Budgets
<child of> Ada.Execution_Time [7017], page 1027,
GS
<in> Ada.Characters.Latin_1 [4924], page 574,
guard [3817], page 378,
<used> [3814], page 378, [8303], page 1304,

30.9 H

handle

an exception [4093], page 419, [7697], page 1284,
an exception occurrence [4130], page 422, [4137], page 423,
handled_sequence_of_statements [4107], page 420,
<used> [2999], page 251, [3097], page 261, [3186], page 272, [3246], page 281, [3527],
page 330, [3644], page 348, [3653], page 348, [8245], page 1303,

handler

execution timer [7014], page 1026,
group budget [7037], page 1029,
interrupt [6675], page 951,
termination [6784], page 972,
timing event [7051], page 1032,

Handling

<child of> Ada.Characters [4860], page 566,

Has_Element

<in> Ada.Containers.Doubly_Linked_Lists [6120], page 815,
<in> Ada.Containers.Hashed_Maps [6180], page 843,

- <in> Ada.Containers.Hashtable_Sets [6296], page 871,
- <in> Ada.Containers.Ordered_Maps [6234], page 850,
- <in> Ada.Containers.Ordered_Sets [6367], page 880,
- <in> Ada.Containers.Vectors [6058], page 787,

Hash

- <child of> Ada.Strings [5403], page 646,
- <child of> Ada.Strings.Bounded [5404], page 647,
- <child of> Ada.Strings.Unbounded [5405], page 647,

Hash_Type

- <in> Ada.Containers [5990], page 779,

Hashed_Maps

- <child of> Ada.Containers [6150], page 840,

Hashed_Sets

- <child of> Ada.Containers [6258], page 867,

Head

- <in> Ada.Strings.Bounded [5267], page 619, [5268], page 620,
- <in> Ada.Strings.Fixed [5210], page 597, [5211], page 597,
- <in> Ada.Strings.Unbounded [5320], page 633, [5321], page 633,

head (of a queue) [6816], page 979,

heap management

- user-defined [4684], page 526,
- <See also> allocator [2843], page 230,

held priority [6977], page 1016,

heterogeneous input-output [5858], page 746,

hexadecimal

- literal [1227], page 41,

hexadecimal digit

- a category of Character [4890], page 569,

hexadecimal literal [1225], page 41,

Hexadecimal_Digit_Set

- <in> Ada.Strings.Maps.Constants [5332], page 636,

hidden from all visibility [3372], page 308, [3382], page 310,

- by lack of a with_clause [3386], page 311,
- for a declaration completed by a subsequent declaration [3385], page 310,
- for overridden declaration [3383], page 310,
- within the declaration itself [3384], page 310,

hidden from direct visibility [3373], page 308, [3391], page 311,
by an inner homograph [3392], page 311,
where hidden from all visibility [3393], page 311,
hiding [3371], page 308,
High_Order_First [4590], page 508,
<in> Interfaces.COBOL [6606], page 933,
<in> System [4623], page 512,
highest precedence operator [2703], page 217,
highest_precedence_operator [2578], page 203,
Hold
<in> Ada.Asynchronous_Task_Control [6973], page 1016,
homograph [3376], page 308,
Hour
<in> Ada.Calendar.Formatting [3790], page 366,
Hour_Number <subtype of> Natural
<in> Ada.Calendar.Formatting [3783], page 366,
HT
<in> Ada.Characters.Latin_1 [4904], page 573,
HTJ
<in> Ada.Characters.Latin_1 [5002], page 576,
HTS
<in> Ada.Characters.Latin_1 [5001], page 576,
Hyphen
<in> Ada.Characters.Latin_1 [4940], page 574,
hyphen–minus [1156], page 36,

30.10 I

i

<in> Ada.Numerics.Generic_Complex_Types [7210], page 1084,
<in> Interfaces.Fortran [6643], page 946,
identifier [1181], page 38,
<used> [1255], page 44, [1259], page 44, [1273], page 46, [1278], page 46, [1296], page 49,

[2281], page 179, [2328], page 184, [2344], page 187, [2982], page 248, [3000], page 251, [3031], page 255, [3236], page 279, [3248], page 281, [3520], page 329, [3528], page 330, [3577], page 338, [3588], page 338, [3645], page 348, [3654], page 348, [4169], page 427, [4186], page 431, [4183], page 431, [4745], page 535, [6412], page 894, [6418], page 894, [6424], page 895, [6825], page 981, [6828], page 981, [6862], page 991, [6864], page 991, [6881], page 994, [6882], page 995, [6984], page 1020, [7419], page 1164, [7451], page 1175, [7510], page 1242, [7529], page 1242, [7549], page 1242, [7555], page 1243, [7578], page 1243, [7581], page 1243, [7590], page 1243, [7598], page 1243, [7610], page 1243, [7615], page 1244, [7622], page 1244, [7646], page 1244, [7649], page 1244, [7655], page 1244, [7667], page 1262, [7782], page 1290,

identifier specific to a pragma [1267], page 45,

identifier_extend [1192], page 38,

<used> [1184], page 38, [7742], page 1289,

identifier_start [1185], page 38,

<used> [1183], page 38, [7740], page 1289,

Identity

<in> Ada.Strings.Maps [5172], page 587,

<in> Ada.Strings.Wide_Maps [5363], page 639,

<in> Ada.Strings.Wide_Wide_Maps [5394], page 644,

Identity attribute [4160], page 425, [6755], page 966,

idle task [6978], page 1016,

if_statement [2955], page 245,

<used> [2922], page 240, [8087], page 1298,

illegal

construct [1026], page 22,

partition [1034], page 22,

Im

<in> Ada.Numerics.Generic_Complex_Arrays [7339], page 1130, [7352], page 1133,

<in> Ada.Numerics.Generic_Complex_Types [7214], page 1084,

image

of a value [1585], page 81, [1589], page 83, [7498], page 1237, [7499], page 1238,

<in> Ada.Calendar.Formatting [3801], page 368, [3803], page 369,

<in> Ada.Numerics.Discrete_Random [5468], page 656,

<in> Ada.Numerics.Float_Random [5457], page 655,

<in> Ada.Task_Identification [6749], page 965,

<in> Ada.Text_IO.Editing [7198], page 1074,

Image attribute [1591], page 84,
Imaginary
 <in> Ada.Numerics.Generic_Complex_Types [7209], page 1084,
Imaginary <subtype of> Imaginary
 <in> Interfaces.Fortran [6642], page 946,
immediate scope
 of (a view of) an entity [3360], page 307,
 of a declaration [3351], page 306,
immediately enclosing [3347], page 305,
immediately visible [3369], page 308, [3388], page 311,
immediately within [3345], page 305,
implementation advice [1047], page 23,
 summary of advice [7668], page 1267,
implementation defined [1054], page 25,
 summary of characteristics [7665], page 1250,
implementation permissions [1046], page 22,
implementation requirements [1043], page 22,
implementation–dependent
 <See> unspecified [1057], page 25,
implemented
 by a protected entry [3597], page 339,
 by a protected subprogram [3596], page 339,
 by a task entry [3531], page 330,
implicit declaration [1299], page 49, [7686], page 1284,
implicit initial values
 for a subtype [1464], page 62,
implicit subtype conversion [2822], page 228, [2823], page 228,
 Access attribute [2231], page 172,
 access discriminant [1927], page 126,
 array bounds [2786], page 225,
 array index [2309], page 181,
 assignment to view conversion [2818], page 227,
 assignment_statement [2951], page 243,
 bounds of a decimal fixed point type [1789], page 108,
 bounds of a fixed point type [1785], page 108,
 bounds of a range [1555], page 77, [1853], page 116,

choices of aggregate [2443], page 198,
component defaults [1465], page 62,
delay expression [3758], page 361,
derived type discriminants [1504], page 69,
discriminant values [1945], page 129,
entry index [3680], page 350,
expressions in aggregate [2401], page 193,
expressions of aggregate [2444], page 199,
function return [3197], page 274, [3201], page 274,
generic formal object of mode in [4323], page 460,
inherited enumeration literal [1510], page 70,
initialization expression [1467], page 63,
initialization expression of allocator [2856], page 232,
named number value [1479], page 66,
operand of concatenation [2671], page 212,
parameter passing [3168], page 271, [3170], page 271, [3178], page 272,
pragma Interrupt_Priority [6808], page 976, [6870], page 992,
pragma Priority [6807], page 976, [6869], page 992,
qualified_expression [2839], page 230,
reading a view conversion [2819], page 228,
result of inherited function [1508], page 70,
implicit_dereference [2288], page 179,
 <used> [2285], page 179, [7974], page 1295,
Import pragma [6411], page 894, [7554], page 1242,
imported
 aspect of representation [6447], page 897,
imported entity [6436], page 896,
in (membership test) [2486], page 201, [2640], page 206,
inaccessible partition [7063], page 1035,
inactive
 a task state [3499], page 328,
Include
 <in> Ada.Containers.Hashing_Maps [6169], page 842,
 <in> Ada.Containers.Hashing_Sets [6276], page 869,
 <in> Ada.Containers.Ordering_Maps [6212], page 848,
 <in> Ada.Containers.Ordering_Sets [6338], page 878,

included

one range in another [1544], page 76,

incomplete type [1329], page 51, [2198], page 160, [7703], page 1285,

incomplete view [2199], page 160,

tagged [2200], page 160,

incomplete_type_declaration [2195], page 160,

<used> [1353], page 53, [7798], page 1290,

Increment

<in> Interfaces.C.Pointers [6559], page 923,

indefinite subtype [1427], page 60, [1926], page 126,

Indefinite_Doubly_Linked_Lists

<child of> Ada.Containers [6390], page 888,

Indefinite_Hashed_Maps

<child of> Ada.Containers [6391], page 889,

Indefinite_Hashed_Sets

<child of> Ada.Containers [6393], page 890,

Indefinite_Ordered_Maps

<child of> Ada.Containers [6392], page 890,

Indefinite_Ordered_Sets

<child of> Ada.Containers [6394], page 891,

Indefinite_Vectors

<child of> Ada.Containers [6389], page 887,

independent subprogram [4222], page 449,

independently addressable [3900], page 389,

index

of an element of an open direct file [5593], page 683,

<in> Ada.Direct_IO [5639], page 692,

<in> Ada.Streams.Stream_IO [5884], page 748,

<in> Ada.Strings.Bounded [5239], page 615, [5240], page 615, [5241], page 616, [5242], page 616, [5243], page 616, [5244], page 616,

<in> Ada.Strings.Fixed [5182], page 592, [5183], page 593, [5184], page 593, [5185], page 593, [5186], page 593, [5187], page 593,

<in> Ada.Strings.Unbounded [5292], page 629, [5293], page 629, [5294], page 629, [5295], page 629, [5296], page 629, [5297], page 630,

index range [1843], page 115,

index subtype [1837], page 115,

index type [1838], page 115,
Index_Check [4196], page 435,
[<partial>] [2310], page 181, [2319], page 183, [2448], page 199, [2450], page 199, [2668],
page 212, [2807], page 227, [2837], page 230, [2859], page 232,
index_constraint [1858], page 117,
 <used> [1399], page 56, [7825], page 1291,
Index_Error
 <in> Ada.Strings [5149], page 584,
Index_Non_Blank
 <in> Ada.Strings.Bounded [5245], page 616, [5246], page 616,
 <in> Ada.Strings.Fixed [5188], page 593, [5189], page 593,
 <in> Ada.Strings.Unbounded [5298], page 630, [5299], page 630,
index_subtype_definition [1824], page 114,
 <used> [1822], page 114, [7877], page 1292,
indexed_component [2302], page 181,
 <used> [2273], page 179, [7964], page 1295,
indivisible [6734], page 963,
inferable discriminants [6577], page 930,
Information
 <child of> Ada.Directories [5974], page 774,
information hiding
 <See> package [3225], page 279,
 <See> private types and private extensions [3252], page 283,
information systems [6658], page 949, [7169], page 1054,
informative [1011], page 21,
inheritance
 <See> derived types and classes [1482], page 66,
 <See also> tagged types and type extension [2012], page 136,
inherited
 from an ancestor type [1526], page 75,
inherited component [1496], page 68, [1497], page 68,
inherited discriminant [1495], page 68,
inherited entry [1499], page 68,
inherited protected subprogram [1498], page 68,
inherited subprogram [1500], page 68,
initialization

of a protected object [3608], page 340,
of a protected object [6685], page 955, [6689], page 955,
of a task object [3540], page 331, [7440], page 1172,
of an object [1469], page 63,
initialization expression [1432], page 61, [1452], page 61,
Initialize [3296], page 295,
 <in> Ada.Finalization [3301], page 296, [3305], page 296,
initialized allocator [2848], page 231,
initialized by default [1468], page 63,
Inline pragma [3136], page 266, [7560], page 1243,
innermost dynamically enclosing [4134], page 422,
input [5576], page 680,
Input attribute [4782], page 544, [4786], page 546,
Input clause [4488], page 487, [4798], page 547,
input–output
 unspecified for access types [5583], page 681,
Insert
 <in> Ada.Containers.Doubly_Linked_Lists [6095], page 812, [6096], page 812, [6097],
page 812,
 <in> Ada.Containers.Hashed_Maps [6166], page 841, [6167], page 841, [6168], page 841,
 <in> Ada.Containers.Hashed_Sets [6274], page 868, [6275], page 869,
 <in> Ada.Containers.Ordered_Maps [6209], page 848, [6210], page 848, [6211], page 848,
 <in> Ada.Containers.Ordered_Sets [6336], page 877, [6337], page 877,
 <in> Ada.Containers.Vectors [6022], page 783, [6023], page 783, [6024], page 783, [6025],
page 783, [6026], page 783, [6027], page 784, [6028], page 784, [6029], page 784,
 <in> Ada.Strings.Bounded [5257], page 618, [5258], page 618,
 <in> Ada.Strings.Fixed [5200], page 595, [5201], page 595,
 <in> Ada.Strings.Unbounded [5310], page 631, [5311], page 632,
Insert_Space
 <in> Ada.Containers.Vectors [6034], page 784, [6035], page 784,
inspectable object [7394], page 1157,
inspection point [7393], page 1157,
Inspection_Point pragma [7390], page 1157, [7564], page 1243,
instance
 of a generic function [4294], page 456,
 of a generic package [4291], page 456,

- of a generic procedure [4293], page 456,
- of a generic subprogram [4292], page 456,
- of a generic unit [4256], page 454,
- instructions for comment submission [1003], page 13,
- int
 - <in> Interfaces.C [6475], page 902,
- Integer [1676], page 96, [1696], page 97,
 - <in> Standard [4838], page 557,
- integer literal [1201], page 39,
- integer literals [1683], page 96, [1705], page 98,
- integer type [1653], page 95, [7704], page 1285,
- Integer_Address
 - <in> System.Storage_Elements [4637], page 517,
- Integer_IO
 - <in> Ada.Text_IO [5757], page 703,
- Integer_Text_IO
 - <child of> Ada [5828], page 730,
- integer_type_definition [1657], page 95,
 - <used> [1364], page 54, [7807], page 1290,
- Integer_Wide_Text_IO
 - <child of> Ada [5849], page 745,
- Integer_Wide_Wide_Text_IO
 - <child of> Ada [5852], page 745,
- interaction
 - between tasks [3491], page 328,
- interface [2120], page 152,
 - limited [2125], page 152,
 - nonlimited [2126], page 152,
 - protected [2123], page 152,
 - synchronized [2122], page 152,
 - task [2124], page 152,
 - type [2121], page 152,
- interface to assembly language [6660], page 949,
- interface to C [6468], page 901,
- interface to COBOL [6579], page 931,
- interface to Fortran [6633], page 945,

interface to other languages [6399], page 894,
interface type [7705], page 1285,
Interface_Ancessor_Tags
 <in> Ada.Tags [2038], page 138,
interface_list [2117], page 152,
 <used> [1487], page 66, [2116], page 152, [3263], page 283, [3511], page 329, [3515],
page 329, [3568], page 338, [3572], page 338, [4353], page 463, [8233], page 1302,
interface_type_definition [2115], page 152,
 <used> [1370], page 54, [4375], page 470, [8441], page 1309,
Interfaces [6465], page 900,
Interfaces.C [6470], page 902,
Interfaces.C.Pointers [6554], page 923,
Interfaces.C.Strings [6532], page 916,
Interfaces.COBOLE [6581], page 932,
Interfaces.Fortran [6635], page 945,
interfacing pragma [6402], page 894,
 Convention [6407], page 894,
 Export [6405], page 894,
 Import [6403], page 894,
internal call [3617], page 344,
internal code [4551], page 500,
internal requeue [3620], page 344,
Internal_Tag
 <in> Ada.Tags [2033], page 138,
interpretation
 of a complete context [3467], page 324,
 of a constituent of a complete context [3473], page 325,
 overload resolution [3472], page 325,
interrupt [6666], page 951,
 example using asynchronous_select [3865], page 385, [3870], page 385,
interrupt entry [7436], page 1171,
interrupt handler [6674], page 951,
Interrupt_Handler pragma [6678], page 954, [7568], page 1243,
Interrupt_ID
 <in> Ada.Interrupts [6698], page 957,
Interrupt_Priority pragma [6797], page 975, [7571], page 1243,

Interrupt_Priority <subtype of> Any_Priority

<in> System [4628], page 512,

Interrupts

<child of> Ada [6697], page 957,

Intersection

<in> Ada.Containers.Hashed_Sets [6283], page 869, [6284], page 870,

<in> Ada.Containers.Ordered_Sets [6347], page 878, [6348], page 879,

intertask communication [3614], page 343,

<See also> task [3495], page 328,

Intrinsic calling convention [3111], page 263,

invalid cursor

of a list container [6132], page 829,

of a map [6146], page 839,

of a set [6254], page 866,

of a vector [6074], page 809,

invalid representation [4670], page 523,

Inverse

<in> Ada.Numerics.Generic_Complex_Arrays [7366], page 1135,

<in> Ada.Numerics.Generic_Real_Arrays [7326], page 1121,

Inverted_Exclamation

<in> Ada.Characters.Latin_1 [5027], page 577,

Inverted_Question

<in> Ada.Characters.Latin_1 [5059], page 578,

involve an inner product

complex [7372], page 1137,

real [7332], page 1122,

IO_Exceptions

<child of> Ada [5905], page 752,

IS1

<in> Ada.Characters.Latin_1 [4992], page 576,

IS2

<in> Ada.Characters.Latin_1 [4991], page 576,

IS3

<in> Ada.Characters.Latin_1 [4990], page 576,

IS4

<in> Ada.Characters.Latin_1 [4989], page 576,

Is_A_Group_Member

<in> Ada.Execution_Time.Group_Budgets [7025], page 1028,

Is_Alphanumeric

<in> Ada.Characters.Handling [4870], page 567,

Is_Attached

<in> Ada.Interrupts [6701], page 957,

Is_Basic

<in> Ada.Characters.Handling [4866], page 566,

Is_Callable

<in> Ada.Task_Identification [6753], page 965,

Is_Character

<in> Ada.Characters.Conversions [5125], page 580,

Is_Control

<in> Ada.Characters.Handling [4861], page 566,

Is_Decimal_Digit

<in> Ada.Characters.Handling [4868], page 566,

Is_Descendant_At_Same_Level

<in> Ada.Tags [2035], page 138,

Is_Digit

<in> Ada.Characters.Handling [4867], page 566,

Is_Empty

<in> Ada.Containers.Doubly_Linked_Lists [6088], page 811,

<in> Ada.Containers.Hashed_Maps [6158], page 840,

<in> Ada.Containers.Hashed_Sets [6268], page 868,

<in> Ada.Containers.Ordered_Maps [6201], page 847,

<in> Ada.Containers.Ordered_Sets [6330], page 877,

<in> Ada.Containers.Vectors [6009], page 781,

Is_Graphic

<in> Ada.Characters.Handling [4862], page 566,

Is_Held

<in> Ada.Asynchronous_Task_Control [6975], page 1016,

Is_Hexadecimal_Digit

<in> Ada.Characters.Handling [4869], page 566,

Is_In

<in> Ada.Strings.Maps [5164], page 586,

<in> Ada.Strings.Wide_Maps [5355], page 638,

<in> Ada.Strings.Wide_Wide_Maps [5386], page 643,
Is_ISO_646
<in> Ada.Characters.Handling [4879], page 567,
Is_Letter
<in> Ada.Characters.Handling [4863], page 566,
Is_Lower
<in> Ada.Characters.Handling [4864], page 566,
Is_Member
<in> Ada.Execution_Time.Group_Budgets [7024], page 1028,
Is_Nul_Terminated
<in> Interfaces.C [6494], page 903, [6504], page 905, [6524], page 907, [6514], page 906,
Is_Open
<in> Ada.Direct_IO [5633], page 692,
<in> Ada.Sequential_IO [5608], page 684,
<in> Ada.Streams.Stream_IO [5876], page 747,
<in> Ada.Text_IO [5686], page 699,
Is_Reserved
<in> Ada.Interrupts [6700], page 957,
Is_Round_Robin
<in> Ada.Dispatching.Round_Robin [6848], page 986,
Is_Sorted
<in> Ada.Containers.Doubly_Linked_Lists [6124], page 815,
<in> Ada.Containers.Vectors [6062], page 787,
Is_Special
<in> Ada.Characters.Handling [4871], page 567,
Is_String
<in> Ada.Characters.Conversions [5126], page 580,
Is_Subset
<in> Ada.Containers.Hashed_Sets [6290], page 870,
<in> Ada.Containers.Ordered_Sets [6354], page 879,
<in> Ada.Strings.Maps [5165], page 586,
<in> Ada.Strings.Wide_Maps [5356], page 638,
<in> Ada.Strings.Wide_Wide_Maps [5387], page 643,
Is_Terminated
<in> Ada.Task_Identification [6752], page 965,
Is_Upper

<in> Ada.Characters.Handling [4865], page 566,
Is_Wide_Character
<in> Ada.Characters.Conversions [5129], page 580,
Is_Wide_String
<in> Ada.Characters.Conversions [5130], page 580,
ISO 1989:2002 [1094], page 30,
ISO 8601:2004 [1100], page 30,
ISO/IEC 10646:2003 [1108], page 30, [1640], page 93, [1648], page 94, [1645], page 94,
ISO/IEC 14882:2003 [1111], page 30,
ISO/IEC 1539–1:2004 [1091], page 30,
ISO/IEC 6429:1992 [1097], page 30,
ISO/IEC 646:1991 [1088], page 30,
ISO/IEC 8859–1:1987 [1102], page 30,
ISO/IEC 9899:1999 [1105], page 30,
ISO/IEC TR 19769:2004 [1114], page 30,
ISO_646 <subtype of> Character
<in> Ada.Characters.Handling [4878], page 567,
ISO_646_Set
<in> Ada.Strings.Maps.Constants [5335], page 636,
issue
an entry call [3701], page 353,
italics
nongraphic characters [1642], page 94,
pseudo–names of anonymous types [1374], page 54, [4835], page 556,
syntax rules [1065], page 26,
terms introduced or defined [1116], page 30,
Iterate
<in> Ada.Containers.Doubly_Linked_Lists [6121], page 815,
<in> Ada.Containers.Hashed_Maps [6184], page 843,
<in> Ada.Containers.Hashed_Sets [6300], page 871,
<in> Ada.Containers.Ordered_Maps [6235], page 850,
<in> Ada.Containers.Ordered_Sets [6368], page 881,
<in> Ada.Containers.Vectors [6059], page 787,
<in> Ada.Environment_Variables [5982], page 775,
iteration_scheme [2983], page 249,
<used> [2980], page 248, [8110], page 1299,

30.11 J

j

<in> Ada.Numerics.Generic_Complex_Types [7211], page 1084,
<in> Interfaces.Fortran [6644], page 946,

30.12 K

Key

<in> Ada.Containers.Hashed_Maps [6160], page 840,
<in> Ada.Containers.Hashed_Sets [6302], page 872,
<in> Ada.Containers.Ordered_Maps [6203], page 847,
<in> Ada.Containers.Ordered_Sets [6372], page 882,

Kind

<in> Ada.Directories [5946], page 759, [5958], page 760,
known discriminants [1919], page 125,
known_discriminant_part [1897], page 123,
<used> [1358], page 53, [1895], page 123, [3510], page 329, [3567], page 338, [8248],
page 1303,

30.13 L

label [2930], page 241,

<used> [2904], page 240, [8071], page 1297,

Landau symbol $O(X)$ [5987], page 779,

language

- interface to assembly [6661], page 949,
- interface to non-Ada [6400], page 894,
- language-defined categories
 - [<partial>] [1350], page 52,
- language-defined category
 - of types [1320], page 51,
- language-defined check [4175], page 431, [4212], page 448,
- language-defined class
 - [<partial>] [1349], page 52,
 - of types [1319], page 51,
- Language-Defined Library Units [4831], page 553,
- Last
 - <in> Ada.Containers.Doubly_Linked_Lists [6111], page 814,
 - <in> Ada.Containers.Ordered_Maps [6222], page 849,
 - <in> Ada.Containers.Ordered_Sets [6357], page 880,
 - <in> Ada.Containers.Vectors [6047], page 786,
- Last attribute [1559], page 77, [1877], page 120,
- last element
 - of a hashed set [6316], page 873,
 - of a ordered set [6386], page 883,
 - of a set [6248], page 855,
- last node
 - of a hashed map [6191], page 844,
 - of a map [6142], page 830,
 - of an ordered map [6241], page 851,
- Last(N) attribute [1879], page 120,
- last_bit [4573], page 502,
 - <used> [4568], page 502, [8481], page 1310,
- Last_Bit attribute [4586], page 507,
- Last_Element
 - <in> Ada.Containers.Doubly_Linked_Lists [6112], page 814,
 - <in> Ada.Containers.Ordered_Maps [6223], page 849,
 - <in> Ada.Containers.Ordered_Sets [6358], page 880,
 - <in> Ada.Containers.Vectors [6048], page 786,
- Last_Index
 - <in> Ada.Containers.Vectors [6046], page 786,

Last_Key

<in> Ada.Containers.Ordered_Maps [6224], page 849,

lateness [6961], page 1014,

Latin-1 [1638], page 93,

Latin_1

<child of> Ada.Characters [4893], page 573,

layout

aspect of representation [4554], page 501,

Layout_Error

<in> Ada.IO.Exceptions [5913], page 752,

<in> Ada.Text_IO [5821], page 708,

LC_A

<in> Ada.Characters.Latin_1 [4957], page 575,

LC_A.Acute

<in> Ada.Characters.Latin_1 [5093], page 579,

LC_A.Circumflex

<in> Ada.Characters.Latin_1 [5094], page 579,

LC_A.Diaeresis

<in> Ada.Characters.Latin_1 [5096], page 579,

LC_A.Grave

<in> Ada.Characters.Latin_1 [5092], page 579,

LC_A.Ring

<in> Ada.Characters.Latin_1 [5097], page 579,

LC_A.Tilde

<in> Ada.Characters.Latin_1 [5095], page 579,

LC_AE.Diphthong

<in> Ada.Characters.Latin_1 [5098], page 579,

LC_B

<in> Ada.Characters.Latin_1 [4958], page 575,

LC_C

<in> Ada.Characters.Latin_1 [4959], page 575,

LC_C.Cedilla

<in> Ada.Characters.Latin_1 [5099], page 579,

LC_D

<in> Ada.Characters.Latin_1 [4960], page 575,

LC_E

<in> Ada.Characters.Latin_1 [4961], page 575,
LC_E.Acute
<in> Ada.Characters.Latin_1 [5101], page 579,
LC_E.Circumflex
<in> Ada.Characters.Latin_1 [5102], page 579,
LC_E.Diaeresis
<in> Ada.Characters.Latin_1 [5103], page 579,
LC_E.Grave
<in> Ada.Characters.Latin_1 [5100], page 579,
LC_F
<in> Ada.Characters.Latin_1 [4962], page 575,
LC_G
<in> Ada.Characters.Latin_1 [4963], page 575,
LC_German.Sharp_S
<in> Ada.Characters.Latin_1 [5091], page 579,
LC_H
<in> Ada.Characters.Latin_1 [4964], page 575,
LC_I
<in> Ada.Characters.Latin_1 [4965], page 575,
LC_I.Acute
<in> Ada.Characters.Latin_1 [5105], page 579,
LC_I.Circumflex
<in> Ada.Characters.Latin_1 [5106], page 579,
LC_I.Diaeresis
<in> Ada.Characters.Latin_1 [5107], page 579,
LC_I.Grave
<in> Ada.Characters.Latin_1 [5104], page 579,
LC_Icelandic_Eth
<in> Ada.Characters.Latin_1 [5108], page 579,
LC_Icelandic_Thorn
<in> Ada.Characters.Latin_1 [5122], page 579,
LC_J
<in> Ada.Characters.Latin_1 [4966], page 575,
LC_K
<in> Ada.Characters.Latin_1 [4967], page 575,
LC_L

<in> Ada.Characters.Latin_1 [4968], page 575,
LC_M
<in> Ada.Characters.Latin_1 [4969], page 575,
LC_N
<in> Ada.Characters.Latin_1 [4970], page 575,
LC_N_Tilde
<in> Ada.Characters.Latin_1 [5109], page 579,
LC_O
<in> Ada.Characters.Latin_1 [4971], page 575,
LC_O_Acute
<in> Ada.Characters.Latin_1 [5111], page 579,
LC_O_Circumflex
<in> Ada.Characters.Latin_1 [5112], page 579,
LC_O_Diaeresis
<in> Ada.Characters.Latin_1 [5114], page 579,
LC_O_Grave
<in> Ada.Characters.Latin_1 [5110], page 579,
LC_O_Oblique_Stroke
<in> Ada.Characters.Latin_1 [5116], page 579,
LC_O_Tilde
<in> Ada.Characters.Latin_1 [5113], page 579,
LC_P
<in> Ada.Characters.Latin_1 [4972], page 575,
LC_Q
<in> Ada.Characters.Latin_1 [4973], page 575,
LC_R
<in> Ada.Characters.Latin_1 [4974], page 575,
LC_S
<in> Ada.Characters.Latin_1 [4975], page 575,
LC_T
<in> Ada.Characters.Latin_1 [4976], page 575,
LC_U
<in> Ada.Characters.Latin_1 [4977], page 575,
LC_U_Acute
<in> Ada.Characters.Latin_1 [5118], page 579,
LC_U_Circumflex

<in> Ada.Characters.Latin_1 [5119], page 579,
LC_U_Diaeresis
<in> Ada.Characters.Latin_1 [5120], page 579,
LC_U_Grave
<in> Ada.Characters.Latin_1 [5117], page 579,
LC_V
<in> Ada.Characters.Latin_1 [4978], page 575,
LC_W
<in> Ada.Characters.Latin_1 [4979], page 575,
LC_X
<in> Ada.Characters.Latin_1 [4980], page 576,
LC_Y
<in> Ada.Characters.Latin_1 [4981], page 576,
LC_Y_Acute
<in> Ada.Characters.Latin_1 [5121], page 579,
LC_Y_Diaeresis
<in> Ada.Characters.Latin_1 [5123], page 579,
LC_Z
<in> Ada.Characters.Latin_1 [4982], page 576,
Leading_Nonseparate
<in> Interfaces.COBOLE [6603], page 933,
Leading_Part attribute [5545], page 674,
Leading_Separate
<in> Interfaces.COBOLE [6601], page 933,
Leap_Seconds_Count <subtype of> Integer
<in> Ada.Calendar.Arithmetic [3771], page 365,
leaving [3324], page 300,
left [3325], page 300,
left parenthesis [1150], page 36,
Left_Angle_Quotation
<in> Ada.Characters.Latin_1 [5037], page 577,
Left_Curly_Bracket
<in> Ada.Characters.Latin_1 [4983], page 576,
Left_Parenthesis
<in> Ada.Characters.Latin_1 [4935], page 574,
Left_Square_Bracket

- `<in>` `Ada.Characters.Latin_1` [4951], page 575,
- legal
 - `construct` [1025], page 22,
 - `partition` [1033], page 22,
- legality rules [1022], page 22,
- length
 - of a dimension of an array [1845], page 115,
 - of a list container [6081], page 810,
 - of a map [6140], page 830,
 - of a one-dimensional array [1846], page 115,
 - of a set [6246], page 855,
 - of a vector container [5993], page 779,
- `<in>` `Ada.Containers.Doubly_Linked_Lists` [6087], page 811,
- `<in>` `Ada.Containers.Hashed_Maps` [6157], page 840,
- `<in>` `Ada.Containers.Hashed_Sets` [6267], page 868,
- `<in>` `Ada.Containers.Ordered_Maps` [6200], page 847,
- `<in>` `Ada.Containers.Ordered_Sets` [6329], page 877,
- `<in>` `Ada.Containers.Vectors` [6007], page 781,
- `<in>` `Ada.Strings.Bounded` [5222], page 611,
- `<in>` `Ada.Strings.Unbounded` [5277], page 625,
- `<in>` `Ada.Text_IO.Editing` [7196], page 1074,
- `<in>` `Interfaces.COBOL` [6617], page 934, [6621], page 935, [6625], page 935,
- `Length` attribute [1885], page 120,
- `Length(N)` attribute [1887], page 121,
- `Length_Check` [4197], page 437,
- `[<partial>]` [2602], page 205, [2782], page 224, [2812], page 227,
- `Length_Error`
 - `<in>` `Ada.Strings` [5147], page 584,
- `Length_Range` `<subtype of>` `Natural`
 - `<in>` `Ada.Strings.Bounded` [5221], page 611,
- less than operator [2472], page 201, [2624], page 206,
- less than or equal operator [2476], page 201, [2628], page 206,
- less-than sign [1165], page 36,
- `Less_Than_Sign`
 - `<in>` `Ada.Characters.Latin_1` [4946], page 575,
- letter

a category of Character [4885], page 568,
letter_lowercase [1126], page 33,
<used> [1187], page 38, [7744], page 1289,
letter_modifier [1128], page 33,
<used> [1189], page 38, [7746], page 1289,
letter_other [1129], page 33,
<used> [1190], page 38, [7747], page 1289,

Letter_Set

<in> Ada.Strings.Maps.Constants [5327], page 636,
letter_titlecase [1127], page 33,
<used> [1188], page 38, [7745], page 1289,
letter_uppercase [1125], page 32,
<used> [1186], page 38, [7743], page 1289,

level

accessibility [2210], page 164,
library [2221], page 168,
lexical element [1174], page 36,
lexicographic order [2648], page 209,

LF

<in> Ada.Characters.Latin_1 [4905], page 573,
library [4006], page 407,
[<partial>] [3941], page 395,
informal introduction [3906], page 394,
<See also> library level, library unit, library_item
library level [2220], page 168,
Library unit [3912], page 394, [3940], page 395, [7706], page 1285,
informal introduction [3904], page 394,
<See also> language–defined library units
library unit pragma [4011], page 408,
All_Calls_Remote [7107], page 1042,
categorization pragmas [7071], page 1037,
Elaborate_Body [4083], page 418,
Preelaborate [4054], page 414,
Pure [4068], page 416,
library_item [3922], page 395,
informal introduction [3905], page 394,

<used> [3919], page 395, [8333], page 1306,
library_unit_body [3935], page 395,
 <used> [3924], page 395, [8337], page 1306,
library_unit_declaration [3926], page 395,
 <used> [3923], page 395, [8336], page 1306,
library_unit_renaming_declaration [3931], page 395,
 <used> [3925], page 395, [8338], page 1306,
lifetime [2214], page 164,
limited interface [2130], page 152,
limited type [3285], page 293, [7708], page 1285,
 becoming nonlimited [3277], page 287, [3289], page 294,
limited view [3958], page 396,
Limited_Controlled
 <in> Ada.Finalization [3304], page 296,
limited_with_clause [3969], page 400,
 <used> [3967], page 400, [8352], page 1306,
line [1176], page 36,
 <in> Ada.Text_IO [5734], page 702,
line terminator [5658], page 697,
Line_Length
 <in> Ada.Text_IO [5709], page 700,
link name [6457], page 897,
link-time error
 <See> post-Compilation error [1031], page 22,
 <See> post-Compilation error [1074], page 28,
Linker_Options pragma [6427], page 895, [7574], page 1243,
linking
 <See> partition building [4036], page 410,
List
 <in> Ada.Containers.Doubly_Linked_Lists [6083], page 811,
list container [6078], page 810,
List pragma [1272], page 46, [7577], page 1243,
literal [2356], page 189,
 based [1218], page 41,
 decimal [1205], page 40,
 numeric [1199], page 39,

<See also> aggregate [2370], page 190,
little_endian [4594], page 508,
load_time [6712], page 960,
local_to [3349], page 305,
local_name [4422], page 481,
<used> [3210], page 275, [4452], page 486, [4463], page 487, [4544], page 500, [4561],
page 502, [4565], page 502, [4733], page 534, [6413], page 894, [6419], page 894, [6425],
page 895, [6573], page 928, [6715], page 962, [6722], page 963, [6725], page 963, [6728],
page 963, [6731], page 963, [7148], page 1047, [7513], page 1242, [7516], page 1242, [7519],
page 1242, [7526], page 1242, [7530], page 1242, [7535], page 1242, [7550], page 1242, [7556],
page 1243, [7584], page 1243, [7593], page 1243, [7652], page 1244, [7658], page 1244, [7661],
page 1244, [8475], page 1310,
locking_policy [6867], page 991,
Locking_Policy_pragma [6861], page 991, [7580], page 1243,
Log
<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7236], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5412], page 649,
Logical
<in> Interfaces.Fortran [6639], page 945,
logical_operator [2592], page 205,
<See also> not_operator [2710], page 218,
logical_operator [2573], page 203,
long
<in> Interfaces.C [6477], page 902,
Long_Binary
<in> Interfaces.COBOLE [6585], page 932,
long_double
<in> Interfaces.C [6488], page 903,
Long_Float [1750], page 104, [1752], page 104, [1753], page 105,
Long_Floating
<in> Interfaces.COBOLE [6583], page 932,
Long_Integer [1697], page 97, [1698], page 97, [1702], page 98,
Look_Ahead
<in> Ada.Text_IO [5742], page 702,
loop_parameter [2989], page 249,
loop_parameter_specification [2986], page 249,

- <used> [2985], page 249, [8114], page 1299,
- loop_statement [2978], page 248,
 - <used> [2924], page 240, [8089], page 1298,
- low line [1168], page 36,
- low-level programming [6654], page 949,
- Low_Line
 - <in> Ada.Characters.Latin_1 [4955], page 575,
- Low_Order_First [4593], page 508,
 - <in> Interfaces.COBOL [6607], page 933,
 - <in> System [4624], page 512,
- lower bound
 - of a range [1538], page 76,
- lower-case letter
 - a category of Character [4886], page 568,
- Lower_Case_Map
 - <in> Ada.Strings.Maps.Constants [5336], page 636,
- Lower_Set
 - <in> Ada.Strings.Maps.Constants [5328], page 636,

30.14 M

- Machine attribute [5550], page 675,
- machine code insertion [4647], page 518, [6659], page 949,
- machine numbers
 - of a fixed point type [1779], page 107,
 - of a floating point type [1738], page 104,
- machine scalar [4494], page 487,
- Machine_Code
 - <child of> System [4652], page 519,
- Machine_Emax attribute [5487], page 664,
- Machine_Emin attribute [5485], page 664,
- Machine_Mantissa attribute [5483], page 663,
- Machine_Overflows attribute [5497], page 665, [5575], page 680,

Machine.Radix attribute [5480], page 663, [5571], page 679,
Machine.Radix clause [4490], page 487, [7171], page 1054,
Machine.Rounding attribute [5525], page 671,
Machine.Rounds attribute [5495], page 665, [5573], page 679,
macro
 <See> generic unit [4229], page 450,
Macron
 <in> Ada.Characters.Latin_1 [5041], page 577,
main subprogram
 for a partition [4039], page 410,
malloc
 <See> allocator [2842], page 230,
Map
 <in> Ada.Containers.Hashed_Maps [6151], page 840,
 <in> Ada.Containers.Ordered_Maps [6196], page 846,
map container [6136], page 830,
Maps
 <child of> Ada.Strings [5156], page 585,
mark_non_spacing [1130], page 33, [1131], page 33,
 <used> [1193], page 38, [7749], page 1289,
mark_spacing_combining
 <used> [1194], page 38, [7750], page 1289,
marshalling [7131], page 1045,
Masculine_Ordinal_Indicator
 <in> Ada.Characters.Latin_1 [5054], page 578,
master [3326], page 300,
match
 a character to a pattern character [5177], page 590,
 a character to a pattern character, with respect to a character mapping function [5179],
page 591,
 a string to a pattern string [5178], page 590,
matching components [2646], page 208,
Max attribute [1568], page 78,
Max_Base_Digits [1736], page 103,
 <in> System [4611], page 511,
Max_Binary_Modulus [1668], page 95,

<in> System [4609], page 511,
Max_Decimal_Digits
 <in> Ada.Decimal [7177], page 1055,
Max_Delta
 <in> Ada.Decimal [7176], page 1055,
Max_Digits [1737], page 103,
 <in> System [4612], page 511,
Max_Digits_Binary
 <in> Interfaces.COBOL [6586], page 932,
Max_Digits_Long_Binary
 <in> Interfaces.COBOL [6587], page 932,
Max_Image_Width
 <in> Ada.Numerics.Discrete_Random [5467], page 656,
 <in> Ada.Numerics.Float_Random [5456], page 655,
Max_Int [1681], page 96,
 <in> System [4608], page 511,
Max_Length
 <in> Ada.Strings.Bounded [5218], page 611,
Max_Mantissa
 <in> System [4613], page 511,
Max_Nonbinary_Modulus [1669], page 95,
 <in> System [4610], page 511,
Max_Picture_Length
 <in> Ada.Text_IO.Editing [7189], page 1073,
Max_Scale
 <in> Ada.Decimal [7173], page 1055,
Max_Size_In_Storage_Elements attribute [4713], page 531,
maximum box error
 for a component of the result of evaluating a complex function [7315], page 1116,
maximum line length [5666], page 698,
maximum page length [5667], page 698,
maximum relative error
 for a component of the result of evaluating a complex function [7314], page 1116,
 for the evaluation of an elementary function [7308], page 1113,
Members
 <in> Ada.Execution_Time.Group_Budgets [7026], page 1028,

Membership

<in> Ada.Strings [5153], page 585,

membership test [2639], page 206,

Memory_Size

<in> System [4620], page 511,

Merge

<in> Ada.Containers.Doubly_Linked_Lists [6126], page 815,

<in> Ada.Containers.Vectors [6064], page 787,

message

<See> dispatching call [2080], page 145,

method

<See> dispatching subprogram [2081], page 145,

metrics [1045], page 22,

Micro_Sign

<in> Ada.Characters.Latin_1 [5048], page 577,

Microseconds

<in> Ada.Real_Time [6946], page 1009,

Middle_Dot

<in> Ada.Characters.Latin_1 [5051], page 578,

Milliseconds

<in> Ada.Real_Time [6947], page 1009,

Min attribute [1566], page 78,

Min_Delta

<in> Ada.Decimal [7175], page 1055,

Min_Handler_Ceiling

<in> Ada.Execution_Time.Group_Budgets [7021], page 1028,

<in> Ada.Execution_Time.Timers [7005], page 1025,

Min_Int [1680], page 96,

<in> System [4607], page 510,

Min_Scale

<in> Ada.Decimal [7174], page 1055,

minus [1157], page 36,

minus operator [2494], page 201, [2658], page 211, [2681], page 213,

Minus_Sign

<in> Ada.Characters.Latin_1 [4941], page 574,

Minute

<in> Ada.Calendar.Formatting [3791], page 366,
Minute_Number <subtype of> Natural
<in> Ada.Calendar.Formatting [3784], page 366,
Minutes
<in> Ada.Real_Time [6949], page 1009,
mixed–language programs [6401], page 894, [6662], page 949,
Mod attribute [1686], page 96,
mod operator [2513], page 201, [2695], page 213,
mod_clause [7444], page 1174,
<used> [4562], page 502, [8476], page 1310,
mode [3063], page 256,
<used> [3056], page 256, [4302], page 458, [8420], page 1308,
<in> Ada.Direct_IO [5630], page 691,
<in> Ada.Sequential_IO [5605], page 684,
<in> Ada.Streams.Stream_IO [5873], page 747,
<in> Ada.Text_IO [5683], page 699,
mode conformance [3120], page 264,
 required [3449], page 320, [3452], page 320, [4373], page 469, [4393], page 471, [4394],
page 471, [4475], page 487,
mode of operation
 nonstandard [1081], page 29,
 standard [1083], page 29,
Mode_Error
<in> Ada.Direct_IO [5643], page 692,
<in> Ada.IO_Exceptions [5907], page 752,
<in> Ada.Sequential_IO [5613], page 684,
<in> Ada.Streams.Stream_IO [5889], page 748,
<in> Ada.Text_IO [5815], page 708,
Model attribute [5564], page 678, [7299], page 1108,
model interval [7282], page 1104,
 associated with a value [7283], page 1104,
model number [7281], page 1104,
model–oriented attributes
 of a floating point subtype [5554], page 676,
Model_Emin attribute [5558], page 677, [7292], page 1106,
Model_Epsilon attribute [5560], page 677,

Model_Mantissa attribute [5556], page 676, [7290], page 1105,
Model_Small attribute [5562], page 677,
Modification_Time
 <in> Ada.Directories [5948], page 759, [5960], page 761,
modular type [1655], page 95,
Modular_IO
 <in> Ada.Text_IO [5766], page 704,
modular_type_definition [1663], page 95,
 <used> [1659], page 95, [7855], page 1291,
module
 <See> package [3227], page 279,
modulus
 of a modular type [1667], page 95,
 <in> Ada.Numerics.Generic_Complex_Arrays [7344], page 1131, [7357], page 1133,
 <in> Ada.Numerics.Generic_Complex_Types [7221], page 1085,
Modulus attribute [1688], page 97,
Monday
 <in> Ada.Calendar.Formatting [3775], page 365,
Month
 <in> Ada.Calendar [3749], page 360,
 <in> Ada.Calendar.Formatting [3788], page 366,
Month_Number <subtype of> Integer
 <in> Ada.Calendar [3744], page 360,
More_Entries
 <in> Ada.Directories [5954], page 760,
Move
 <in> Ada.Containers.Doubly_Linked_Lists [6094], page 812,
 <in> Ada.Containers.Hashed_Maps [6165], page 841,
 <in> Ada.Containers.Hashed_Sets [6273], page 868,
 <in> Ada.Containers.Ordered_Maps [6208], page 848,
 <in> Ada.Containers.Ordered_Sets [6335], page 877,
 <in> Ada.Containers.Vectors [6021], page 783,
 <in> Ada.Strings.Fixed [5181], page 592,
multi-dimensional array [1842], page 115,
Multiplication_Sign
 <in> Ada.Characters.Latin_1 [5083], page 578,

multiply [1153], page 36,
multiply operator [2505], page 201, [2687], page 213,
multiplying operator [2683], page 213,
multiplying_operator [2577], page 203,
<used> [2552], page 201, [8046], page 1297,
MW
<in> Ada.Characters.Latin_1 [5014], page 576,

30.15 N

n-dimensional array_aggregate [2434], page 197,
NAK
<in> Ada.Characters.Latin_1 [4916], page 574,
name [2270], page 179,
[<partial>] [1281], page 49,
of (a view of) an entity [1304], page 50,
of a pragma [1263], page 45,
of an external file [5579], page 680,
<used> [1260], page 44, [1390], page 56, [2284], page 179, [2287], page 179, [2289],
page 179, [2563], page 201, [2733], page 219, [2938], page 242, [3003], page 252, [3008],
page 253, [3137], page 266, [3141], page 267, [3145], page 267, [3156], page 267, [3405],
page 314, [3432], page 317, [3438], page 318, [3441], page 319, [3447], page 320, [3458],
page 323, [3689], page 352, [3717], page 356, [3874], page 385, [3939], page 395, [3970],
page 400, [3973], page 400, [4053], page 413, [4067], page 416, [4075], page 418, [4079],
page 418, [4082], page 418, [4119], page 420, [4124], page 421, [4259], page 455, [4279],
page 455, [4390], page 471, [4401], page 475, [4426], page 481, [4468], page 487, [4748],
page 535, [6679], page 954, [6682], page 954, [7082], page 1038, [7094], page 1039, [7103],
page 1042, [7106], page 1042, [7392], page 1157, [7452], page 1175, [7503], page 1242, [7522],
page 1242, [7539], page 1242, [7543], page 1242, [7546], page 1242, [7562], page 1243, [7565],
page 1243, [7569], page 1243, [7604], page 1243, [7619], page 1244, [7628], page 1244, [7631],
page 1244, [7640], page 1244, [8292], page 1304,
<in> Ada.Direct_IO [5631], page 691,
<in> Ada.Sequential_IO [5606], page 684,

- <in> Ada.Streams.Stream_IO [5874], page 747,
- <in> Ada.Text_IO [5684], page 699,
- <in> System [4605], page 510,
- name resolution rules [1019], page 22,
- Name_Error
 - <in> Ada.Direct_IO [5644], page 692,
 - <in> Ada.Directories [5962], page 761,
 - <in> Ada.IO_Exceptions [5908], page 752,
 - <in> Ada.Sequential_IO [5614], page 684,
 - <in> Ada.Streams.Stream_IO [5890], page 748,
 - <in> Ada.Text_IO [5816], page 708,
- named
 - in a use clause [3411], page 315,
 - in a with_clause [3978], page 400,
- named association [3157], page 267, [4282], page 455,
- named component association [2393], page 191,
- named discriminant association [1935], page 128,
- named entry index [3677], page 350,
- named number [1429], page 60,
- named type [1372], page 54,
- named_array_aggregate [2427], page 196,
 - <used> [2417], page 196, [8010], page 1296,
- Names
 - <child of> Ada.Interrupts [6707], page 958,
- Nanoseconds
 - <in> Ada.Real_Time [6945], page 1009,
- Native_Binary
 - <in> Interfaces.COBOL [6608], page 933,
- Natural [1677], page 96,
- Natural <subtype of> Integer
 - <in> Standard [4839], page 557,
- NBH
 - <in> Ada.Characters.Latin_1 [4996], page 576,
- NBSP
 - <in> Ada.Characters.Latin_1 [5026], page 577,
- needed

of a compilation unit by another [4038], page 410,
remote call interface [7115], page 1043,
shared passive library unit [7088], page 1039,
needed component
 extension_aggregate record_component_association_list [2410], page 195,
 record_aggregate record_component_association_list [2396], page 192,
needs finalization [3308], page 297,
NEL
 <in> Ada.Characters.Latin_1 [4998], page 576,
new
 <See> allocator [2841], page 230,
New_Char_Array
 <in> Interfaces.C.Strings [6538], page 916,
New_Line
 <in> Ada.Text_IO [5714], page 701,
New_Page
 <in> Ada.Text_IO [5719], page 701,
New_String
 <in> Interfaces.C.Strings [6539], page 916,
Next
 <in> Ada.Containers.Doubly_Linked_Lists [6113], page 814, [6115], page 814,
 <in> Ada.Containers.Hashed_Maps [6175], page 842, [6176], page 842,
 <in> Ada.Containers.Hashed_Sets [6292], page 870, [6293], page 871,
 <in> Ada.Containers.Ordered_Maps [6225], page 849, [6226], page 849,
 <in> Ada.Containers.Ordered_Sets [6359], page 880, [6360], page 880,
 <in> Ada.Containers.Vectors [6049], page 786, [6050], page 786,
No_Break_Space
 <in> Ada.Characters.Latin_1 [5025], page 577,
No_Element
 <in> Ada.Containers.Doubly_Linked_Lists [6086], page 811,
 <in> Ada.Containers.Hashed_Maps [6154], page 840,
 <in> Ada.Containers.Hashed_Sets [6262], page 867,
 <in> Ada.Containers.Ordered_Maps [6199], page 847,
 <in> Ada.Containers.Ordered_Sets [6326], page 876,
 <in> Ada.Containers.Vectors [6002], page 780,
No_Index

<in> Ada.Containers.Vectors [5998], page 780,
No_Return pragma [3209], page 275, [7583], page 1243,
No_Tag
 <in> Ada.Tags [2028], page 137,
node
 of a list [6080], page 810,
 of a map [6139], page 830,
nominal subtype [1424], page 60, [1456], page 62,
 associated with a dereference [2292], page 180,
 associated with a type_conversion [2768], page 223,
 associated with an indexed_component [2307], page 181,
 of a component [1854], page 116,
 of a formal parameter [3071], page 257,
 of a function result [3072], page 257,
 of a generic formal object [4316], page 460,
 of a record component [1973], page 131,
 of the result of a function_call [3161], page 268,
non–normative
 <See> informative [1012], page 21,
non–returning [3212], page 275,
nondispatching call
 on a dispatching operation [2072], page 145,
nonexistent [4721], page 533, [4729], page 534,
nongraphic character [1586], page 81,
nonlimited interface [2131], page 152,
nonlimited type [3286], page 293,
 becoming nonlimited [3276], page 287, [3288], page 294,
nonlimited_with_clause [3972], page 400,
 <used> [3968], page 400, [8353], page 1306,
nonstandard integer type [1700], page 98,
nonstandard mode [1082], page 29,
nonstandard real type [1725], page 102,
normal completion [3320], page 299,
normal library unit [7077], page 1037,
normal state of an object [4223], page 449, [4665], page 522,
 [<partial>] [3887], page 387, [5915], page 754,

Normalize_Scalars pragma [7381], page 1153, [7587], page 1243,
normalized exponent [5500], page 666,
normalized number [5491], page 665,
normative [1009], page 21,
not equal operator [2468], page 201, [2620], page 206,
not in (membership test) [2487], page 201, [2641], page 206,
not operator [2524], page 201, [2708], page 218,

Not_Sign

<in> Ada.Characters.Latin_1 [5038], page 577,
notes [1049], page 23,
notwithstanding [4029], page 409, [6435], page 896, [6459], page 898, [6696], page 956,
[7085], page 1039, [7089], page 1039, [7116], page 1043, [7427], page 1168,

NUL

<in> Ada.Characters.Latin_1 [4895], page 573,
<in> Interfaces.C [6490], page 903,

null access value [2363], page 189,
null array [1868], page 118,
null constraint [1336], page 51,
null extension [2068], page 143,
null pointer

<See> null access value [2364], page 189,
null procedure [3221], page 277,
null range [1541], page 76,
null record [1974], page 131,
null slice [2321], page 183,
null string literal [1250], page 43,
null value

of an access type [2187], page 158,

Null_Address

<in> System [4617], page 511,

Null_Bounded_String

<in> Ada.Strings.Bounded [5220], page 611,

null_exclusion [2159], page 156,

<used> [1386], page 56, [1902], page 123, [2150], page 156, [2161], page 156, [3047],
page 256, [3057], page 256, [3427], page 317, [4303], page 458, [7936], page 1294,

Null_Id

<in> Ada.Exceptions [4142], page 423,
Null_Occurrence
<in> Ada.Exceptions [4148], page 424,
null_procedure_declaration [3218], page 277,
<used> [1289], page 49, [7790], page 1290,
Null_Ptr
<in> Interfaces.C.Strings [6536], page 916,
Null_Set
<in> Ada.Strings.Maps [5158], page 585,
<in> Ada.Strings.Wide_Maps [5349], page 637,
<in> Ada.Strings.Wide_Wide_Maps [5380], page 642,
null_statement [2929], page 241,
<used> [2909], page 240, [8075], page 1297,
Null_Task_Id
<in> Ada.Task_Identification [6748], page 965,
Null_Unbounded_String
<in> Ada.Strings.Unbounded [5276], page 625,
number sign [1146], page 36,
Number_Base <subtype of> Integer
<in> Ada.Text_IO [5675], page 698,
number_decimal [1132], page 33,
<used> [1195], page 38, [7751], page 1289,
number_declaration [1475], page 65,
<used> [1286], page 49, [7787], page 1290,
number_letter [1133], page 34,
<used> [1191], page 38, [7748], page 1289,
Number_Sign
<in> Ada.Characters.Latin_1 [4930], page 574,
numeral [1210], page 40,
<used> [1207], page 40, [1216], page 40, [1234], page 41, [7768], page 1289,
Numeric
<in> Interfaces.COBOLE [6598], page 933,
numeric type [1530], page 76,
numeric_literal [1202], page 40,
<used> [2560], page 201, [8052], page 1297,
numerics [7206], page 1083,

<child of> Ada [5406], page 648,

30.16 O

O(f(N)) [5988], page 779,

object [1415], page 58, [7709], page 1285,

[<partial>] [1316], page 51,

object-oriented programming (OOP)

<See> dispatching operations of tagged types [2078], page 145,

<See> tagged types and type extensions [2010], page 136,

object_declaration [1436], page 61,

<used> [1285], page 49, [7786], page 1290,

object_renaming_declaration [3425], page 317,

<used> [3417], page 316, [8206], page 1302,

obsolescent feature [7422], page 1166,

occur immediately within [3344], page 305,

occurrence

of an interrupt [6667], page 951,

octal

literal [1224], page 41,

octal literal [1222], page 41,

one's complement

modular types [1701], page 98,

one-dimensional array [1841], page 115,

only as a completion

entry_body [3668], page 349,

OOP (object-oriented programming)

<See> dispatching operations of tagged types [2079], page 145,

<See> tagged types and type extensions [2011], page 136,

opaque type

<See> private types and private extensions [3253], page 283,

Open

<in> Ada.Direct_IO [5625], page 691,

- <in> Ada.Sequential_IO [5600], page 684,
- <in> Ada.Streams.Stream_IO [5868], page 746,
- <in> Ada.Text_IO [5678], page 699,
- open alternative [3831], page 379,
- open entry [3691], page 353,
 - of a protected object [3697], page 353,
 - of a task [3695], page 353,
- operand
 - of a qualified_expression [2831], page 230,
 - of a type_conversion [2735], page 219,
- operand interval [7284], page 1104,
- operand type
 - of a type_conversion [2736], page 219,
- operates on a type [1408], page 57,
- operational aspect [4438], page 482,
 - specifiable attributes [4474], page 487,
- operational item [4416], page 481,
- operator [3215], page 276,
 - & [2497], page 201, [2661], page 212,
 - * [2504], page 201, [2686], page 213,
 - ** [2518], page 201, [2717], page 218,
 - + [2489], page 201, [2653], page 211, [2676], page 213,
 - [2493], page 201, [2657], page 211, [2680], page 213,
 - / [2510], page 201, [2692], page 213,
 - /= [2467], page 201, [2619], page 206,
 - < [2471], page 201, [2623], page 206,
 - <= [2475], page 201, [2627], page 206,
 - = [2463], page 201, [2615], page 206,
 - > [2479], page 201, [2631], page 206,
 - >= [2483], page 201, [2635], page 206,
 - abs [2522], page 201, [2706], page 217,
 - ampersand [2499], page 201, [2663], page 212,
 - and [2455], page 201, [2595], page 205,
 - binary [2582], page 203,
 - binary adding [2651], page 211,
 - concatenation [2501], page 201, [2665], page 212,

divide [2512], page 201, [2694], page 213,
equal [2465], page 201, [2617], page 206,
equality [2611], page 206,
exponentiation [2520], page 201, [2715], page 218,
greater than [2481], page 201, [2633], page 206,
greater than or equal [2485], page 201, [2637], page 206,
highest precedence [2704], page 217,
less than [2473], page 201, [2625], page 206,
less than or equal [2477], page 201, [2629], page 206,
logical [2593], page 205,
minus [2495], page 201, [2659], page 211, [2682], page 213,
mod [2514], page 201, [2696], page 213,
multiply [2506], page 201, [2688], page 213,
multiplying [2684], page 213,
not [2525], page 201, [2709], page 218,
not equal [2469], page 201, [2621], page 206,
or [2457], page 201, [2597], page 205,
ordering [2613], page 206,
plus [2491], page 201, [2655], page 211, [2678], page 213,
predefined [2580], page 203,
relational [2608], page 206,
rem [2516], page 201, [2698], page 213,
times [2508], page 201, [2690], page 213,
unary [2584], page 203,
unary adding [2674], page 213,
user-defined [3217], page 276,
xor [2459], page 201, [2599], page 205,
operator precedence [2572], page 203,
operator_symbol [3039], page 256,
 <used> [2282], page 179, [2330], page 184, [3032], page 255, [3042], page 256, [8140],
page 1299,
optimization [4210], page 448, [4216], page 448,
Optimize pragma [1277], page 46, [7589], page 1243,
or else (short-circuit control form) [2461], page 201, [2590], page 204,
or operator [2456], page 201, [2596], page 205,
Ordered_Maps

<child of> Ada.Containers [6194], page 846,
Ordered_Sets
<child of> Ada.Containers [6321], page 876,
ordering operator [2612], page 206,
ordinary file [5967], page 761,
ordinary fixed point type [1758], page 106, [1780], page 107,
ordinary_fixed_point_definition [1764], page 106,
<used> [1762], page 106, [7865], page 1292,
OSC
<in> Ada.Characters.Latin_1 [5022], page 577,
other_control [1141], page 35,
other_format [1135], page 34,
<used> [1197], page 38, [7753], page 1289,
other_private_use [1142], page 35,
other_surrogate [1143], page 35,
output [5577], page 680,
Output attribute [4780], page 543, [4784], page 545,
Output clause [4489], page 487, [4799], page 547,
overall interpretation
of a complete context [3468], page 324,
Overflow_Check [4198], page 438,
[<partial>] [1690], page 97, [2568], page 202, [2975], page 247, [7286], page 1105, [7300],
page 1108, [7304], page 1112, [7309], page 1113, [7316], page 1116,
Overlap
<in> Ada.Containers.Hashed_Sets [6289], page 870,
<in> Ada.Containers.Ordered_Sets [6353], page 879,
overload resolution [3463], page 324,
overloadable [3375], page 308,
overloaded [3374], page 308,
enumeration literal [1633], page 92,
overloading rules [1020], page 22, [3464], page 324,
overridable [3379], page 308,
override [3378], page 308, [4295], page 457,
a primitive subprogram [1412], page 58,
overriding operation [7710], page 1286,
overriding_indicator [3399], page 312,

<used> [2107], page 150, [3018], page 255, [3094], page 261, [3219], page 277, [3445], page 320, [3636], page 347, [3986], page 403, [4261], page 455, [8269], page 1303,

Overwrite

<in> Ada.Strings.Bounded [5259], page 618, [5260], page 619,

<in> Ada.Strings.Fixed [5202], page 595, [5203], page 596,

<in> Ada.Strings.Unbounded [5312], page 632, [5313], page 632,

30.17 P

Pack pragma [4451], page 486, [7592], page 1243,

Package [3224], page 279, [7711], page 1286,

package instance [4287], page 456,

package_body [3243], page 281,

<used> [2255], page 176, [3937], page 395, [7959], page 1295,

package_body_stub [3988], page 403,

<used> [3982], page 403, [8359], page 1306,

package_declaration [3229], page 279,

<used> [1290], page 49, [3928], page 395, [8340], page 1306,

package_renaming_declaration [3439], page 319,

<used> [3419], page 316, [3932], page 395, [8208], page 1302,

package_specification [3231], page 279,

<used> [3230], page 279, [4239], page 450, [8183], page 1301,

packed [4457], page 486,

Packed_Decimal

<in> Interfaces.COBOL [6589], page 932,

Packed_Format

<in> Interfaces.COBOL [6609], page 933,

Packed_Signed

<in> Interfaces.COBOL [6611], page 934,

Packed_Unsigned

<in> Interfaces.COBOL [6610], page 934,

packing

aspect of representation [4456], page 486,

padding bits [4430], page 482,

Page

<in> Ada.Text_IO [5735], page 702,

Page pragma [1275], page 46, [7595], page 1243,

page terminator [5659], page 697,

Page_Length

<in> Ada.Text_IO [5711], page 700,

Paragraph_Sign

<in> Ada.Characters.Latin_1 [5050], page 577,

parallel processing

<See> task [3492], page 328,

parameter

<See> formal parameter [3066], page 257,

<See> generic formal parameter [4230], page 450,

<See also> discriminant [1892], page 123,

<See also> loop parameter [2990], page 249,

parameter assigning back [3175], page 272,

parameter copy back [3173], page 272,

parameter mode [3067], page 257,

parameter passing [3163], page 270,

parameter_and_result_profile [3045], page 256,

<used> [2158], page 156, [2166], page 156, [3028], page 255, [7947], page 1294,

parameter_association [3151], page 267,

<used> [3149], page 267, [8169], page 1300,

parameter_profile [3043], page 256,

<used> [2157], page 156, [2164], page 156, [3025], page 255, [3639], page 347, [3643],
page 347, [3657], page 348, [8275], page 1303,

parameter_specification [3054], page 256,

<used> [3053], page 256, [8148], page 1300,

Parameterless_Handler

<in> Ada.Interrupts [6699], page 957,

Params_Stream_Type

<in> System.RPC [7158], page 1051,

parent [7712], page 1286,

parent body

of a subunit [3997], page 404,

parent declaration
 of a library unit [3947], page 396,
 of a library_item [3946], page 396,
parent subtype [1489], page 66,
parent type [1490], page 66,
parent unit
 of a library unit [3949], page 396,
Parent_Tag
 <in> Ada.Tags [2036], page 138,
parent_unit_name [3938], page 395,
 <used> [3030], page 255, [3037], page 256, [3235], page 279, [3247], page 281, [3995],
page 404, [8192], page 1301,
part
 of an object or value [1334], page 51,
partial view
 of a type [3264], page 284,
partition [4033], page 410, [7713], page 1286,
partition building [4034], page 410,
partition communication subsystem (PCS) [7153], page 1050,
Partition_Check
 [<partial>] [7143], page 1046,
Partition_Elaboration_Policy pragma [7418], page 1164, [7597], page 1243,
Partition_Id
 <in> System.RPC [7156], page 1050,
Partition_Id attribute [7066], page 1035,
pass by copy [3078], page 260,
pass by reference [3081], page 260,
passive partition [7058], page 1035,
Pattern_Error
 <in> Ada.Strings [5148], page 584,
PCS (partition communication subsystem) [7154], page 1050,
pending interrupt occurrence [6670], page 951,
per-object constraint [1980], page 131,
per-object expression [1979], page 131,
percent sign [1172], page 36,
Percent_Sign

<in> Ada.Characters.Latin_1 [4932], page 574,
perfect result set [7302], page 1110,

periodic task

example [3763], page 363,

<See> delay_until_statement [3764], page 363,

Pi

<in> Ada.Numerics [5408], page 648,

Pic_String

<in> Ada.Text_IO.Editing [7187], page 1073,

Picture

<in> Ada.Text_IO.Editing [7184], page 1073,

picture String

for edited output [7180], page 1059,

Picture_Error

<in> Ada.Text_IO.Editing [7190], page 1073,

Pilcrow_Sign

<in> Ada.Characters.Latin_1 [5049], page 577,

plain_char

<in> Interfaces.C [6483], page 902,

plane

character [1122], page 32,

PLD

<in> Ada.Characters.Latin_1 [5004], page 576,

PLU

<in> Ada.Characters.Latin_1 [5005], page 576,

plus operator [2490], page 201, [2654], page 211, [2677], page 213,

plus sign [1154], page 36,

Plus_Minus_Sign

<in> Ada.Characters.Latin_1 [5044], page 577,

Plus_Sign

<in> Ada.Characters.Latin_1 [4938], page 574,

PM

<in> Ada.Characters.Latin_1 [5023], page 577,

point [1160], page 36,

Pointer

<in> Interfaces.C.Pointers [6555], page 923,

<See> access value [2145], page 156,
<See> type System.Address [4630], page 515,
pointer type
<See> access type [2146], page 156,
Pointer_Error
<in> Interfaces.C.Pointers [6558], page 923,
Pointers
<child of> Interfaces.C [6554], page 923,
polymorphism [2006], page 136, [2075], page 145,
pool element [2172], page 157, [4693], page 527,
pool type [4691], page 527,
pool-specific access type [2169], page 157, [2173], page 157,
Pos attribute [1707], page 99,
position [4569], page 502,
<used> [4566], page 502, [8479], page 1310,
Position attribute [4582], page 506,
position number [1529], page 76,
of an enumeration value [1632], page 92,
of an integer value [1684], page 96,
positional association [3158], page 267, [4283], page 455,
positional component association [2394], page 191,
positional discriminant association [1936], page 128,
positional_array_aggregate [2418], page 196,
<used> [2416], page 196, [8009], page 1296,
Positive [1678], page 96,
Positive <subtype of> Integer
<in> Standard [4840], page 558,
Positive_Count <subtype of> Count
<in> Ada.Direct_IO [5623], page 691,
<in> Ada.Streams.Stream_IO [5866], page 746,
<in> Ada.Text_IO [5672], page 698,
possible interpretation [3469], page 325,
for direct_names [3395], page 311,
for selector_names [3396], page 311,
post-Compilation error [1029], page 22,
post-Compilation rules [1030], page 22,

potentially blocking operation [3632], page 345,
 Abort_Task [6759], page 967,
 delay_statement [3761], page 363, [6959], page 1013,
 remote subprogram call [7138], page 1046,
 RPC operations [7167], page 1052,
 Suspend_Until_True [6969], page 1016,
potentially use-visible [3412], page 315,
Pound_Sign
 <in> Ada.Characters.Latin_1 [5029], page 577,
Pragma [1253], page 44, [1254], page 44, [7500], page 1242, [7714], page 1286,
pragma argument [1265], page 45,
pragma name [1264], page 45,
pragma, categorization [7070], page 1037,
 Remote_Call_Interface [7100], page 1041,
 Remote_Types [7091], page 1039,
 Shared_Passive [7079], page 1038,
pragma, configuration [4016], page 408,
 Assertion_Policy [4171], page 427,
 Detect_Blocking [7416], page 1163,
 Discard_Names [6717], page 962,
 Locking_Policy [6866], page 991,
 Normalize Scalars [7383], page 1154,
 Partition_Elaboration_Policy [7421], page 1164,
 Priority_Specific_Dispatching [6834], page 981,
 Profile [6987], page 1020,
 Queuing_Policy [6885], page 995,
 Restrictions [4752], page 535,
 Reviewable [7388], page 1155,
 Suppress [4188], page 431,
 Task_Dispatching_Policy [6832], page 981,
 Unsuppress [4190], page 431,
pragma, identifier specific to [1268], page 45,
pragma, interfacing
 Convention [6408], page 894,
 Export [6406], page 894,
 Import [6404], page 894,

Linker_Options [6409], page 894,
pragma, library unit [4012], page 408,
All_Calls_Remote [7108], page 1042,
categorization pragmas [7072], page 1037,
Elaborate_Body [4084], page 418,
Preelaborate [4055], page 414,
Pure [4069], page 416,
pragma, program unit [4009], page 407,
Convention [6455], page 897,
Export [6453], page 897,
Import [6451], page 897,
Inline [3134], page 266,
library unit pragmas [4014], page 408,
pragma, representation [4415], page 481,
Asynchronous [7150], page 1047,
Atomic [6736], page 964,
Atomic_Components [6740], page 964,
Controlled [4735], page 534,
Convention [6443], page 897,
Discard_Names [6719], page 962,
Export [6441], page 897,
Import [6439], page 897,
Pack [4454], page 486,
Volatile [6738], page 964,
Volatile_Components [6742], page 964,
pragma_argument_association [1258], page 44,
<used> [1257], page 44, [6985], page 1020, [7616], page 1244, [7779], page 1290,
pragmas
All_Calls_Remote [7104], page 1042, [7501], page 1242,
Assert [4163], page 427, [7504], page 1242,
Assertion_Policy [4167], page 427, [7508], page 1242,
Asynchronous [7146], page 1047, [7511], page 1242,
Atomic [6720], page 963, [7514], page 1242,
Atomic_Components [6726], page 963, [7517], page 1242,
Attach_Handler [6680], page 954, [7520], page 1242,
Controlled [4731], page 534, [7524], page 1242,

Convention [6422], page 895, [7527], page 1242,
Detect_Blocking [7413], page 1163, [7531], page 1242,
Discard_Names [6713], page 962, [7533], page 1242,
Elaborate [4072], page 418, [7536], page 1242,
Elaborate_All [4076], page 418, [7540], page 1242,
Elaborate_Body [4080], page 418, [7544], page 1242,
Export [6416], page 894, [7547], page 1242,
Import [6410], page 894, [7553], page 1242,
Inline [3135], page 266, [7559], page 1243,
Inspection_Point [7389], page 1157, [7563], page 1243,
Interrupt_Handler [6677], page 954, [7567], page 1243,
Interrupt_Priority [6796], page 975, [7570], page 1243,
Linker_Options [6426], page 895, [7573], page 1243,
List [1271], page 46, [7576], page 1243,
Locking_Policy [6860], page 991, [7579], page 1243,
No_Return [3208], page 275, [7582], page 1243,
NormalizeScalars [7380], page 1153, [7586], page 1243,
Optimize [1276], page 46, [7588], page 1243,
Pack [4450], page 486, [7591], page 1243,
Page [1274], page 46, [7594], page 1243,
Partition_Elaboration_Policy [7417], page 1164, [7596], page 1243,
Preelaborable_Initialization [4056], page 414, [7599], page 1243,
Preelaborate [4051], page 413, [7602], page 1243,
Priority [6793], page 975, [7605], page 1243,
Priority_Specific_Dispatching [6826], page 981, [7608], page 1243,
Profile [6982], page 1020, [7613], page 1244,
Pure [4065], page 416, [7617], page 1244,
Queuing_Policy [6879], page 994, [7620], page 1244,
Relative_Deadline [6849], page 988, [7623], page 1244,
Remote_Call_Interface [7101], page 1042, [7626], page 1244,
Remote_Types [7092], page 1039, [7629], page 1244,
Restrictions [4739], page 535, [7632], page 1244,
Reviewable [7385], page 1155, [7636], page 1244,
Shared_Passive [7080], page 1038, [7638], page 1244,
Storage_Size [4526], page 496, [7641], page 1244,
Suppress [4181], page 431, [7449], page 1175, [7644], page 1244,

- Task_Dispatching_Policy [6823], page 981, [7647], page 1244,
- Unchecked_Union [6571], page 928, [7650], page 1244,
- Unsuppress [4184], page 431, [7653], page 1244,
- Volatile [6723], page 963, [7656], page 1244,
- Volatile_Components [6729], page 963, [7659], page 1244,
- precedence of operators [2571], page 203,
- Pred attribute [1577], page 80,
- predefined environment [4832], page 553,
- predefined exception [4097], page 419,
- predefined library unit
 - <See> language–defined library units
- predefined operation
 - of a type [1409], page 57,
- predefined operations
 - of a discrete type [1715], page 101,
 - of a fixed point type [1814], page 113,
 - of a floating point type [1756], page 106,
 - of a record type [1984], page 133,
 - of an access type [2237], page 174,
 - of an array type [1888], page 121,
- predefined operator [2579], page 203,
 - [<partial>] [1378], page 54,
- predefined type [1379], page 54,
 - <See> language–defined types
- prelaborable
 - of an elaborable construct [4059], page 414,
- prelaborable initialization [4064], page 415,
- Prelaborable_Initialization pragma [4057], page 414, [7600], page 1243,
- Preelaborate pragma [4052], page 413, [7603], page 1243,
- prelaborated [4062], page 415,
 - [<partial>] [4061], page 415, [7086], page 1039,
- preempt
 - a running task [6839], page 983,
- preference
 - for root numeric operators and ranges [3486], page 327,
- preference control

<See> requeue [3714], page 356,
prefix [2283], page 179,
of a prefixed view [2333], page 185,
<used> [2303], page 181, [2315], page 182, [2325], page 183, [2341], page 187, [2347],
page 187, [3142], page 267, [3146], page 267, [7987], page 1295,
prefixed view [2332], page 185,
prefixed view profile [3132], page 265,
Prepend
<in> Ada.Containers.Doubly_Linked_Lists [6098], page 812,
<in> Ada.Containers.Vectors [6030], page 784, [6031], page 784,
prescribed result
for the evaluation of a complex arithmetic operation [7233], page 1089,
for the evaluation of a complex elementary function [7260], page 1095,
for the evaluation of an elementary function [5445], page 653,
Previous
<in> Ada.Containers.Doubly_Linked_Lists [6114], page 814, [6116], page 814,
<in> Ada.Containers.Ordered_Maps [6227], page 849, [6228], page 849,
<in> Ada.Containers.Ordered_Sets [6361], page 880, [6362], page 880,
<in> Ada.Containers.Vectors [6051], page 786, [6052], page 786,
primary [2559], page 201,
<used> [2557], page 201, [8048], page 1297,
primitive function [5501], page 667,
primitive operation
[<partial>] [1315], page 50,
primitive operations [7715], page 1286,
of a type [1410], page 57,
primitive operator
of a type [1413], page 58,
primitive subprograms
of a type [1411], page 57,
priority [6802], page 976,
of a protected object [6868], page 991,
Priority attribute [6895], page 999,
priority inheritance [6803], page 976,
priority inversion [6840], page 983,
priority of an entry call [6887], page 995,

Priority pragma [6794], page 975, [7606], page 1243,
Priority <subtype of> Any_Priority
 <in> System [4627], page 512,
Priority_Specific_Dispatching pragma [6827], page 981, [7609], page 1243,
private declaration of a library unit [3955], page 396,
private descendant
 of a library unit [3957], page 396,
private extension [1331], page 51, [2017], page 137, [2063], page 143, [7716], page 1286,
 [<partial>] [3270], page 285, [4355], page 463,
private library unit [3954], page 396,
private operations [3274], page 287,
private part [3354], page 306,
 of a package [3240], page 279,
 of a protected unit [3595], page 339,
 of a task unit [3530], page 330,
private type [1330], page 51, [7717], page 1286,
 [<partial>] [3269], page 285,
private types and private extensions [3251], page 283,
private_extension_declaration [3259], page 283,
 <used> [1355], page 53, [7800], page 1290,
private_type_declaration [3256], page 283,
 <used> [1354], page 53, [7799], page 1290,
procedure [3012], page 255, [7718], page 1286,
 null [3222], page 277,
procedure instance [4289], page 456,
procedure_call_statement [3140], page 267,
 <used> [2913], page 240, [3843], page 381, [8318], page 1305,
procedure_or_entry_call [3842], page 381,
 <used> [3840], page 381, [3857], page 384, [8316], page 1305,
procedure_specification [3023], page 255,
 <used> [3021], page 255, [3220], page 277, [8126], page 1299,
processing node [7052], page 1034,
profile [3070], page 257,
 associated with a dereference [2293], page 180,
 fully conformant [3127], page 265,
 mode conformant [3121], page 264,

subtype conformant [3124], page 265,
type conformant [3118], page 264,
Profile pragma [6983], page 1020, [7614], page 1244,
profile resolution rule
 name with a given expected profile [3483], page 326,
progenitor [7719], page 1286,
progenitor subtype [2139], page 153,
progenitor type [2140], page 153,
program [4030], page 409, [7720], page 1286,
program execution [4031], page 409,
program library
 <See> library [3907], page 394,
 <See> library [4007], page 407,
Program unit [3910], page 394, [7721], page 1287,
program unit pragma [4008], page 407,
 Convention [6454], page 897,
 Export [6452], page 897,
 Import [6450], page 897,
 Inline [3133], page 266,
 library unit pragmas [4013], page 408,
Program_Error
 raised by failure of run-time check [1059], page 25, [1079], page 29, [1085], page 29,
 [1714], page 100, [2230], page 172, [2263], page 177, [2820], page 228, [2866], page 232,
 [2869], page 232, [2872], page 232, [3092], page 261, [3160], page 268, [3202], page 274,
 [3205], page 274, [3214], page 276, [3337], page 301, [3338], page 301, [3339], page 302,
 [3340], page 302, [3341], page 302, [3342], page 302, [3454], page 321, [3612], page 341,
 [3634], page 346, [3699], page 353, [3834], page 380, [3886], page 387, [4047], page 412,
 [4099], page 419, [4201], page 443, [4359], page 465, [4642], page 517, [4672], page 523,
 [4725], page 533, [4727], page 533, [5476], page 659, [5587], page 682, [6578], page 930,
 [6688], page 955, [6692], page 955, [6708], page 958, [6709], page 959, [6710], page 959,
 [6711], page 959, [6758], page 966, [6762], page 967, [6771], page 968, [6874], page 992,
 [6876], page 993, [6877], page 993, [6892], page 997, [6896], page 999, [6930], page 1007,
 [6971], page 1016, [6980], page 1017, [7068], page 1036, [7125], page 1044, [7142], page 1046,
 [7441], page 1172,
 <in> Standard [4852], page 563,
propagate [4131], page 422,

an exception occurrence by an execution, to a dynamically enclosing execution [4136],
page 423,
proper_body [2253], page 176,
 <used> [2251], page 176, [3996], page 404, [7956], page 1295,
protected action [3626], page 345,
 complete [3629], page 345,
 start [3627], page 345,
protected calling convention [3113], page 264,
protected declaration [3561], page 337,
protected entry [3560], page 337,
protected function [3624], page 344,
protected interface [2128], page 152,
protected object [3496], page 328, [3557], page 337,
protected operation [3558], page 337,
protected procedure [3623], page 344,
protected subprogram [3559], page 337, [3622], page 344,
protected tagged type [2138], page 152,
protected type [7722], page 1287,
protected unit [3562], page 337,
protected_body [3585], page 338,
 <used> [2257], page 176, [7961], page 1295,
protected_body_stub [3992], page 403,
 <used> [3984], page 403, [8361], page 1306,
protected_definition [3574], page 338,
 <used> [3569], page 338, [3573], page 338, [8253], page 1303,
protected_element_declaration [3582], page 338,
 <used> [3576], page 338, [8255], page 1303,
protected_operation_declaration [3578], page 338,
 <used> [3575], page 338, [3583], page 338, [8254], page 1303,
protected_operation_item [3589], page 338,
 <used> [3587], page 338, [8263], page 1303,
protected_type_declaration [3565], page 338,
 <used> [1361], page 53, [7805], page 1290,
ptrdiff_t
 <in> Interfaces.C [6484], page 903,

PU1

<in> Ada.Characters.Latin_1 [5010], page 576,

PU2

<in> Ada.Characters.Latin_1 [5011], page 576,
public declaration of a library unit [3953], page 396,
public descendant

of a library unit [3956], page 396,
public library unit [3952], page 396,
punctuation_connector [1134], page 34,

<used> [1196], page 38, [7752], page 1289,
pure [4070], page 416,

Pure pragma [4066], page 416, [7618], page 1244,

Put

<in> Ada.Text_IO [5740], page 702, [5749], page 703, [5763], page 704, [5771], page 704,
[5781], page 705, [5784], page 706, [5792], page 706, [5794], page 706, [5802], page 707,
[5804], page 707, [5810], page 708, [5813], page 708,

<in> Ada.Text_IO.Bounded_IO [5831], page 739, [5832], page 739,

<in> Ada.Text_IO.Complex_IO [7270], page 1098, [7272], page 1098,

<in> Ada.Text_IO.Editing [7199], page 1074, [7200], page 1074, [7201], page 1075,

<in> Ada.Text_IO.Unbounded_IO [5840], page 742, [5841], page 742,

Put_Line

<in> Ada.Text_IO [5756], page 703,

<in> Ada.Text_IO.Bounded_IO [5833], page 740, [5834], page 740,

<in> Ada.Text_IO.Unbounded_IO [5842], page 742, [5843], page 742,

30.18 Q

qualified_expression [2826], page 229,

<used> [2564], page 201, [2846], page 231, [4649], page 518, [8056], page 1297,

Query_Element

<in> Ada.Containers.Doubly_Linked_Lists [6092], page 812,

<in> Ada.Containers.Hashed_Maps [6163], page 841,

<in> Ada.Containers.Hashed_Sets [6272], page 868,

<in> Ada.Containers.Ordered_Maps [6206], page 847,

<in> Ada.Containers.Ordered_Sets [6334], page 877,
<in> Ada.Containers.Vectors [6017], page 782, [6018], page 782,

Question

<in> Ada.Characters.Latin_1 [4949], page 575,
queuing policy [6878], page 994, [6886], page 995,
Queuing_Policy pragma [6880], page 994, [7621], page 1244,

Quotation

<in> Ada.Characters.Latin_1 [4929], page 574,
quotation mark [1145], page 36,
quoted string
<See> string_literal [1244], page 42,

30.19 R

raise

an exception [4092], page 419,
an exception [4127], page 421,
an exception [7696], page 1284,
an exception occurrence [4135], page 422,

Raise_Exception

<in> Ada.Exceptions [4149], page 424,
raise_statement [4123], page 421,
<used> [2919], page 240, [8085], page 1298,

Random

<in> Ada.Numerics.Discrete_Random [5461], page 656,
<in> Ada.Numerics.Float_Random [5450], page 655,
random number [5446], page 654,
range [1533], page 76, [1537], page 76,
of a scalar subtype [1550], page 77,
<used> [1532], page 76, [1832], page 114, [1863], page 117, [2542], page 201, [8038],
page 1297,

Range attribute [1561], page 78, [1881], page 120,

Range(N) attribute [1883], page 120,

range_attribute_designator [2349], page 187,
 <used> [2348], page 187, [7992], page 1295,
range_attribute_reference [2346], page 187,
 <used> [1534], page 76, [7846], page 1291,
Range_Check [4199], page 439,
 [<partial>] [1404], page 56, [1574], page 79, [1581], page 80, [1608], page 88, [1602],
page 86, [1605], page 87, [1614], page 89, [1620], page 90, [1712], page 100, [1793], page 108,
[2366], page 189, [2446], page 199, [2604], page 205, [2711], page 218, [2720], page 219,
[2772], page 223, [2784], page 225, [2797], page 226, [2803], page 227, [2833], page 230,
[4787], page 546, [5473], page 658, [5509], page 668, [5514], page 669, [5537], page 673,
[5542], page 674, [5547], page 675, [5552], page 676, [7460], page 1181, [7473], page 1202,
[7476], page 1204, [7482], page 1219, [7489], page 1226, [7492], page 1231, [7465], page 1188,
[7468], page 1189,
range_constraint [1531], page 76,
 <used> [1395], page 56, [1773], page 107, [7425], page 1168, [7822], page 1291,
Ravenscar [6988], page 1020,
RCI
 generic [7112], page 1042,
 library unit [7110], page 1042,
 package [7111], page 1042,
Re
 <in> Ada.Numerics.Generic_Complex_Arrays [7338], page 1130, [7351], page 1133,
 <in> Ada.Numerics.Generic_Complex_Types [7212], page 1084,
re-raise statement [4126], page 421,
read
 the value of an object [1422], page 59,
 <in> Ada.Direct_IO [5634], page 692,
 <in> Ada.Sequential_IO [5609], page 684,
 <in> Ada.Storage_IO [5652], page 695,
 <in> Ada.Streams [4766], page 539,
 <in> Ada.Streams.Stream_IO [5879], page 747, [5880], page 747,
 <in> System.RPC [7159], page 1051,
Read attribute [4774], page 541, [4778], page 543,
Read clause [4486], page 487, [4796], page 547,
ready
 a task state [3503], page 328,

ready queue [6815], page 979,

ready task [6818], page 979,

Real

<in> Interfaces.Fortran [6637], page 945,

real literal [1200], page 39,

real literals [1722], page 102,

real time [6953], page 1009,

real type [1328], page 51, [1716], page 102, [7723], page 1287,

real-time systems [6655], page 949, [6791], page 974,

Real_Arrays

<child of> Ada.Numerics [7331], page 1122,

Real_Matrix

<in> Ada.Numerics.Generic_Real_Arrays [7321], page 1119,

real_range_specification [1730], page 103,

<used> [1729], page 103, [1766], page 106, [1770], page 107, [7862], page 1292,

Real_Time

<child of> Ada [6931], page 1008,

real_type_definition [1717], page 102,

<used> [1365], page 54, [7808], page 1290,

Real_Vector

<in> Ada.Numerics.Generic_Real_Arrays [7320], page 1119,

receiving stub [7135], page 1045,

reclamation of storage [4717], page 532,

recommended level of support [4448], page 484,

Address attribute [4500], page 488,

Alignment attribute for objects [4512], page 491,

Alignment attribute for subtypes [4511], page 491,

bit ordering [4601], page 509,

Component_Size attribute [4538], page 497,

enumeration_representation_clause [4552], page 501,

pragma Pack [4458], page 486,

record_representation_clause [4579], page 504,

required in Systems Programming Annex [6664], page 950,

Size attribute [4517], page 493, [4522], page 495,

Stream_Size attribute [4770], page 541,

unchecked conversion [4661], page 521,

with respect to nonstatic expressions [4449], page 484,
record [1949], page 130,
 explicitly limited [1972], page 131,
record extension [1491], page 67, [2061], page 143, [7724], page 1287,
record layout
 aspect of representation [4556], page 501,
record type [1950], page 130, [7725], page 1287,
record_aggregate [2381], page 191,
 <used> [2372], page 190, [7994], page 1296,
record_component_association [2386], page 191,
 <used> [2385], page 191, [7998], page 1296,
record_component_association_list [2383], page 191,
 <used> [2382], page 191, [2404], page 194, [8006], page 1296,
record_definition [1954], page 130,
 <used> [1953], page 130, [2066], page 143, [7908], page 1293,
record_extension_part [2065], page 143,
 <used> [1488], page 66, [7844], page 1291,
record_representation_clause [4560], page 502,
 <used> [4420], page 481, [8460], page 1310,
record_type_definition [1952], page 130,
 <used> [1367], page 54, [7810], page 1290,
reentrant [4833], page 556,

Reference

 <in> Ada.Interrupts [6706], page 958,
 <in> Ada.Task_Attributes [6767], page 968,
reference parameter passing [3083], page 260,
references [1086], page 30,

Registered_Trade_Mark_Sign

 <in> Ada.Characters.Latin_1 [5040], page 577,

Reinitialize

 <in> Ada.Task_Attributes [6769], page 968,
relation [2537], page 201,
 <used> [2533], page 201, [8029], page 1296,
relational operator [2607], page 206,
relational_operator [2574], page 203,
 <used> [2539], page 201, [8035], page 1297,

Relative_Deadline pragma [6850], page 988, [7624], page 1244,
relaxed mode [7280], page 1103,
release
 execution resource associated with protected object [3630], page 345,
rem operator [2515], page 201, [2697], page 213,
Remainder attribute [5530], page 672,
remote access [7059], page 1035,
remote access type [7096], page 1040,
remote access-to-class-wide type [7098], page 1040,
remote access-to-subprogram type [7097], page 1040,
remote call interface [7076], page 1037, [7109], page 1042,
remote procedure call
 asynchronous [7151], page 1047,
remote subprogram [7113], page 1042,
remote subprogram binding [7130], page 1044,
remote subprogram call [7126], page 1044,
remote types library unit [7075], page 1037, [7095], page 1039,
Remote_Call_Interface pragma [7102], page 1042, [7627], page 1244,
Remote_Types pragma [7093], page 1039, [7630], page 1244,
Remove_Task
 <in> Ada.Execution_Time.Group_Budgets [7023], page 1028,
Rename
 <in> Ada.Directories [5935], page 758,
renamed entity [3424], page 316,
renamed view [3423], page 316,
renaming [7726], page 1287,
renaming-as-body [3442], page 319,
renaming-as-declaration [3443], page 319,
renaming_declaration [3416], page 316,
 <used> [1291], page 49, [7792], page 1290,
rendezvous [3683], page 351,
Replace
 <in> Ada.Containers.Hashing_Maps [6170], page 842,
 <in> Ada.Containers.Hashing_Sets [6277], page 869, [6304], page 872,
 <in> Ada.Containers.Ordering_Maps [6213], page 848,
 <in> Ada.Containers.Ordering_Sets [6339], page 878, [6374], page 882,

Replace_Element

- <in> Ada.Containers.Doubly_Linked_Lists [6091], page 811,
- <in> Ada.Containers.Hashed_Maps [6162], page 841,
- <in> Ada.Containers.Hashed_Sets [6271], page 868,
- <in> Ada.Containers.Ordered_Maps [6205], page 847,
- <in> Ada.Containers.Ordered_Sets [6333], page 877,
- <in> Ada.Containers.Vectors [6015], page 782, [6016], page 782,
- <in> Ada.Strings.Bounded [5235], page 613,
- <in> Ada.Strings.Unbounded [5288], page 627,

Replace_Slice

- <in> Ada.Strings.Bounded [5255], page 618, [5256], page 618,
- <in> Ada.Strings.Fixed [5198], page 595, [5199], page 595,
- <in> Ada.Strings.Unbounded [5308], page 631, [5309], page 631,

Replenish

- <in> Ada.Execution_Time.Group_Budgets [7027], page 1028,

Replicate

- <in> Ada.Strings.Bounded [5271], page 620, [5272], page 621, [5273], page 621,

representation

- change of [4603], page 509,
- representation aspect [4432], page 482,
- representation attribute [4459], page 486,
- representation item [4413], page 481,
- representation of an object [4428], page 482,
- representation pragma [4414], page 481,
 - Asynchronous [7149], page 1047,
 - Atomic [6735], page 964,
 - Atomic_Components [6739], page 964,
 - Controlled [4734], page 534,
 - Convention [6442], page 897,
 - Discard_Names [6718], page 962,
 - Export [6440], page 897,
 - Import [6438], page 897,
 - Pack [4453], page 486,
 - Volatile [6737], page 964,
 - Volatile_Components [6741], page 964,
- representation-oriented attributes

of a fixed point subtype [5569], page 679,
of a floating point subtype [5478], page 663,
representation_clause
 <See> aspect_clause [4427], page 481,
represented in canonical form [5492], page 665,
requested decimal precision
 of a floating point type [1733], page 103,
requeue [3713], page 356,
requeue-with-abort [3725], page 358,
requeue_statement [3716], page 356,
 <used> [2916], page 240, [8082], page 1298,
require overriding [2113], page 150,
requires a completion [2264], page 177, [2268], page 177,
 declaration of a partial view [3265], page 284,
 declaration to which a pragma Elaborate_Body applies [4085], page 418,
 deferred constant declaration [3282], page 290,
 generic_package_declaration [3238], page 279,
 generic_subprogram_declaration [3069], page 257,
 incomplete_type_declaration [2202], page 161,
 package_declaration [3237], page 279,
 protected entry_declaration [3667], page 349,
 protected_declaration} [3599], page 339,
 subprogram_declaration [3068], page 257,
 task_declaration} [3533], page 330,
requires late initialization [1457], page 62,
Reraise_Occurrence
 <in> Ada.Exceptions [4151], page 424,
Reserve_Capacity
 <in> Ada.Containers.Hashing_Maps [6156], page 840,
 <in> Ada.Containers.Hashing_Sets [6266], page 868,
 <in> Ada.Containers.Vectors [6006], page 781,
reserved interrupt [6673], page 951,
reserved word [1279], page 47,
Reserved_128
 <in> Ada.Characters.Latin_1 [4993], page 576,
Reserved_129

<in> Ada.Characters.Latin_1 [4994], page 576,
Reserved_132

<in> Ada.Characters.Latin_1 [4997], page 576,
Reserved_153

<in> Ada.Characters.Latin_1 [5018], page 577,
Reserved_Check

[<partial>] [6686], page 955,

Reset

<in> Ada.Direct_IO [5629], page 691,

<in> Ada.Numerics.Discrete_Random [5462], page 656, [5466], page 656,

<in> Ada.Numerics.Float_Random [5452], page 655, [5455], page 655,

<in> Ada.Sequential_IO [5604], page 684,

<in> Ada.Streams.Stream_IO [5872], page 747,

<in> Ada.Text_IO [5681], page 699,

resolution rules [1021], page 22,

resolve

overload resolution [3471], page 325,

restriction [4743], page 535,

<used> [4741], page 535, [7635], page 1244,

restriction_parameter_argument [4747], page 535,

<used> [4746], page 535, [8488], page 1310,

Restrictions

Immediate_Reclamation [7398], page 1159,

Max_Asynchronous_Select_Nesting [6921], page 1006,

Max_Entry_Queue_Length [6929], page 1007,

Max_Protected_Entries [6915], page 1005,

Max_Select_Alternatives [6913], page 1004,

Max_Storage_At_Blocking [6917], page 1006,

Max_Task_Entries [6914], page 1004,

Max_Tasks [6925], page 1006,

No_Abort_Statements [6899], page 1002,

No_Access_Subprograms [7402], page 1160,

No_Allocators [7396], page 1159,

No_Asynchronous_Control [7455], page 1176,

No_Delay [7408], page 1161,

No_Dependence [4756], page 537,

No_Dispatch [7406], page 1161,
No_Dynamic_Attachment [6904], page 1003,
No_Dynamic_Priorities [6903], page 1003,
No_Exceptions [7399], page 1160,
No_Fixed_Point [7401], page 1160,
No_Floating_Point [7400], page 1160,
No_Implementation_Attributes [4753], page 536,
No_Implementation_Pragmas [4754], page 537,
No_Implicit_Heap_Allocations [6902], page 1002,
No_IO [7407], page 1161,
No_Local_Allocators [7397], page 1159,
No_Local_Protected_Objects [6905], page 1003,
No_Local_Timing_Events [6906], page 1003,
No_Nested_Finalization [6898], page 1002,
No_Obsolescent_Features [4755], page 537,
No_Protected_Type_Allocators [6907], page 1003,
No_Protected_Types [7395], page 1159,
No_Recursion [7409], page 1161,
No_Reentrancy [7410], page 1161,
No_Relative_Delay [6908], page 1004,
No_Requeue_Statements [6909], page 1004,
No_Select_Statements [6910], page 1004,
No_Specific_Termination_Handlers [6911], page 1004,
No_Task_Allocators [6901], page 1002,
No_Task_Hierarchy [6897], page 1002,
No_Task_Termination [6916], page 1005,
No_Terminate_Alternatives [6900], page 1002,
No_Unchecked_Access [7403], page 1160,
No_Unchecked_Conversion [7456], page 1177,
No_Unchecked_Deallocation [7457], page 1177,
Simple_Barriers [6912], page 1004,
Restrictions pragma [4740], page 535, [7633], page 1244,
result interval
for a component of the result of evaluating a complex function [7313], page 1116,
for the evaluation of a predefined arithmetic operation [7285], page 1104,
for the evaluation of an elementary function [7307], page 1113,

result subtype

of a function [3190], page 272,

return object

extended_return_statement [3195], page 273,

simple_return_statement [3200], page 274,

return statement [3179], page 272,

return_subtype_indication [3187], page 272,

<used> [3184], page 272, [8176], page 1301,

Reverse_Elements

<in> Ada.Containers.Doubly_Linked_Lists [6103], page 813,

<in> Ada.Containers.Vectors [6040], page 785,

Reverse_Find

<in> Ada.Containers.Doubly_Linked_Lists [6118], page 814,

<in> Ada.Containers.Vectors [6056], page 786,

Reverse_Find_Index

<in> Ada.Containers.Vectors [6055], page 786,

Reverse_Iterate

<in> Ada.Containers.Doubly_Linked_Lists [6122], page 815,

<in> Ada.Containers.Ordered_Maps [6236], page 851,

<in> Ada.Containers.Ordered_Sets [6369], page 881,

<in> Ada.Containers.Vectors [6060], page 787,

Reverse_Solidus

<in> Ada.Characters.Latin_1 [4952], page 575,

Reviewable pragma [7386], page 1155, [7637], page 1244,

RI

<in> Ada.Characters.Latin_1 [5006], page 576,

right parenthesis [1151], page 36,

Right_Angle_Quotation

<in> Ada.Characters.Latin_1 [5055], page 578,

Right_Curly_Bracket

<in> Ada.Characters.Latin_1 [4985], page 576,

Right_Parenthesis

<in> Ada.Characters.Latin_1 [4936], page 574,

Right_Square_Bracket

<in> Ada.Characters.Latin_1 [4953], page 575,

Ring_Above

<in> Ada.Characters.Latin_1 [5043], page 577,
root library unit [3948], page 396,
root type
 of a class [1513], page 72,
root_integer [1679], page 96,
 <partial> [1519], page 75,
root_real [1720], page 102,
 <partial> [1520], page 75,
Root_Storage_Pool
 <in> System.Storage_Pools [4686], page 526,
Root_Stream_Type
 <in> Ada.Streams [4761], page 538,
rooted at a type [1514], page 72,
rotate [6467], page 901,
Round attribute [1813], page 113,
Round_Robin
 <child of> Ada.Dispatching [6843], page 985,
Rounding attribute [5521], page 670,
RPC
 <child of> System [7155], page 1050,
RPC–receiver [7166], page 1052,
RPC_Receiver
 <in> System.RPC [7163], page 1051,
RS
 <in> Ada.Characters.Latin_1 [4925], page 574,
run–time check
 <See> language–defined check [4177], page 431,
run–time error [1037], page 22, [1076], page 28, [4178], page 431, [4214], page 448,
run–time polymorphism [2076], page 145,
run–time semantics [1036], page 22,
run–time type
 <See> tag [2022], page 137,
running a program
 <See> program execution [4032], page 409,
running task [6821], page 979,

30.20 S

safe range

of a floating point type [1741], page 104,

of a floating point type [1743], page 104,

Safe_First attribute [5566], page 678, [7294], page 1107,

Safe_Last attribute [5568], page 679, [7296], page 1107,

safety-critical systems [7375], page 1153,

satisfies

a discriminant constraint [1943], page 128,

a range constraint [1543], page 76,

an index constraint [1869], page 118,

for an access value [2192], page 158,

Saturday

<in> Ada.Calendar.Formatting [3780], page 365,

Save

<in> Ada.Numerics.Discrete_Random [5465], page 656,

<in> Ada.Numerics.Float_Random [5454], page 655,

Save_Occurrence

<in> Ada.Exceptions [4158], page 424,

scalar type [1325], page 51, [1527], page 76, [7727], page 1287,

scalar_constraint [1394], page 56,

<used> [1392], page 56, [7820], page 1291,

scale

of a decimal fixed point subtype [1811], page 113, [7487], page 1225,

Scale attribute [1810], page 112,

Scaling attribute [5512], page 668,

SCHAR_MAX

<in> Interfaces.C [6473], page 902,

SCHAR_MIN

<in> Interfaces.C [6472], page 902,

SCI

<in> Ada.Characters.Latin_1 [5019], page 577,
scope
informal definition [1303], page 50,
of (a view of) an entity [3361], page 307,
of a declaration [3358], page 307,
of a use_clause [3410], page 314,
of a with_clause [3975], page 400,
of an attribute_definition_clause [3359], page 307,
Search_Type
<in> Ada.Directories [5951], page 760,
Second
<in> Ada.Calendar.Formatting [3792], page 367,
Second_Duration <subtype of> Day_Duration
<in> Ada.Calendar.Formatting [3786], page 366,
Second_Number <subtype of> Natural
<in> Ada.Calendar.Formatting [3785], page 366,
Seconds
<in> Ada.Calendar [3751], page 360,
<in> Ada.Real_Time [6948], page 1009,
Seconds_Count
<in> Ada.Real_Time [6950], page 1009,
Seconds_Of
<in> Ada.Calendar.Formatting [3794], page 367,
Section_Sign
<in> Ada.Characters.Latin_1 [5033], page 577,
secure systems [7376], page 1153,
select an entry call
from an entry queue [3705], page 354, [3706], page 354,
immediately [3702], page 353,
select_alternative [3819], page 378,
<used> [3813], page 378, [8304], page 1304,
select_statement [3805], page 377,
<used> [2928], page 241, [8093], page 1298,
selected_component [2324], page 183,
<used> [2275], page 179, [7966], page 1295,
selection

of an entry caller [3682], page 350,
selective_accept [3811], page 378,
 <used> [3806], page 377, [8297], page 1304,
selector_name [2327], page 184,
 <used> [1933], page 127, [2326], page 183, [2391], page 191, [3152], page 267, [4273],
page 455, [4409], page 475, [8457], page 1310,
semantic dependence
 of one compilation unit upon another [3959], page 398,
semicolon [1164], page 36,
 <in> Ada.Characters.Latin_1 [4945], page 575,
separate compilation [3908], page 394,
separator [1178], page 37,
separator_line [1137], page 34,
separator_paragraph [1138], page 34,
separator_space [1136], page 34,
sequence of characters
 of a string_literal [1249], page 42,
sequence_of_statements [2900], page 240,
 <used> [2957], page 245, [2971], page 246, [2981], page 248, [3816], page 378, [3825],
page 379, [3828], page 379, [3841], page 381, [3848], page 383, [3855], page 384, [3860],
page 384, [4108], page 420, [4115], page 420, [8102], page 1298,
sequential
 actions [3903], page 391, [6743], page 964,
sequential access [5591], page 683,
sequential file [5588], page 682,
Sequential_IO
 <child of> Ada [5596], page 683,
service
 an entry queue [3704], page 354,
set
 execution timer object [7012], page 1025,
 group budget object [7038], page 1029,
 termination handler [6789], page 972,
 timing event object [7049], page 1032,
 <in> Ada.Containers.Hash_Sets [6259], page 867,
 <in> Ada.Containers.Ordered_Sets [6323], page 876,

<in> Ada.Environment_Variables [5979], page 775,
set container [6243], page 855,
Set_Bounded_String
<in> Ada.Strings.Bounded [5225], page 612,
Set_Col
<in> Ada.Text_IO [5728], page 701,
Set_Deadline
<in> Ada.Dispatching.EDF [6855], page 988,
Set_Dependents_Fallback_Handler
<in> Ada.Task_Termination [6779], page 972,
Set_Directory
<in> Ada.Directories [5929], page 758,
Set_Error
<in> Ada.Text_IO [5689], page 699,
Set_Exit_Status
<in> Ada.Command_Line [5925], page 755,
Set_False
<in> Ada.Synchronous_Task_Control [6966], page 1015,
Set_Handler
<in> Ada.Execution_Time.Group_Budgets [7031], page 1028,
<in> Ada.Execution_Time.Timers [7006], page 1025,
<in> Ada.Real_Time.Timing_Events [7045], page 1031,
Set_Im
<in> Ada.Numerics.Generic_Complex_Arrays [7341], page 1131, [7354], page 1133,
<in> Ada.Numerics.Generic_Complex_Types [7216], page 1085,
Set_Index
<in> Ada.Direct_IO [5638], page 692,
<in> Ada.Streams.Stream_IO [5883], page 748,
Set_Input
<in> Ada.Text_IO [5687], page 699,
Set_Length
<in> Ada.Containers.Vectors [6008], page 781,
Set_Line
<in> Ada.Text_IO [5730], page 701,
Set_Line_Length
<in> Ada.Text_IO [5705], page 700,

Set_Mode

<in> Ada.Streams.Stream_IO [5886], page 748,

Set_Output

<in> Ada.Text_IO [5688], page 699,

Set_Page_Length

<in> Ada.Text_IO [5708], page 700,

Set_Priority

<in> Ada.Dynamic_Priorities [6889], page 996,

Set_Quantum

<in> Ada.Dispatching.Round_Robin [6846], page 986,

Set_Re

<in> Ada.Numerics.Generic_Complex_Arrays [7340], page 1131, [7353], page 1133,

<in> Ada.Numerics.Generic_Complex_Types [7215], page 1085,

Set_Specific_Handler

<in> Ada.Task_Termination [6781], page 972,

Set_True

<in> Ada.Synchronous_Task_Control [6965], page 1015,

Set_Unbounded_String

<in> Ada.Strings.Unbounded [5283], page 626,

Set_Value

<in> Ada.Task_Attributes [6768], page 968,

shared passive library unit [7074], page 1037, [7083], page 1038,

shared variable

protection of [3899], page 389,

Shared_Passive pragma [7081], page 1038, [7639], page 1244,

shift [6466], page 901,

short

<in> Interfaces.C [6476], page 902,

short-circuit control form [2588], page 204,

Short_Float [1751], page 104,

Short_Integer [1699], page 97,

SI

<in> Ada.Characters.Latin_1 [4910], page 573,

signal

as defined between actions [3901], page 390,

<See> interrupt [6665], page 951,

signal (an exception)

<See> raise [4088], page 419,

signal handling

example [3864], page 385,

signed integer type [1654], page 95,

signed_char

<in> Interfaces.C [6478], page 902,

signed_integer_type_definition [1660], page 95,

<used> [1658], page 95, [7854], page 1291,

Signed_Zeros attribute [5499], page 666,

simple entry call [3686], page 352,

simple name

of a file [5971], page 761,

simple_expression [2545], page 201,

<used> [1535], page 76, [1662], page 95, [1732], page 103, [2541], page 201, [4572],
page 502, [4574], page 502, [7857], page 1292,

Simple_Name

<in> Ada.Directories [5938], page 758, [5956], page 760,

simple_return_statement [3180], page 272,

<used> [2914], page 240, [8080], page 1298,

simple_statement [2908], page 240,

<used> [2905], page 240, [8072], page 1297,

Sin

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7239], page 1092,

<in> Ada.Numerics.Generic_Elementary_Functions [5416], page 649,

single

class expected type [3485], page 327,

single entry [3675], page 350,

Single_Precision_Complex_Types

<in> Interfaces.Fortran [6640], page 945,

single_protected_declaration [3570], page 338,

<used> [1447], page 61, [7837], page 1291,

single_task_declaration [3513], page 329,

<used> [1446], page 61, [7836], page 1291,

Sinh

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7247], page 1092,

<in> Ada.Numerics.Generic_Elementary_Functions [5431], page 650,
size
of an object [4429], page 482,
<in> Ada.Direct_IO [5640], page 692,
<in> Ada.Directories [5947], page 759, [5959], page 761,
<in> Ada.Streams.Stream_IO [5885], page 748,
Size attribute [4514], page 492, [4519], page 493,
Size clause [4478], page 487, [4516], page 492, [4521], page 494,
size_t
<in> Interfaces.C [6485], page 903,
Skip_Line
<in> Ada.Text_IO [5715], page 701,
Skip_Page
<in> Ada.Text_IO [5722], page 701,
slice [2314], page 182,
<used> [2274], page 179, [7965], page 1295,
<in> Ada.Strings.Bounded [5236], page 613,
<in> Ada.Strings.Unbounded [5289], page 627,
small
of a fixed point type [1778], page 107,
Small attribute [1797], page 109,
Small clause [1799], page 110, [4481], page 487,
SO
<in> Ada.Characters.Latin_1 [4909], page 573,
Soft_Hyphen
<in> Ada.Characters.Latin_1 [5039], page 577,
SOH
<in> Ada.Characters.Latin_1 [4896], page 573,
solidus [1161], page 36,
<in> Ada.Characters.Latin_1 [4943], page 574,
Solve
<in> Ada.Numerics.Generic_Complex_Arrays [7364], page 1135,
<in> Ada.Numerics.Generic_Real_Arrays [7325], page 1121,
Sort
<in> Ada.Containers.Doubly_Linked_Lists [6125], page 815,
<in> Ada.Containers.Vectors [6063], page 787,

SOS

<in> Ada.Characters.Latin_1 [5017], page 577,

SPA

<in> Ada.Characters.Latin_1 [5015], page 576,

Space

<in> Ada.Characters.Latin_1 [4927], page 574,

<in> Ada.Strings [5144], page 584,

special file [5966], page 761,

special graphic character

a category of Character [4892], page 570,

Special_Set

<in> Ada.Strings.Maps.Constants [5334], page 636,

Specialized Needs Annexes [1006], page 20,

specifiable

of Address for entries [7439], page 1172,

of Address for stand-alone objects and for program units [4497], page 488,

of Alignment for first subtypes [4507], page 490,

of Alignment for objects [4503], page 490,

of Bit_Order for record types and record extensions [4598], page 508,

of Component_Size for array types [4536], page 497,

of External_Tag for a tagged type [4542], page 498, [7471], page 1192,

of Input for a type [4794], page 547,

of Machine_Radix for decimal first subtypes [7170], page 1054,

of Output for a type [4795], page 547,

of Read for a type [4792], page 547,

of Size for first subtypes [4520], page 494,

of Size for stand-alone objects [4515], page 492,

of Small for fixed point types [1798], page 110,

of Storage_Pool for a non-derived access-to-object type [4700], page 528,

of Storage_Size for a non-derived access-to-object type [4699], page 528,

of Storage_Size for a task first subtype [7448], page 1174,

of Write for a type [4793], page 547,

specifiable (of an attribute and for an entity) [4471], page 487,

specific handler [6787], page 972,

specific type [1515], page 72,

Specific_Handler

<in> Ada.Task_Termination [6782], page 972,
specified

of an aspect of representation of an entity [4443], page 484,
of an operational aspect of an entity [4445], page 484,
specified (not!) [1056], page 25,
specified discriminant [1916], page 125,

Splice

<in> Ada.Containers.Doubly_Linked_Lists [6106], page 813, [6107], page 813, [6108],
page 813,

Split

<in> Ada.Calendar [3752], page 360,
<in> Ada.Calendar.Formatting [3795], page 367, [3798], page 368, [3799], page 368,
[3800], page 368,
<in> Ada.Execution_Time [6996], page 1022,
<in> Ada.Real_Time [6951], page 1009,

Sqrt

<in> Ada.Numerics.Generic_Complex_Elementary_Functions [7235], page 1092,
<in> Ada.Numerics.Generic_Elementary_Functions [5411], page 649,

SS2

<in> Ada.Characters.Latin_1 [5007], page 576,

SS3

<in> Ada.Characters.Latin_1 [5008], page 576,

SSA

<in> Ada.Characters.Latin_1 [4999], page 576,

ST

<in> Ada.Characters.Latin_1 [5021], page 577,
stand-alone constant [1473], page 64,
corresponding to a formal object of mode in [4319], page 460,
stand-alone object [1430], page 61,
[<partial>] [4320], page 460,
stand-alone variable [1474], page 64,
Standard [4836], page 557,
standard error file [5657], page 697,
standard input file [5655], page 697,
standard mode [1084], page 29,
standard output file [5656], page 697,

standard storage pool [4705], page 528,
Standard_Error
 <in> Ada.Text_IO [5692], page 699, [5699], page 700,
Standard_Input
 <in> Ada.Text_IO [5690], page 699, [5697], page 700,
Standard_Output
 <in> Ada.Text_IO [5691], page 699, [5698], page 700,
Start_Search
 <in> Ada.Directories [5952], page 760,
State
 <in> Ada.Numerics.Discrete_Random [5464], page 656,
 <in> Ada.Numerics.Float_Random [5453], page 655,
statement [2903], page 240,
 <used> [2902], page 240, [8069], page 1297,
statement_identifier [2932], page 241,
 <used> [2931], page 241, [2979], page 248, [2997], page 251, [8109], page 1298,
static [2875], page 234,
 constant [2880], page 236,
 constraint [2886], page 236,
 delta constraint [2889], page 236,
 digits constraint [2888], page 236,
 discrete_range [2882], page 236,
 discriminant constraint [2891], page 236,
 expression [2877], page 234,
 function [2879], page 235,
 index constraint [2890], page 236,
 range [2881], page 236,
 range constraint [2887], page 236,
 scalar subtype [2884], page 236,
 string subtype [2885], page 236,
 subtype [2883], page 236,
 subtype [4317], page 460,
static semantics [1027], page 22,
statically
 constrained [2892], page 237,
 denote [2878], page 235,

statically compatible

for a constraint and a scalar subtype [2897], page 239,

for a constraint and an access or composite subtype [2898], page 239,

for two subtypes [2899], page 239,

statically deeper [2215], page 164, [2218], page 168,

statically determined tag [2073], page 145,

[<partial>] [2095], page 148, [2099], page 148,

statically matching

effect on subtype-specific aspects [4442], page 483,

for constraints [2894], page 238,

for ranges [2896], page 239,

for subtypes [2895], page 239,

required [2091], page 147, [2225], page 171, [2758], page 222, [2753], page 220, [3122], page 264, [3125], page 265, [3130], page 265, [3194], page 273, [3268], page 285, [3433], page 317, [4313], page 459, [4356], page 464, [4368], page 467, [4369], page 467, [4372], page 469, [4411], page 476,

statically tagged [2088], page 146,

Status_Error

<in> Ada.Direct_IO [5642], page 692,

<in> Ada.Directories [5961], page 761,

<in> Ada.IO_Exceptions [5906], page 752,

<in> Ada.Sequential_IO [5612], page 684,

<in> Ada.Streams.Stream_IO [5888], page 748,

<in> Ada.Text_IO [5814], page 708,

storage deallocation

unchecked [4715], page 532,

storage element [4491], page 487,

storage management

user-defined [4682], page 526,

storage node [7053], page 1034,

storage place

of a component [4558], page 501,

storage place attributes

of a component [4580], page 506,

storage pool [2171], page 157,

storage pool element [4692], page 527,

storage pool type [4690], page 527,
Storage_Array
 <in> System.Storage_Elements [4635], page 516,
Storage_Check [4205], page 446,
 [<partial>] [4103], page 419, [4531], page 497, [4706], page 528, [6918], page 1006, [6922],
page 1006, [6926], page 1007,
Storage_Count <subtype of> Storage_Offset
 <in> System.Storage_Elements [4633], page 516,
Storage_Element
 <in> System.Storage_Elements [4634], page 516,
Storage_Elements
 <child of> System [4631], page 516,
Storage_Error
 raised by failure of run–time check [2874], page 233, [3455], page 321, [4100], page 419,
[4105], page 419, [4206], page 446, [4533], page 497, [4708], page 528, [4709], page 528,
[5586], page 682, [6920], page 1006, [6924], page 1006, [6928], page 1007,
 <in> Standard [4853], page 563,
Storage_IO
 <child of> Ada [5649], page 695,
Storage_Offset
 <in> System.Storage_Elements [4632], page 516,
Storage_Pool attribute [4696], page 527,
Storage_Pool clause [4483], page 487, [4701], page 528,
Storage_Pools
 <child of> System [4685], page 526,
Storage_Size
 <in> System.Storage_Pools [4689], page 527,
Storage_Size attribute [4524], page 496, [4698], page 527, [7447], page 1174,
Storage_Size clause [4484], page 487, [4702], page 528,
 <See also> pragma Storage_Size [4525], page 496,
Storage_Size pragma [4527], page 496, [7642], page 1244,
Storage_Unit
 <in> System [4618], page 511,
stream [4757], page 538,
 <in> Ada.Streams.Stream_IO [5878], page 747,
 <in> Ada.Text_IO.Text_Streams [5898], page 751,

- <in> Ada.Wide_Text_IO.Text_Streams [5901], page 751,
- <in> Ada.Wide_Wide_Text_IO.Text_Streams [5904], page 752,
- stream file [5590], page 682,
- stream type [4758], page 538,
- Stream_Access
 - <in> Ada.Streams.Stream_IO [5862], page 746,
 - <in> Ada.Text_IO.Text_Streams [5897], page 751,
 - <in> Ada.Wide_Text_IO.Text_Streams [5900], page 751,
 - <in> Ada.Wide_Wide_Text_IO.Text_Streams [5903], page 752,
- Stream_Element
 - <in> Ada.Streams [4762], page 538,
- Stream_Element_Array
 - <in> Ada.Streams [4765], page 539,
- Stream_Element_Count <subtype of> Stream_Element_Offset
 - <in> Ada.Streams [4764], page 539,
- Stream_Element_Offset
 - <in> Ada.Streams [4763], page 538,
- Stream_IO
 - <child of> Ada.Streams [5861], page 746,
- Stream_Size attribute [4769], page 540,
- Stream_Size clause [4485], page 487,
- Streams
 - <child of> Ada [4759], page 538,
- strict mode [7279], page 1103,
- String
 - <in> Standard [4847], page 562,
- string type [1889], page 122,
- String_Access
 - <in> Ada.Strings.Unbounded [5278], page 625,
- string_element [1247], page 42,
 - <used> [1246], page 42, [7774], page 1290,
- string_literal [1245], page 42,
 - <used> [2561], page 201, [3040], page 256, [8139], page 1299,
- Strings
 - <child of> Ada [5143], page 584,
 - <child of> Interfaces.C [6532], page 916,

Strlen

<in> Interfaces.C.Strings [6546], page 917,

structure

<See> record type [1951], page 130,

STS

<in> Ada.Characters.Latin_1 [5012], page 576,

STX

<in> Ada.Characters.Latin_1 [4897], page 573,

SUB

<in> Ada.Characters.Latin_1 [4921], page 574,

Sub_Second

<in> Ada.Calendar.Formatting [3793], page 367,

subaggregate

of an array_aggregate [2435], page 197,

subcomponent [1333], page 51,

subprogram [3011], page 255, [7728], page 1287,

abstract [2112], page 150,

subprogram call [3139], page 266,

subprogram instance [4288], page 456,

subprogram_body [3093], page 261,

<used> [2254], page 176, [3591], page 338, [3936], page 395, [8346], page 1306,

subprogram_body_stub [3985], page 403,

<used> [3981], page 403, [8358], page 1306,

subprogram_declaration [3017], page 255,

<used> [1287], page 49, [3579], page 338, [3590], page 338, [3927], page 395, [7788],
page 1290,

subprogram_default [4387], page 471,

<used> [4383], page 471, [4386], page 471, [8445], page 1309,

subprogram_renaming_declaration [3444], page 320,

<used> [3420], page 316, [3934], page 395, [8345], page 1306,

subprogram_specification [3020], page 255,

<used> [2108], page 150, [3019], page 255, [3095], page 261, [3446], page 320, [3987],
page 403, [4236], page 450, [4382], page 471, [4385], page 471, [8363], page 1307,

subsystem [3913], page 394, [7707], page 1285,

subtype [1337], page 51, [7729], page 1287,

constraint of [1341], page 52,

type of [1339], page 52,
values belonging to [1344], page 52,
subtype (of an object)
 <See> actual subtype of an object [1426], page 60,
 <See> actual subtype of an object [1461], page 62,
subtype conformance [3123], page 265,
 [<partial>] [2238], page 174, [3726], page 358,
 required [2092], page 147, [2236], page 173, [2765], page 223, [3434], page 317, [3453],
page 320, [3534], page 331, [3535], page 331, [3601], page 340, [3602], page 340, [3720],
page 357, [4314], page 459,
subtype conversion
 <See> type conversion [2723], page 219,
 <See also> implicit subtype conversion [2727], page 219,
subtype-specific
 of a representation item [4435], page 482,
 of an aspect [4437], page 482,
subtype_declaration [1382], page 55,
 <used> [1284], page 49, [7785], page 1290,
subtype_indication [1385], page 56,
 <used> [1384], page 55, [1438], page 61, [1486], page 66, [1831], page 114, [1834],
page 114, [1862], page 117, [2154], page 156, [2845], page 231, [3188], page 272, [3262],
page 283, [8198], page 1301,
subtype_mark [1389], page 56,
 <used> [1387], page 56, [1825], page 114, [1903], page 123, [2118], page 152, [2162],
page 156, [2407], page 194, [2544], page 201, [2732], page 219, [2827], page 229, [3048],
page 256, [3058], page 256, [3408], page 314, [3428], page 317, [4280], page 455, [4304],
page 458, [4352], page 463, [8205], page 1301,
subtypes
 of a profile [3075], page 258,
subunit [3994], page 404, [3998], page 404,
 of a program unit [3999], page 404,
 <used> [3921], page 395, [8335], page 1306,
Succ attribute [1570], page 79,
Success
 <in> Ada.Command_Line [5923], page 755,
successor element

- of a hashed set [6317], page 873,
- of a ordered set [6387], page 883,
- of a set [6249], page 855,

successor node

- of a hashed map [6192], page 844,
- of a map [6143], page 830,
- of an ordered map [6242], page 851,

Sunday

- <in> Ada.Calendar.Formatting [3781], page 365,

super

- <See> view conversion [2742], page 219,

Superscript_One

- <in> Ada.Characters.Latin_1 [5053], page 578,

Superscript_Three

- <in> Ada.Characters.Latin_1 [5046], page 577,

Superscript_Two

- <in> Ada.Characters.Latin_1 [5045], page 577,

support external streaming [4801], page 548,

Suppress pragma [4182], page 431, [7450], page 1175, [7645], page 1244,

suppressed check [4191], page 432,

Suspend_Until_True

- <in> Ada.Synchronous_Task_Control [6968], page 1015,

Suspension_Object

- <in> Ada.Synchronous_Task_Control [6964], page 1015,

Swap

- <in> Ada.Containers.Doubly_Linked_Lists [6104], page 813,
- <in> Ada.Containers.Vectors [6041], page 785, [6042], page 785,

Swap_Links

- <in> Ada.Containers.Doubly_Linked_Lists [6105], page 813,

Symmetric_Difference

- <in> Ada.Containers.Hashed_Sets [6287], page 870, [6288], page 870,
- <in> Ada.Containers.Ordered_Sets [6351], page 879, [6352], page 879,

SYN

- <in> Ada.Characters.Latin_1 [4917], page 574,

synchronization [3493], page 328,

synchronized [7730], page 1287,

synchronized interface [2127], page 152,
synchronized tagged type [2132], page 152,
Synchronous_Task_Control
 <child of> Ada [6963], page 1015,
syntactic category [1066], page 27,
syntax
 complete listing [7735], page 1289,
 cross reference [8496], page 1311,
 notation [1060], page 25,
 under Syntax heading [1014], page 22,
System [4604], page 510,
System.Address_To_Access_Conversions [4643], page 517,
System.Machine_Code [4652], page 519,
System.RPC [7155], page 1050,
System.Storage_Elements [4631], page 516,
System.Storage_Pools [4685], page 526,
System_Name
 <in> System [4606], page 510,
systems programming [6653], page 949,

30.21 T

Tag

 <in> Ada.Tags [2027], page 137,
Tag attribute [2046], page 140, [2048], page 140,
tag indeterminate [2090], page 146,
tag of an object [2019], page 137,
 class-wide object [2052], page 141,
 object created by an allocator [2051], page 141,
 preserved by type conversion and parameter passing [2055], page 141,
 returned by a function [2053], page 141, [2054], page 141,
 stand-alone object, component, or aggregate [2050], page 141,

Tag_Array

<in> Ada.Tags [2037], page 138,
Tag_Check [4200], page 441,
[<partial>] [2096], page 148, [2791], page 225, [2814], page 227, [2948], page 243,
Tag_Error
 <in> Ada.Tags [2039], page 138,
tagged incomplete view [2201], page 160,
tagged type [2013], page 136, [7731], page 1287,
 protected [2136], page 152,
 synchronized [2134], page 152,
 task [2135], page 152,
Tags
 <child of> Ada [2026], page 137,
Tail
 <in> Ada.Strings.Bounded [5269], page 620, [5270], page 620,
 <in> Ada.Strings.Fixed [5212], page 597, [5213], page 597,
 <in> Ada.Strings.Unbounded [5322], page 633, [5323], page 633,
tail (of a queue) [6817], page 979,
tamper with cursors
 of a list [6128], page 816,
 of a map [6144], page 831,
 of a set [6250], page 856,
 of a vector [6066], page 788,
tamper with elements
 of a list [6129], page 817,
 of a map [6145], page 831,
 of a set [6251], page 856,
 of a vector [6067], page 789,
Tan
 <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7241], page 1092,
 <in> Ada.Numerics.Generic_Elementary_Functions [5420], page 649,
Tanh
 <in> Ada.Numerics.Generic_Complex_Elementary_Functions [7249], page 1092,
 <in> Ada.Numerics.Generic_Elementary_Functions [5433], page 650,
target
 of an assignment operation [2942], page 242,
 of an assignment_statement [2943], page 242,

target entry
 of a requeue_statement [3718], page 356,

target object
 of a call on an entry or a protected subprogram [3616], page 343,
 of a requeue_statement [3619], page 344,

target statement
 of a goto_statement [3009], page 253,

target subtype
 of a type_conversion [2734], page 219,

task [3490], page 328,
 activation [3546], page 333,
 completion [3553], page 335,
 dependence [3552], page 335,
 execution [3544], page 333,
 termination [3554], page 335,

task declaration [3507], page 329,

task dispatching [6811], page 978,

task dispatching point [6813], page 978,
 [<partial>] [6837], page 983, [6841], page 985,

task dispatching policy [6835], page 982,
 [<partial>] [6819], page 979,

task interface [2129], page 152,

task priority [6801], page 976,

task state
 abnormal [3879], page 386,
 blocked [3500], page 328,
 callable [3893], page 388,
 held [6976], page 1016,
 inactive [3498], page 328,
 ready [3502], page 328,
 terminated [3504], page 328,

task tagged type [2137], page 152,

task type [7732], page 1288,

task unit [3497], page 328,

Task_Array
 <in> Ada.Execution_Time.Group_Budgets [7020], page 1028,

Task_Attributes

<child of> Ada [6764], page 967,

task_body [3524], page 330,

<used> [2256], page 176, [7960], page 1295,

task_body_stub [3990], page 403,

<used> [3983], page 403, [8360], page 1306,

task_definition [3517], page 329,

<used> [3512], page 329, [3516], page 329, [8237], page 1302,

Task_Dispatching_Policy pragma [6824], page 981, [7648], page 1244,

Task_Id

<in> Ada.Task_Identification [6747], page 965,

Task_Identification

<child of> Ada [6746], page 965,

task_item [3521], page 329,

<used> [3519], page 329, [8239], page 1302,

Task_Termination

<child of> Ada [6776], page 971,

task_type_declaration [3508], page 329,

<used> [1360], page 53, [7804], page 1290,

Tasking_Error

raised by failure of run-time check [3550], page 334, [3712], page 355, [4101], page 419, [4726], page 533, [4728], page 533, [6770], page 968, [6891], page 997, [6979], page 1017,

<in> Standard [4854], page 563,

template [4227], page 450,

for a formal package [4410], page 475,

<See> generic unit [4228], page 450,

term [2550], page 201,

<used> [2547], page 201, [8042], page 1297,

Term=[mentioned],Sec=[in a with_clause] [3976], page 400,

terminal interrupt

example [3866], page 385,

terminate_alternative [3829], page 379,

<used> [3822], page 378, [8309], page 1305,

terminated

a task state [3505], page 328,

Terminated attribute [3896], page 388,

termination

of a partition [7061], page 1035,

termination handler [6783], page 972,

fall-back [6786], page 972,

specific [6788], page 972,

Termination_Handler

<in> Ada.Task_Termination [6778], page 971,

Terminator_Error

<in> Interfaces.C [6529], page 907,

tested type

of a membership test [2643], page 206,

text of a program [1173], page 36,

Text_IO

<child of> Ada [5668], page 698,

Text_Streams

<child of> Ada.Text_IO [5896], page 751,

<child of> Ada.Wide_Text_IO [5899], page 751,

<child of> Ada.Wide_Wide_Text_IO [5902], page 752,

throw (an exception)

<See> raise [4089], page 419,

Thursday

<in> Ada.Calendar.Formatting [3778], page 365,

tick [1149], page 36,

<in> Ada.Real_Time [6941], page 1008,

<in> System [4615], page 511,

Tilde

<in> Ada.Characters.Latin_1 [4986], page 576,

Time

<in> Ada.Calendar [3742], page 359,

<in> Ada.Real_Time [6932], page 1008,

time base [3739], page 359,

time limit

example [3869], page 385,

time type [3738], page 359,

Time-dependent Reset procedure

of the random number generator [5472], page 658,

time-out

- example [3868], page 385,
- <See> asynchronous_select [3867], page 385,
- <See> selective_accept [3810], page 378,
- <See> timed_entry_call [3835], page 381,

Time_Error

- <in> Ada.Calendar [3754], page 361,

Time_First

- <in> Ada.Real_Time [6933], page 1008,

Time_Last

- <in> Ada.Real_Time [6934], page 1008,

Time_Of

- <in> Ada.Calendar [3753], page 360,
- <in> Ada.Calendar.Formatting [3796], page 367, [3797], page 367,
- <in> Ada.Execution_Time [6997], page 1022,
- <in> Ada.Real_Time [6952], page 1009,

Time_Of_Event

- <in> Ada.Real_Time.Timing_Events [7048], page 1031,

Time_Offset

- <in> Ada.Calendar.Time_Zones [3766], page 364,

Time_Remaining

- <in> Ada.Execution_Time.Timers [7010], page 1025,

Time_Span

- <in> Ada.Real_Time [6936], page 1008,

Time_Span_First

- <in> Ada.Real_Time [6937], page 1008,

Time_Span_Last

- <in> Ada.Real_Time [6938], page 1008,

Time_Span_Unit

- <in> Ada.Real_Time [6940], page 1008,

Time_Span_Zero

- <in> Ada.Real_Time [6939], page 1008,

Time_Unit

- <in> Ada.Real_Time [6935], page 1008,

Time_Zones

- <child of> Ada.Calendar [3765], page 364,

timed_entry_call [3836], page 381,

<used> [3807], page 378, [8298], page 1304,

Timer

<in> Ada.Execution_Time.Timers [7003], page 1025,

timer interrupt

example [3871], page 385,

Timer_Handler

<in> Ada.Execution_Time.Timers [7004], page 1025,

Timer_Resource_Error

<in> Ada.Execution_Time.Timers [7011], page 1025,

Timers

<child of> Ada.Execution_Time [7002], page 1024,

times operator [2507], page 201, [2689], page 213,

timing

<See> delay_statement [3728], page 359,

Timing_Event

<in> Ada.Real_Time.Timing_Events [7042], page 1031,

Timing_Event_Handler

<in> Ada.Real_Time.Timing_Events [7043], page 1031,

Timing_Events

<child of> Ada.Real_Time [7041], page 1031,

To_Ada

<in> Interfaces.C [6492], page 903, [6496], page 904, [6498], page 904, [6502], page 904, [6506], page 905, [6508], page 905, [6518], page 906, [6522], page 906, [6526], page 907, [6528], page 907, [6512], page 905, [6516], page 906,

<in> Interfaces.COBOLE [6595], page 933, [6597], page 933,

<in> Interfaces.Fortran [6648], page 946, [6650], page 946, [6652], page 946,

To_Address

<in> System.Address_To_Access_Conversions [4645], page 518,

<in> System.Storage_Elements [4638], page 517,

To_Basic

<in> Ada.Characters.Handling [4874], page 567, [4877], page 567,

To_Binary

<in> Interfaces.COBOLE [6627], page 935, [6630], page 935,

To_Bounded_String

<in> Ada.Strings.Bounded [5223], page 611,

To_C

<in> Interfaces.C [6491], page 903, [6495], page 904, [6497], page 904, [6501], page 904, [6505], page 905, [6507], page 905, [6521], page 906, [6525], page 907, [6527], page 907, [6511], page 905, [6515], page 906, [6517], page 906,

To_Character

<in> Ada.Characters.Conversions [5137], page 580,

To_Chars_Ptr

<in> Interfaces.C.Strings [6537], page 916,

To_COBOL

<in> Interfaces.COBOLE [6594], page 933, [6596], page 933,

To_Cursor

<in> Ada.Containers.Vectors [6011], page 782,

To_Decimal

<in> Interfaces.COBOLE [6618], page 934, [6622], page 935, [6626], page 935, [6629], page 935,

To_Display

<in> Interfaces.COBOLE [6619], page 934,

To_Domain

<in> Ada.Strings.Maps [5174], page 587,

<in> Ada.Strings.Wide_Maps [5365], page 639,

<in> Ada.Strings.Wide_Wide_Maps [5396], page 644,

To_Duration

<in> Ada.Real_Time [6943], page 1009,

To_Fortran

<in> Interfaces.Fortran [6647], page 946, [6649], page 946, [6651], page 946,

To_Index

<in> Ada.Containers.Vectors [6012], page 782,

To_Integer

<in> System.Storage_Elements [4639], page 517,

To_ISO_646

<in> Ada.Characters.Handling [4881], page 567, [4882], page 567,

To_Long_Binary

<in> Interfaces.COBOLE [6631], page 935,

To_Lower

<in> Ada.Characters.Handling [4872], page 567, [4875], page 567,

To_Mapping

- <in> Ada.Strings.Maps [5173], page 587,
- <in> Ada.Strings.Wide_Maps [5364], page 639,
- <in> Ada.Strings.Wide_Wide_Maps [5395], page 644,

To_Packed

- <in> Interfaces.COBOL [6623], page 935,

To_Picture

- <in> Ada.Text_IO.Editing [7186], page 1073,

To_Pointer

- <in> System.Address_To_Access_Conversions [4644], page 518,

To_Range

- <in> Ada.Strings.Maps [5175], page 587,
- <in> Ada.Strings.Wide_Maps [5366], page 639,
- <in> Ada.Strings.Wide_Wide_Maps [5397], page 644,

To_Ranges

- <in> Ada.Strings.Maps [5163], page 586,
- <in> Ada.Strings.Wide_Maps [5354], page 637,
- <in> Ada.Strings.Wide_Wide_Maps [5385], page 642,

To_Sequence

- <in> Ada.Strings.Maps [5169], page 586,
- <in> Ada.Strings.Wide_Maps [5360], page 638,
- <in> Ada.Strings.Wide_Wide_Maps [5391], page 643,

To_Set

- <in> Ada.Containers.Hashed_Sets [6264], page 868,
- <in> Ada.Containers.Ordered_Sets [6328], page 877,
- <in> Ada.Strings.Maps [5161], page 585, [5162], page 585, [5167], page 586, [5168], page 586,
- <in> Ada.Strings.Wide_Maps [5352], page 637, [5353], page 637, [5358], page 638, [5359], page 638,
- <in> Ada.Strings.Wide_Wide_Maps [5383], page 642, [5384], page 642, [5389], page 643, [5390], page 643,

To_String

- <in> Ada.Characters.Conversions [5140], page 580,
- <in> Ada.Strings.Bounded [5224], page 611,
- <in> Ada.Strings.Unbounded [5282], page 626,

To_Time_Span

- <in> Ada.Real_Time [6944], page 1009,

To_Unbounded_String

<in> Ada.Strings.Unbounded [5280], page 625, [5281], page 626,

To_Upper

<in> Ada.Characters.Handling [4873], page 567, [4876], page 567,

To_Vector

<in> Ada.Containers.Vectors [6003], page 781, [6004], page 781,

To_Wide_Character

<in> Ada.Characters.Conversions [5131], page 580, [5141], page 581,

To_Wide_String

<in> Ada.Characters.Conversions [5132], page 580, [5142], page 581,

To_Wide_Wide_Character

<in> Ada.Characters.Conversions [5135], page 580,

To_Wide_Wide_String

<in> Ada.Characters.Conversions [5136], page 580,

token

<See> lexical element [1175], page 36,

Trailing_Nonseparate

<in> Interfaces.COBOL [6604], page 933,

Trailing_Separate

<in> Interfaces.COBOL [6602], page 933,

transfer of control [2935], page 241,

Translate

<in> Ada.Strings.Bounded [5251], page 617, [5252], page 617, [5253], page 617, [5254], page 618,

<in> Ada.Strings.Fixed [5194], page 594, [5195], page 594, [5196], page 594, [5197], page 595,

<in> Ada.Strings.Unbounded [5304], page 631, [5305], page 631, [5306], page 631, [5307], page 631,

Translation_Error

<in> Ada.Strings [5150], page 584,

Transpose

<in> Ada.Numerics.Generic_Complex_Arrays [7363], page 1134,

<in> Ada.Numerics.Generic_Real_Arrays [7323], page 1121,

triggering_alternative [3853], page 384,

<used> [3851], page 384, [8322], page 1305,

triggering_statement [3856], page 384,

<used> [3854], page 384, [8324], page 1305,
Trim
<in> Ada.Strings.Bounded [5264], page 619, [5265], page 619, [5266], page 619,
<in> Ada.Strings.Fixed [5206], page 596, [5207], page 596, [5208], page 596, [5209],
page 596,
<in> Ada.Strings.Unbounded [5316], page 632, [5317], page 632, [5318], page 632, [5319],
page 633,
Trim_End
<in> Ada.Strings [5155], page 585,
True [1651], page 95,
Truncation
<in> Ada.Strings [5152], page 585,
Truncation attribute [5528], page 672,
Tuesday
<in> Ada.Calendar.Formatting [3776], page 365,
two's complement
modular types [1703], page 98,
type [1314], page 50, [7733], page 1288,
abstract [2110], page 150,
needs finalization [3309], page 297,
of a subtype [1338], page 52,
synchronized tagged [2133], page 152,
<See also> tag [2023], page 137,
<See also> language–defined types
type conformance [3117], page 264,
[<partial>] [1502], page 69, [3377], page 308, [3398], page 312, [4004], page 406,
required [2267], page 177, [2355], page 188, [3484], page 326, [3532], page 330, [3598],
page 339, [3600], page 339, [3719], page 356, [4312], page 459,
type conversion [2724], page 219,
access [2755], page 221, [2761], page 222, [2763], page 222, [2799], page 226,
arbitrary order [1070], page 27,
array [2750], page 220, [2780], page 224,
composite (non–array) [2744], page 220, [2789], page 225,
enumeration [2746], page 220, [2778], page 224,
numeric [2748], page 220, [2775], page 223,
unchecked [4654], page 520,

<See also> `qualified_expression` [2825], page 229,
type conversion, implicit
 <See> implicit subtype conversion [2728], page 219,
type extension [2014], page 137, [2059], page 143,
type of a `discrete_range` [1864], page 117,
type of a range [1540], page 76,
type parameter
 <See> discriminant [1891], page 123,
type profile
 <See> profile, type conformant [3119], page 264,
type resolution rules [3478], page 326,
 if any type in a specified class of types is expected [3479], page 326,
 if expected type is specific [3481], page 326,
 if expected type is universal or class-wide [3480], page 326,
type tag
 <See> tag [2021], page 137,
type-related
 aspect [4436], page 482,
 aspect [4441], page 482,
 operational item [4440], page 482,
 representation item [4434], page 482,
`type_conversion` [2729], page 219,
 <used> [2277], page 179, [7968], page 1295,
 <See also> unchecked type conversion [4656], page 520,
`type_declaration` [1351], page 53,
 <used> [1283], page 49, [7784], page 1290,
`type_definition` [1362], page 54,
 <used> [1359], page 53, [7803], page 1290,
Type_Set
 <in> `Ada.Text_IO` [5676], page 698,
types
 of a profile [3076], page 258,

30.22 U

UC_A_Acute

<in> Ada.Characters.Latin.1 [5061], page 578,

UC_A_Circumflex

<in> Ada.Characters.Latin.1 [5062], page 578,

UC_A_Diaeresis

<in> Ada.Characters.Latin.1 [5064], page 578,

UC_A_Grave

<in> Ada.Characters.Latin.1 [5060], page 578,

UC_A_Ring

<in> Ada.Characters.Latin.1 [5065], page 578,

UC_A_Tilde

<in> Ada.Characters.Latin.1 [5063], page 578,

UC_AE_Diphthong

<in> Ada.Characters.Latin.1 [5066], page 578,

UC_C_Cedilla

<in> Ada.Characters.Latin.1 [5067], page 578,

UC_E_Acute

<in> Ada.Characters.Latin.1 [5069], page 578,

UC_E_Circumflex

<in> Ada.Characters.Latin.1 [5070], page 578,

UC_E_Diaeresis

<in> Ada.Characters.Latin.1 [5071], page 578,

UC_E_Grave

<in> Ada.Characters.Latin.1 [5068], page 578,

UC_I_Acute

<in> Ada.Characters.Latin.1 [5073], page 578,

UC_I_Circumflex

<in> Ada.Characters.Latin.1 [5074], page 578,

UC_I_Diaeresis

<in> Ada.Characters.Latin.1 [5075], page 578,

UC_I_Grave

<in> Ada.Characters.Latin.1 [5072], page 578,

UC_Icelandic_Eth

<in> Ada.Characters.Latin_1 [5076], page 578,
UC_Icelandic_Thorn
<in> Ada.Characters.Latin_1 [5090], page 578,
UC_N_Tilde
<in> Ada.Characters.Latin_1 [5077], page 578,
UC_O_Acute
<in> Ada.Characters.Latin_1 [5079], page 578,
UC_O_Circumflex
<in> Ada.Characters.Latin_1 [5080], page 578,
UC_O_Diaeresis
<in> Ada.Characters.Latin_1 [5082], page 578,
UC_O_Grave
<in> Ada.Characters.Latin_1 [5078], page 578,
UC_O_Oblique_Stroke
<in> Ada.Characters.Latin_1 [5084], page 578,
UC_O_Tilde
<in> Ada.Characters.Latin_1 [5081], page 578,
UC_U_Acute
<in> Ada.Characters.Latin_1 [5086], page 578,
UC_U_Circumflex
<in> Ada.Characters.Latin_1 [5087], page 578,
UC_U_Diaeresis
<in> Ada.Characters.Latin_1 [5088], page 578,
UC_U_Grave
<in> Ada.Characters.Latin_1 [5085], page 578,
UC_Y_Acute
<in> Ada.Characters.Latin_1 [5089], page 578,
UCHAR_MAX
<in> Interfaces.C [6474], page 902,
ultimate ancestor
 of a type [1524], page 75,
unary adding operator [2673], page 213,
unary operator [2583], page 203,
unary_adding_operator [2576], page 203,
 <used> [2546], page 201, [8041], page 1297,
Unbiased_Rounding attribute [5523], page 671,

Unbounded

- <child of> Ada.Strings [5274], page 625,
- <in> Ada.Text_IO [5673], page 698,

Unbounded_IO

- <child of> Ada.Text_IO [5839], page 742,
- <child of> Ada.Wide_Text_IO [5856], page 745,
- <child of> Ada.Wide_Wide_Text_IO [5857], page 745,

Unbounded_Slice

- <in> Ada.Strings.Unbounded [5290], page 627, [5291], page 627,

Unbounded_String

- <in> Ada.Strings.Unbounded [5275], page 625,
- unchecked storage deallocation [4714], page 532,
- unchecked type conversion [4653], page 520,
- unchecked union object [6576], page 928,
- unchecked union subtype [6575], page 928,
- unchecked union type [6574], page 928,
- Unchecked_Access attribute [4679], page 525, [7405], page 1160,
- <See also> Access attribute [2224], page 169,

Unchecked_Conversion

- <child of> Ada [4658], page 520,

Unchecked_Deallocation

- <child of> Ada [4719], page 532,

Unchecked_Union pragma [6572], page 928, [7651], page 1244,

unconstrained [1346], page 52,

- object [1463], page 62,

- object [3172], page 272,

subtype [1348], page 52, [1493], page 67, [1549], page 77, [1636], page 92, [1672], page 96, [1675], page 96, [1746], page 104, [1784], page 108, [1788], page 108, [1848], page 115, [1851], page 115, [1922], page 125, [2043], page 140,

- subtype [2190], page 158,

- subtype [7462], page 1186,

unconstrained_array_definition [1820], page 114,

- <used> [1818], page 114, [7874], page 1292,

undefined result [4220], page 448,

underline [1169], page 36,

- <used> [1212], page 40, [1237], page 41, [7770], page 1289,

Uniformly_Distributed <subtype of> Float
 <in> Ada.Numerics.Float_Random [5449], page 655,
uninitialized allocator [2849], page 231,
uninitialized variables [4664], page 522,
 [<partial>] [1472], page 64,
union
 C [6570], page 928,
 <in> Ada.Containers.Hash_Sets [6281], page 869, [6282], page 869,
 <in> Ada.Containers.Ordered_Sets [6345], page 878, [6346], page 878,
unit consistency [7124], page 1044,
unit matrix
 complex matrix [7374], page 1150,
 real matrix [7334], page 1129,
unit vector
 complex vector [7373], page 1141,
 real vector [7333], page 1124,
Unit_Matrix
 <in> Ada.Numerics.Generic_Complex_Arrays [7370], page 1136,
 <in> Ada.Numerics.Generic_Real_Arrays [7330], page 1122,
Unit_Vector
 <in> Ada.Numerics.Generic_Complex_Arrays [7350], page 1133,
 <in> Ada.Numerics.Generic_Real_Arrays [7322], page 1120,
universal type [1518], page 74,
universal_fixed
 [<partial>] [1723], page 102,
universal_integer [1704], page 98,
 [<partial>] [1682], page 96,
universal_real
 [<partial>] [1721], page 102,
unknown discriminants [1923], page 125,
unknown_discriminant_part [1896], page 123,
 <used> [1894], page 123, [7891], page 1293,
Unknown_Zone_Error
 <in> Ada.Calendar.Time_Zones [3767], page 364,
unmarshalling [7132], page 1045,
unpolluted [4760], page 538,

unsigned

<in> Interfaces.C [6479], page 902,

<in> Interfaces.COBOL [6600], page 933,

unsigned type

<See> modular type [1656], page 95,

unsigned_char

<in> Interfaces.C [6482], page 902,

unsigned_long

<in> Interfaces.C [6481], page 902,

unsigned_short

<in> Interfaces.C [6480], page 902,

unspecified [1055], page 25,

[<partial>] [1124], page 32, [2025], page 137, [2040], page 139, [2644], page 208, [2647], page 209, [2699], page 216, [3087], page 261, [3249], page 282, [3883], page 387, [4048], page 412, [4106], page 419, [4209], page 447, [4444], page 484, [4646], page 518, [4668], page 522, [4710], page 528, [4790], page 547, [4834], page 556, [5444], page 653, [5470], page 657, [5471], page 658, [5526], page 672, [5584], page 681, [5661], page 697, [5824], page 724, [5825], page 725, [5826], page 726, [5827], page 726, [5917], page 754, [6068], page 807, [6076], page 809, [6065], page 788, [6130], page 828, [6134], page 829, [6127], page 816, [6138], page 830, [6148], page 839, [6187], page 844, [6188], page 844, [6189], page 844, [6193], page 844, [6238], page 851, [6239], page 851, [6245], page 855, [6256], page 866, [6252], page 865, [6253], page 865, [6312], page 873, [6313], page 873, [6314], page 873, [6318], page 873, [6319], page 875, [6320], page 876, [6388], page 887, [6383], page 883, [6384], page 883, [6396], page 892, [6398], page 893, [6836], page 982, [6955], page 1010, [7122], page 1044, [7232], page 1089, [7259], page 1095, [7262], page 1097, [7377], page 1153, [7384], page 1154, [7478], page 1208,

Unsuppress pragma [4185], page 431, [7654], page 1244,

update

the value of an object [1423], page 59,

<in> Interfaces.C.Strings [6547], page 917, [6548], page 917,

Update_Element

<in> Ada.Containers.Doubly_Linked_Lists [6093], page 812,

<in> Ada.Containers.Hashed_Maps [6164], page 841,

<in> Ada.Containers.Ordered_Maps [6207], page 847,

<in> Ada.Containers.Vectors [6019], page 782, [6020], page 783,

Update_Element_Preserving_Key

- <in> Ada.Containers.Hashtable_Sets [6309], page 872,
- <in> Ada.Containers.Ordered_Sets [6381], page 883,

Update_Error

- <in> Interfaces.C.Strings [6549], page 917,

upper bound

- of a range [1539], page 76,

upper-case letter

- a category of Character [4887], page 569,

Upper_Case_Map

- <in> Ada.Strings.Maps.Constants [5337], page 636,

Upper_Set

- <in> Ada.Strings.Maps.Constants [5329], page 636,

US

- <in> Ada.Characters.Latin_1 [4926], page 574,

usage name [1308], page 50,

use-visible [3370], page 308, [3413], page 315,

use_clause [3401], page 314,

- <used> [2249], page 175, [3965], page 400, [4242], page 450, [8351], page 1306,

Use_Error

- <in> Ada.Direct_IO [5645], page 692,
- <in> Ada.Directories [5963], page 761,
- <in> Ada.IO_Exceptions [5909], page 752,
- <in> Ada.Sequential_IO [5615], page 684,
- <in> Ada.Streams.Stream_IO [5891], page 748,
- <in> Ada.Text_IO [5817], page 708,

use_package_clause [3404], page 314,

- <used> [3402], page 314, [8200], page 1301,

use_type_clause [3407], page 314,

- <used> [3403], page 314, [8201], page 1301,

user-defined assignment [3290], page 295,

user-defined heap management [4683], page 526,

user-defined operator [3216], page 276,

user-defined storage management [4681], page 526,

UTC_Time_Offset

- <in> Ada.Calendar.Time_Zones [3768], page 364,

30.23 V

Val attribute [1709], page 100,

Valid

<in> Ada.Text_IO.Editing [7185], page 1073, [7197], page 1074,

<in> Interfaces.COBOLE [6616], page 934, [6620], page 935, [6624], page 935,

Valid attribute [4677], page 524, [7379], page 1153,

Value

<in> Ada.Calendar.Formatting [3802], page 369, [3804], page 369,

<in> Ada.Environment_Variables [5977], page 775,

<in> Ada.Numerics.Discrete_Random [5469], page 656,

<in> Ada.Numerics.Float_Random [5458], page 655,

<in> Ada.Strings.Maps [5171], page 587,

<in> Ada.Strings.Wide_Maps [5362], page 639,

<in> Ada.Strings.Wide_Wide_Maps [5393], page 644,

<in> Ada.Task_Attributes [6766], page 968,

<in> Interfaces.C.Pointers [6556], page 923, [6557], page 923,

<in> Interfaces.C.Strings [6542], page 916, [6543], page 916, [6544], page 917, [6545],

page 917,

Value attribute [1617], page 89,

value conversion [2740], page 219,

values

belonging to a subtype [1343], page 52,

variable [1417], page 59,

variable object [1419], page 59,

variable view [1421], page 59,

variant [1989], page 134,

<used> [1987], page 134, [7921], page 1294,

<See also> tagged type [2009], page 136,

variant_part [1985], page 134,

<used> [1960], page 130, [7913], page 1293,

Vector

- <in> Ada.Containers.Vectors [5999], page 780,
- vector container [5991], page 779,
- Vectors
 - <child of> Ada.Containers [5996], page 780,
- version
 - of a compilation unit [7121], page 1044,
- Version attribute [7118], page 1043,
- vertical line [1170], page 36,
- Vertical_Line
 - <in> Ada.Characters.Latin_1 [4984], page 576,
- view [1301], page 50, [7734], page 1288,
- view conversion [2738], page 219,
- virtual function
 - <See> dispatching subprogram [2082], page 145,
- Virtual_Length
 - <in> Interfaces.C.Pointers [6561], page 924,
- visibility
 - direct [3363], page 308, [3389], page 311,
 - immediate [3367], page 308, [3390], page 311,
 - use clause [3368], page 308, [3414], page 315,
- visibility rules [3362], page 308,
- visible [3366], page 308, [3381], page 310,
 - attribute_definition_clause [3394], page 311,
 - within a use_clause in a context_clause [4022], page 409,
 - within a pragma in a context_clause [4024], page 409,
 - within a pragma that appears at the place of a compilation unit [4028], page 409,
 - within a with_clause [4020], page 409,
 - within the parent_unit_name of a library unit [4018], page 409,
 - within the parent_unit_name of a subunit [4026], page 409,
- visible part [3353], page 306,
 - of a formal package [4412], page 476,
 - of a generic unit [3357], page 307,
 - of a package (other than a generic formal package) [3239], page 279,
 - of a protected unit [3594], page 339,
 - of a task unit [3529], page 330,
 - of a view of a callable entity [3355], page 307,

of a view of a composite type [3356], page 307,
volatile [6733], page 963,
Volatile pragma [6724], page 963, [7657], page 1244,
Volatile_Components pragma [6730], page 963, [7660], page 1244,
VT
 <in> Ada.Characters.Latin_1 [4906], page 573,
VTS
 <in> Ada.Characters.Latin_1 [5003], page 576,

30.24 W

wchar_array
 <in> Interfaces.C [6503], page 904,
wchar_t
 <in> Interfaces.C [6499], page 904,
Wednesday
 <in> Ada.Calendar.Formatting [3777], page 365,
well-formed picture String
 for edited output [7181], page 1059,
Wide_Bounded
 <child of> Ada.Strings [5340], page 636,
Wide_Character [1643], page 94,
 <in> Standard [4843], page 562,
Wide_Character_Mapping
 <in> Ada.Strings.Wide_Maps [5361], page 638,
Wide_Character_Mapping_Function
 <in> Ada.Strings.Wide_Maps [5367], page 639,
Wide_Character_Range
 <in> Ada.Strings.Wide_Maps [5350], page 637,
Wide_Character_Ranges
 <in> Ada.Strings.Wide_Maps [5351], page 637,
Wide_Character_Sequence <subtype of> Wide_String
 <in> Ada.Strings.Wide_Maps [5357], page 638,

Wide_Character_Set

<in> Ada.Strings.Wide_Maps [5348], page 637,

<in> Ada.Strings.Wide_Maps.Wide_Constants [5401], page 646,

Wide_Characters

<child of> Ada [4858], page 566,

Wide_Constants

<child of> Ada.Strings.Wide_Maps [5346], page 636, [5399], page 644,

Wide_Exception_Name

<in> Ada.Exceptions [4144], page 423, [4154], page 424,

Wide_Expanded_Name

<in> Ada.Tags [2030], page 138,

Wide_Fixed

<child of> Ada.Strings [5339], page 636,

Wide_Hash

<child of> Ada.Strings [5342], page 636,

<child of> Ada.Strings.Wide_Bounded [5344], page 636,

<child of> Ada.Strings.Wide_Fixed [5343], page 636,

<child of> Ada.Strings.Wide_Unbounded [5345], page 636,

Wide_Image attribute [1588], page 83,

Wide_Maps

<child of> Ada.Strings [5347], page 637,

wide_nul

<in> Interfaces.C [6500], page 904,

Wide_Space

<in> Ada.Strings [5145], page 584,

Wide_String

<in> Standard [4848], page 563,

Wide_Text_IO

<child of> Ada [5848], page 745,

Wide_Unbounded

<child of> Ada.Strings [5341], page 636,

Wide_Value attribute [1611], page 88,

Wide_Wide_Bounded

<child of> Ada.Strings [5371], page 641,

Wide_Wide_Character [1646], page 94,

<in> Standard [4844], page 562,

Wide_Wide_Character_Mapping
 <in> Ada.Strings.Wide_Wide_Maps [5392], page 644,

Wide_Wide_Character_Mapping_Function
 <in> Ada.Strings.Wide_Wide_Maps [5398], page 644,

Wide_Wide_Character_Range
 <in> Ada.Strings.Wide_Wide_Maps [5381], page 642,

Wide_Wide_Character_Ranges
 <in> Ada.Strings.Wide_Wide_Maps [5382], page 642,

Wide_Wide_Character_Sequence <subtype of> Wide_Wide_String
 <in> Ada.Strings.Wide_Wide_Maps [5388], page 643,

Wide_Wide_Character_Set
 <in> Ada.Strings.Wide_Wide_Maps [5379], page 642,

Wide_Wide_Characters
 <child of> Ada [4859], page 566,

Wide_Wide_Constants
 <child of> Ada.Strings.Wide_Wide_Maps [5377], page 641,

Wide_Wide_Exception_Name
 <in> Ada.Exceptions [4145], page 423, [4155], page 424,

Wide_Wide_Expanded_Name
 <in> Ada.Tags [2031], page 138,

Wide_Wide_Fixed
 <child of> Ada.Strings [5370], page 641,

Wide_Wide_Hash
 <child of> Ada.Strings [5373], page 641,
 <child of> Ada.Strings.Wide_Wide_Bounded [5375], page 641,
 <child of> Ada.Strings.Wide_Wide_Fixed [5374], page 641,
 <child of> Ada.Strings.Wide_Wide_Unbounded [5376], page 641,

Wide_Wide_Image attribute [1584], page 81,

Wide_Wide_Maps
 <child of> Ada.Strings [5378], page 642,

Wide_Wide_Space
 <in> Ada.Strings [5146], page 584,

Wide_Wide_String
 <in> Standard [4849], page 563,

Wide_Wide_Text_IO
 <child of> Ada [5851], page 745,

Wide_Wide_Unbounded

<child of> Ada.Strings [5372], page 641,

Wide_Wide_Value attribute [1599], page 86,

Wide_Wide_Width attribute [1593], page 85,

Wide_Width attribute [1595], page 85,

Width attribute [1597], page 85,

with_clause [3966], page 400,

mentioned in [3977], page 400,

named in [3979], page 400,

<used> [3964], page 400, [8350], page 1306,

within

immediately [3346], page 305,

word [4493], page 487,

Word_Size

<in> System [4619], page 511,

Write

<in> Ada.Direct_IO [5637], page 692,

<in> Ada.Sequential_IO [5610], page 684,

<in> Ada.Storage_IO [5653], page 696,

<in> Ada.Streams [4767], page 539,

<in> Ada.Streams.Stream_IO [5881], page 748, [5882], page 748,

<in> System.RPC [7160], page 1051,

Write attribute [4772], page 541, [4776], page 542,

Write clause [4487], page 487, [4797], page 547,

30.25 X

xor operator [2458], page 201, [2598], page 205,

30.26 Y

Year

<in> Ada.Calendar [3748], page 360,

<in> Ada.Calendar.Formatting [3787], page 366,

Year_Number <subtype of> Integer

<in> Ada.Calendar [3743], page 360,

Yen_Sign

<in> Ada.Characters.Latin_1 [5031], page 577,

Concept Index

&

& 201, 212
& operator 201, 212

*

* 201, 213
* operator 201, 213
** 201, 218
** operator 201, 218

+

+ 201, 211, 213
+ operator 201, 211, 213

-

- 201, 211, 213
- operator 201, 211, 213

/

/ 201, 213
/ operator 201, 213
/= 201, 206
/= operator 201, 206

<

< 201, 206
< operator 201, 206
<= 201, 206
<= operator 201, 206
<child of> Ada... 137, 296, 359, 423, 428, 520, 532, 538
<child of> Ada.Calendar 364, 365
<child of> Ada.Tags 141
<child of> System 516, 518, 519, 526
<in> Ada.Calendar 359, 360, 361
<in> Ada.Calendar.Arithmetic 364, 365
<in> Ada.Calendar.Formatting 365, 366, 367, 368, 369
<in> Ada.Calendar.Time_Zones 364
<in> Ada.Exceptions 423, 424
<in> Ada.Finalization 296
<in> Ada.Streams 538, 539
<in> Ada.Tags 137, 138
<in> System 510, 511, 512
<in> System.Address_To_Access_Conversions .. 518
<in> System.Storage_Elements 516, 517
<in> System.Storage_Pools 526, 527
<See also> abstract type 149

<See also> Access attribute 169
<See also> aggregate 190
<See also> allocator 230
<See also> composite type 51
<See also> discriminant 123
<See also> dispatching operation 136
<See also> exception 419
<See also> format_effector 35
<See also> implicit subtype conversion 219
<See also> literal 189
<See also> loop parameter 249
<See also> not operator 218
<See also> package 279
<See also> pragma Storage_Size 496
<See also> qualified_expression 229
<See also> static 234
<See also> tag 137
<See also> tagged type 136
<See also> tagged types and type extension... 136
<See also> task 328
<See also> unchecked type conversion 520
<See also> Unchecked_Access attribute 525
<See> access type 156
<See> access value 156
<See> actual subtype of an object 60, 62
<See> allocator 230
<See> aspect_clause 481
<See> assignment operation 242
<See> asynchronous_select 385
<See> concatenation operator 201, 212
<See> delay_statement 359
<See> delay_until_statement 363
<See> derived types and classes 66
<See> discriminant 123
<See> dispatching call 145
<See> dispatching operation 136
<See> dispatching operations of tagged types .. 145
<See> dispatching subprogram 145
<See> distinct access paths 261
<See> finalization 295, 299
<See> formal parameter 257
<See> generic formal parameter 450
<See> generic unit 450
<See> handle 419
<See> implicit subtype conversion 219
<See> indexed_component 181
<See> informative 21
<See> initialization 63, 295
<See> initialization expression 61
<See> Initialize 295
<See> initialized allocator 231
<See> intertask communication 343
<See> language-defined check 431
<See> lexical element 36

<See> library 394, 407
 <See> logical operators on boolean arrays 205
 <See> modular type 95
 <See> nonabstract subprogram 150
 <See> nonabstract type 149
 <See> nonexistent 533
 <See> null access value 189
 <See> ordering of storage elements in a word .. 508
 <See> package 279
 <See> partition building 410
 <See> post-Compilation error 22, 28
 <See> private types and private extensions 283
 <See> profile, type conformant 264
 <See> program execution 409
 <See> protected object 338
 <See> raise 419
 <See> record type 130
 <See> record_representation_clause 501
 <See> relational operator 206
 <See> requeue 356
 <See> selected_component 183
 <See> selective_accept 378
 <See> storage element 487
 <See> string_literal 42
 <See> tag 137
 <See> tagged types and type extensions 136
 <See> task 328
 <See> timed_entry_call 381
 <See> type conversion 219
 <See> type System.Address 515
 <See> unchecked type conversion 520
 <See> unspecified 25
 <See> view conversion 219
 <used> 38, 40, 41,
 42, 43, 44, 46, 49, 53, 54, 55, 56, 61, 65, 66, 76, 92,
 95, 102, 103, 106, 107, 114, 117, 123, 127, 130, 134,
 143, 150, 152, 156, 160, 175, 176, 179, 181, 182, 183,
 184, 187, 190, 191, 194, 196, 201, 219, 229, 231, 240,
 241, 242, 245, 246, 248, 249, 251, 252, 253, 255, 256,
 261, 266, 267, 272, 275, 277, 279, 281, 283, 314, 316,
 317, 318, 319, 320, 323, 329, 330, 338, 347, 348, 352,
 356, 359, 377, 378, 379, 381, 383, 384, 385, 395, 399,
 400, 403, 404, 413, 414, 416, 418, 419, 420, 421, 427,
 431, 450, 451, 455, 458, 460, 461, 463, 467, 468, 470,
 471, 475, 481, 486, 487, 496, 500, 502, 518, 534, 535

=
 = 201, 206
 = operator 201, 206

>
 > 201, 206
 > operator 201, 206
 >= 201, 206
 >= operator 201, 206

[
 [<partial>] 32, 49, 50, 51, 52, 54, 56, 64, 69, 70,
 75, 79, 80, 86, 87, 88, 89, 90, 96, 97, 100, 102, 108,
 124, 128, 136, 137, 139, 143, 148, 153, 162, 172, 173,
 174, 176, 180, 181, 183, 186, 189, 191, 195, 199, 202,
 205, 208, 209, 212, 216, 218, 219, 222, 223, 224, 225,
 226, 227, 230, 231, 232, 237, 243, 247, 261, 274, 275,
 282, 285, 308, 312, 313, 318, 320, 331, 340, 349, 358,
 387, 395, 406, 412, 415, 419, 447, 449, 459, 460, 463,
 471, 484, 497, 508, 517, 518, 521, 522, 528, 546, 547

1
 10646:2003, ISO/IEC standard 30
 14882:2003, ISO/IEC standard 30
 1539-1:2004, ISO/IEC standard 30
 16 and 32-bit 30
 19769:2004, ISO/IEC technical report 30
 1989:2002, ISO standard 30

6
 6429:1992, ISO/IEC standard 30
 646:1991, ISO/IEC standard 30

7
 7-bit 30

8
 8-bit 30
 8859-1:1987, ISO/IEC standard 30

9
 9899:1999, ISO/IEC standard 30

A

- a discriminant constraint 128
- a primitive subprogram 58
- a range constraint 76
- a task state 328
- a type 75
- a type by a subtype_mark 56
- AARM 4
- abnormal 299, 386
- abnormal completion 299
- abnormal state of an object 522
- abnormal task 386
- abort completion point 387
- abort—deferred operation 386
- abort_statement 385, 386
- abort_statement task_name 386
- abortable_part 384
- aborting the execution of a construct 386
- abs 201, 217
- abs operator 201, 217
- absolute value 201, 217
- abstract 150
- abstract subprogram 150
- abstract type 149, 150
- abstract_subprogram_declaration 150, 151
- accept_alternative 379
- accept_statement 347, 350
- accept_statement entry_direct_name 348
- acceptable interpretation 325
- access 221, 222, 226
- Access 168, 173
- Access attribute 168, 172, 173
- access attribute_reference 164
- Access attribute_reference prefix 164
- access discriminant 124, 126
- access parameter 257
- access result type 257
- access type 156
- access value 156
- access—to—constant type 157
- access—to—object type 157
- access—to—subprogram type 157
- access—to—variable type 157
- Access_Check 180, 227, 432
- access_definition 156, 159
- access_to_object_definition 156
- access_to_subprogram_definition 156
- access_type_definition 156, 158
- accessibility 164
- accessibility level 164
- Accessibility_Check 172, 225, 226, 232, 274, 443
- accuracy 224
- actions 391
- activation 333
- activation failure 333
- active partition 412
- actual 455
- actual parameter 270
- actual subtype 60, 461
- actual type 461
- actual_parameter_part 267
- Ada 263
- Ada calling convention 263
- Ada program 328
- Ada.Assertions 428
- Ada.Calendar 359
- Ada.Calendar.Arithmetic 364
- Ada.Calendar.Formatting 365
- Ada.Calendar.Time_Zones 364
- Ada.Exceptions 423
- Ada.Finalization 296
- Ada.Streams 538
- Ada.Tags 137
- Ada.Tags.Generic_Dispatching_Constructor 141
- Ada.Unchecked_Conversion 520
- Ada.Unchecked_Deallocation 532
- Address 488
- Address attribute 488
- Address clause 487, 488
- Adjust 295
- adjusting the value of an object 298
- adjustment 298
- advice 23
- Aft 111
- Aft attribute 111
- aggregate 190
- aliased 157
- Alignment 489, 490
- Alignment attribute 489, 490
- Alignment attribute for objects 491
- Alignment attribute for subtypes 491
- Alignment clause 487, 490
- All_Checks 447
- Allocation_Check 232, 443
- allocator 231, 232
- ambiguous 327
- among compilation units 406
- ampersand 36, 201, 212
- ampersand operator 201, 212
- an entry call 353
- an entry queue 354
- an exception 419, 421
- an exception occurrence 422, 423
- an exception occurrence by an execution, to a
 dynamically enclosing execution 423
- an index constraint 118
- ancestor_part 194
- and 201, 205
- and operator 201, 205
- and then (short—circuit control form) 201, 204
- Annotated Ada Reference Manual 4
- anonymous access type 157
- anonymous array type 61
- anonymous protected type 61
- anonymous task type 61
- anonymous type 54

apostrophe..... 36
 applicable index constraint..... 197
 application areas..... 20
 arbitrary order..... 27
 argument of a pragma..... 45
 arithmetic..... 516
 array..... 114, 220, 224
 array bounds..... 225
 array component expression..... 197
 array index..... 181
 array slice..... 182
 array type..... 114
 array_aggregate..... 196, 197, 198
 array_aggregate component expression..... 197
 array_aggregate discrete_choice..... 197
 array_component_association..... 196
 array_type_definition..... 114, 116
 as defined between actions..... 390
 as part of assignment..... 243
 aspect..... 482
 aspect of representation.... 482, 486, 500, 501, 534
 aspect_clause..... 481, 484
 Assert..... 427
 Assert pragma..... 427
 assertion policy..... 428
 Assertion_Policy..... 427
 Assertion_Policy pragma..... 427
 Assertions..... 427
 assigning back of parameters..... 272
 assignment operation..... 242, 243, 297
 assignment to view conversion..... 227
 assignment_statement..... 242, 243, 298, 301
 assignment_statement expression..... 242
 assignment_statement variable_name..... 242
 associated with a dereference..... 180
 associated with a designated profile..... 157
 associated with a protected object..... 341
 associated with a type_conversion..... 223
 associated with an indexed_component..... 181
 asterisk..... 36
 asynchronous_select..... 384
 asynchronous_select with a
 delay_statement trigger..... 384
 asynchronous_select with a
 procedure call trigger..... 384
 asynchronous_select with an
 entry call trigger..... 384
 attribute..... 187
 attribute_definition_clause..... 311, 487
 attribute_definition_clause
 expression or name..... 487
 attribute_definition_clause name..... 487
 attribute_designator..... 187
 attribute_designator expression..... 187
 attribute_reference..... 187, 188

B

base..... 41
 Base..... 78
 base 16 literal..... 41
 base 2 literal..... 41
 base 8 literal..... 41
 Base attribute..... 78
 based..... 41
 based_literal..... 41
 based_numeral..... 41
 basic_declaration..... 49
 basic_declarative_item..... 175
 become nonlimited..... 287, 294
 becoming nonlimited..... 287, 294
 belonging to a subtype..... 52
 between tasks..... 328
 bibliography..... 30
 big..... 508
 big endian..... 508
 binary..... 203
 binary adding..... 211
 binary adding operator..... 211
 binary literal..... 41
 binary operator..... 203
 binary_adding_operator..... 203
 bit ordering..... 508, 509
 Bit_Order..... 508
 Bit_Order attribute..... 508
 Bit_Order clause..... 487, 508
 block_statement..... 251
 blocked..... 328
 blocking, potentially..... 345
 BMP..... 93, 94
 body..... 176, 177
 body_stub..... 403
 Boolean..... 95
 boolean type..... 95
 bounded error..... 22, 29
 bounds of a decimal fixed point type..... 108
 bounds of a fixed point type..... 108
 bounds of a range..... 77, 116
 by a compilation unit..... 410
 by a constituent of a construct..... 550
 by a discrete_choice..... 134
 by a discrete_choice_list..... 135
 by a protected entry..... 339
 by a protected subprogram..... 339
 by a task entry..... 330
 by an expression..... 550
 by an implicit call..... 551
 by an inner homograph..... 311
 by an object name..... 550
 by copy parameter passing..... 260
 by lack of a with_clause..... 311
 by reference parameter passing..... 260
 by-copy type..... 260
 by-reference type..... 260

C

C standard	30
C++ standard	30
call	255
call on a dispatching operation	146, 147
call on an inherited subprogram	70
Callable	388
callable	388
Callable attribute	388
callable construct	255
callable entity	255
calling convention	263
canonical semantics	448
case expression	247
case insensitive	39
case_statement	246, 247
case_statement_alternative	246
case_statement_alternative_discrete_choice	247
category determined for a formal type	461
cause	129, 142, 232, 261, 301, 341, 345, 387, 391, 412, 447, 488, 491, 523, 528, 533, 534, 548
cease to	301, 533
change of	509
change of representation	509
Character	93
character	32
character plane	32
character set	32
character type	93
character_literal	42, 189
characteristics	285
Checking pragmas	431
choice parameter	420
choice_parameter_specification	420, 423
choices of aggregate	198
Class	139, 288
Class attribute	140, 288
class expected type	327
class factory	143
class-wide object	141
class-wide type	72, 126
class-wide type caused by the freezing of the specific type	552
clock	359
closed	353
closed entry	353
closed under derivation	70
COBOL standard	30
code_statement	518, 519
coding	500
colon	36
comma	36
comment	43
comments, instructions for submission	13
comparison	511
compilation	395
Compilation unit	394, 395
compilation_unit	395
compile-time	22, 28
compile-time concept	177
compile-time error	22, 28
compile-time semantics	22
complete	345
complete context	324
completely defined	177
completion	335
completion and leaving (completed and left)	299
component	51
component defaults	62
component subtype	115
component_choice_list	191
component_clause	502
component_clause expressions	502
component_declaration	130, 131
component_declaration_default_expression	130
component_definition	114, 116, 132
component_item	130
component_list	130, 131
Component_Size	497
Component_Size attribute	497
Component_Size clause	487, 497
composite (non-array)	220, 225
composite type	51
composite_constraint	56
composite_constraint with an access subtype	158
compound delimiter	37, 115
compound_statement	240
concatenation	201, 212
concatenation operator	201, 212
condition	245
conditional_entry_call	383
configuration pragma	408
conformance	263
constant	59, 236
constant object	59
constant view	59
constituents of a full type definition	551
constrained	52, 237
Constrained	129
Constrained attribute	129
constrained by its initial value	62
constrained_array_definition	114
constraint	56, 236
constraint of	52
constraint with a subtype	56
Construct	27
construct	22
context_clause	399
context_item	400
control functions	30
controlled	534
Controlled	534
Controlled pragma	534
controlled type	295, 296
controlling formal parameter	146
controlling operand	146

controlling result	146
controlling tag value	147
convention	263
conversion	219, 223
convertible	219
copy back of parameters	272
copy parameter passing	260
core language	20
corresponding constraint	67
corresponding discriminants	125
corresponding subtype	69
corresponding to a formal object of mode in	460
Count	389
Count attribute	389
create	50

D

date and time formatting standard	30
deallocation of storage	532
decimal	40
decimal fixed point type	106, 107
decimal fixed point type digits	107
decimal_fixed_point_definition	107
decimal_literal	40
Declaration	49
declaration of a partial view	284
declaration to which a pragma	
Elaborate_Body applies	418
declarative_item	175
declarative_part	175, 176
declare	50
declared pure	417
default entry queuing policy	354
default policy	354
default_expression	123
default_name	471
deferred constant	290
deferred constant declaration	61, 290, 291
defining name	50
defining_character_literal	92
defining_designator	255
defining_identifier	49
defining_identifier_list	61
defining_operator_symbol	256
defining_program_unit_name	256
Definite	465
Definite attribute	465
definite subtype	60
definition	50
delay expression	361
delay_alternative	379
delay_relative_statement	359
delay_relative_statement expression	359
delay_statement	359, 361, 363
delay_until_statement	359
delay_until_statement expression	359
delimiter	37

Delta	110
Delta attribute	110
delta constraint	236
denote	235, 325
dependence	335
dereference	179, 180
dereference name	179
derived type	66
derived type discriminants	69
derived_type_definition	66, 70
descendant	396
designate	156
designated subtype caused by an allocator	551
designator	255
determined category for a formal type	461
digit	40
Digits	105, 111
Digits attribute	105, 111
digits constraint	236
digits_constraint	107, 108
digits_constraint with a decimal	
fixed point subtype	108
direct	308, 311
direct_name	179
directly or indirectly	72
directly visible	308, 311
discrete array type	206
discrete type	51, 76
discrete_choice	134
discrete_choice_list	134
discrete_range	117, 118, 236
discrete_subtype_definition	114, 116
discrete_subtype_definition range	114
discriminant	51, 123
discriminant constraint	236
discriminant constraint with a subtype	128
discriminant default_expression	123
discriminant values	129
discriminant_association	127
discriminant_association expression	128
Discriminant_Check	186, 191, 195, 226, 227, 230, 232, 433
discriminant_constraint	127, 128
discriminant_part	123
discriminant_specification	123
discriminated type	124
dispatching	137
dispatching operation	145, 146
disruption of an assignment	387, 522
distinct	261
distinct access paths	261
divide	36, 201, 213
divide operator	201, 213
Division_Check	97, 216, 434
documentation (required of an	
implementation)	25
documentation requirements	22
dot	36

downward 174
 downward closure 174
 during an entry call 354
 during elaboration of an object_declaration 63
 during evaluation of a generic_association for a
 formal object of mode in 460
 during evaluation of a parameter_association .. 271
 during evaluation of an aggregate 190
 during evaluation of an initialized allocator 232
 during evaluation of an
 uninitialized allocator 232
 during evaluation of concatenation 212
 during execution of a for loop 249
 during execution of an assignment_statement .. 243
 during parameter copy back 272
 dynamic semantics 22
 dynamically determined tag 145
 dynamically enclosing 422
 dynamically tagged 146

E

effect on subtype-specific aspects 483
 efficiency 448
 Elaborate 418
 Elaborate pragma 418
 Elaborate_All 418
 Elaborate_All pragma 418
 Elaborate_Body 418
 Elaborate_Body pragma 418
 elaborated 176
 elaboration 50, 410
 elaboration control 413
 Elaboration_Check 176, 444
 elementary type 51
 end of a line 36
 entity 550
 entity caused by a body 550
 entity caused by a construct 550
 entity caused by a name 551
 entity caused by the end of an
 enclosing construct 550
 entry 264, 350
 entry call 352
 entry calling convention 264
 entry family 350
 entry index 350
 entry index subtype 131, 350
 entry queue 354
 entry queuing policy 354
 entry_barrier 348
 entry_body 348, 349, 351
 entry_body_formal_part 348
 entry_call_alternative 381
 entry_call_statement 352, 353
 entry_declaration 347, 350
 entry_index 348
 entry_index_specification 348

enumeration 220, 224
 enumeration literal 92
 enumeration type 51, 92
 enumeration_aggregate 500
 enumeration_literal_specification 92
 enumeration_representation_clause 500, 501
 enumeration_representation_clause
 expressions 500
 enumeration_type_definition 92
 environment 406
 environment declarative_part 406
 environment task 410
 equal 201, 206
 equal operator 201, 206
 equality 206
 equality operator 206
 equals sign 36
 erroneous execution 22, 29
 evaluation 50
 example 363, 385
 example using asynchronous_select 385
 Exception 419
 exception occurrence 419
 exception_choice 420
 exception_declaration 419
 exception_handler 420
 exception_renaming_declaration 318
 exclamation point 36
 execution 50, 333
 execution of a selective_accept 380
 execution resource associated with
 protected object 345
 exit_statement 252
 expanded name 184
 expected profile 326
 expected type 326
 explicit declaration 49
 explicit initial value 61
 explicit_actual_parameter 267
 explicit_dereference 179
 explicit_generic_actual_parameter 455
 explicitly assign 410
 explicitly limited 131
 explicitly limited record 131
 exponent 40, 218
 exponentiation 201, 218
 exponentiation operator 201, 218
 expression 201, 234
 expression of extended_return_statement 272
 expression of simple_return_statement 272
 expressions in aggregate 193
 expressions of aggregate 199
 extended_digit 41
 extended_return_statement 272, 273
 extension_aggregate 194, 195
 extension_aggregate_ancestor_expression 195
 extension_aggregate
 record_component_association_list 195

external 24
 external call 344
 external interaction 24
 external requeue 344
 External_Tag 498
 External_Tag attribute 498
 External_Tag clause 487, 498
 extra permission to avoid raising exceptions ... 448
 extra permission to reorder actions 449

F

factor 201
 factory 143
 False 95
 finalization of 301
 finalization of the collection 301
 Finalize 295
 First 77, 119
 First attribute 77, 119
 first subtype 54, 73
 first subtype caused by the
 freezing of the type 551
 First(N) 119
 First(N) attribute 119
 first_bit 502
 First_Bit 506
 First_Bit attribute 507
 fixed point type 106
 fixed point type delta 107
 fixed_point_definition 106, 108
 Float 104
 floating point type 103
 floating_point_definition 103, 104
 for a call on a dispatching operation 145
 for a component 125
 for a constraint and a scalar subtype 239
 for a constraint and an access or
 composite subtype 239
 for a constraint or component_definition 125
 for a declaration completed by a
 subsequent declaration 310
 for a delay_relative_statement 361
 for a delay_until_statement 361
 for a formal package 475
 for a formal parameter 270
 for a partition 410
 for a subtype 62
 for a type 72
 for an access value 158
 for an array_aggregate 197
 for constraints 238
 for direct_names 311
 for discrete_subtype_definitions 265
 for expressions 265
 for known_discriminant_parts 265
 for overridden declaration 310
 for profiles 265

for ranges 239
 for root numeric operators and ranges 327
 for selector_names 311
 for subtypes 239
 for the environment task of a partition 411
 for the expression in an
 assignment_statement 243
 for two subtypes 239
 Fore 110
 Fore attribute 110
 formal object, generic 458
 formal package, generic 474
 formal subprogram actual 471
 formal subprogram default_name 471
 formal subprogram, generic 470
 formal subtype 461
 formal type 461
 formal_abstract_subprogram_declaration 471
 formal_access_type_definition 468
 formal_array_type_definition 467
 formal_concrete_subprogram_declaration 471
 formal_decimal_fixed_point_definition 467
 formal_derived_type_definition 463
 formal_discrete_type_definition 466
 formal_floating_point_definition 466
 formal_interface_type_definition 470
 formal_modular_type_definition 466
 formal_object_declaration 458
 formal_ordinary_fixed_point_definition 466
 formal_package_actual_part 475
 formal_package_association 475
 formal_package_declaration 475
 formal_part 256
 formal_private_type_definition 462
 formal_signed_integer_type_definition 466
 formal_subprogram_declaration 470
 formal_type_declaration 460
 formal_type_definition 460
 format_effector 34
 Fortran standard 30
 freeing storage 532
 from an ancestor type 75
 from an entry queue 354
 full constant declaration 61
 full declaration 290
 full stop 36
 full type 54
 full type definition 54, 55
 full_type_declaration 53, 55
 fully conformant 265
 function 235, 255
 function call 551
 function instance 456
 function return 274
 function_call 267
 function_specification 255

G

garbage collection 534
general access type 157
general_access_modifier 156
generic actual 455
generic actual parameter 455
generic actual subtype 461
generic actual type 461
generic body 453
generic contract issue 414
generic formal 451
generic formal in object actual 458
generic formal object 458
generic formal object default_expression 458
generic formal object of mode in 460
generic formal package 474
generic formal subprogram 470
generic formal subtype 461
generic formal type 461
generic function 451
generic package 451
generic procedure 451
generic subprogram 451
generic unit 450
generic_actual_part 455
generic_association 455, 457
generic_association for a formal
 object of mode in 460
generic_declaration 450, 451
generic_formal_parameter_declaration 451
generic_formal_part 450
generic_instantiation 455, 457, 550
generic_package_declaration 279, 450
generic_renaming_declaration 323
generic_subprogram_declaration 257, 450
global to 305
goto_statement 253
govern a variant 135
govern a variant_part 135
graphic_character 35
greater than 201, 206
greater than operator 201, 206
greater than or equal 201, 206
greater than or equal operator 201, 206
greater-than sign 36
guard 378

H

handled_sequence_of_statements 420, 421
handler 423
hexadecimal literal 41
hidden from all visibility 308, 310
hidden from direct visibility 308, 311
hiding 308
High_Order_First 508
highest precedence 217
highest precedence operator 217

highest_precedence_operator 203
homograph 308
hyphen-minus 36

I

identifier 38
identifier specific to a pragma 45
identifier_extend 38
identifier_start 38
Identity 425
Identity attribute 425
if any type in a specified class of
 types is expected 326
if expected type is specific 326
if expected type is universal or class-wide 326
if_statement 245
Image 84
Image attribute 84
immediate 308, 311
immediately 305, 353
immediately enclosing 305
immediately visible 308, 311
immediately within 305
implementation advice 23
implementation defined 25
implementation permissions 22
implementation requirements 22
implicit declaration 49
implicit subtype conversion 228
implicit_dereference 179
in (membership test) 201, 206
in a use clause 315
in a with_clause 400
inactive 328
incomplete type 51, 160
incomplete view 160
incomplete_type_declaration 160, 161, 162
indefinite subtype 60, 126
independent subprogram 449
independently addressable 389
index constraint 236
index constraint with a subtype 118
index range 115
index subtype 115
index type 115
Index_Check .. 181, 183, 199, 212, 227, 230, 232, 435
index_constraint 117, 118
index_constraint discrete_range 118
index_subtype_definition 114
indexed_component 181
indexed_component expression 181
informal definition 50
informal introduction 394
informative 21
inherited component 68
inherited discriminant 68
inherited entry 68

inherited enumeration literal	70
inherited protected subprogram	68
inherited subprogram	68
initialization expression	61, 63
initialization expression of allocator	232
Initialize	295
initialized allocator	231, 232
initialized by default	63
Inline	266
Inline pragma	266
innermost dynamically enclosing	422
Input	544, 546
Input attribute	544, 546
Input clause	487, 547
instance of Unchecked_Deallocation	301
instructions for comment submission	13
Integer	96, 97
integer literal	39
integer literals	96, 98
integer type	95
integer_type_definition	95, 97
interface	152
interface_list	152
interface_type_definition	152
internal call	344
internal code	500
internal requeue	344
intertask communication	343
Intrinsic	263
Intrinsic calling convention	263
invalid representation	523
ISO 1989:2002	30
ISO 8601:2004	30
ISO/IEC 10646:2003	30, 93, 94
ISO/IEC 14882:2003	30
ISO/IEC 1539-1:2004	30
ISO/IEC 6429:1992	30
ISO/IEC 646:1991	30
ISO/IEC 8859-1:1987	30
ISO/IEC 9899:1999	30
ISO/IEC TR 19769:2004	30
iteration_scheme	249

K

known	125
known discriminants	125
known_discriminant_part	123

L

label	241
language-defined	431, 448
language-defined check	431, 448
Last	77, 120
Last attribute	77, 120
Last(N)	120
Last(N) attribute	120
Last_Bit	507
last_bit	502
Last_Bit attribute	507
Latin-1	93
layout	501
leaving	300
left	300
left parenthesis	36
legality rules	22
Length	120
Length attribute	120
Length(N)	120
Length(N) attribute	121
Length_Check	205, 224, 227, 437
less than	201, 206
less than operator	201, 206
less than or equal	201, 206
less than or equal operator	201, 206
less-than sign	36
letter_lowercase	33
letter_modifier	33
letter_other	33
letter_titlecase	33
letter_uppercase	32
lexical element	36
lexicographic order	209
library	168, 407
library level	168
Library unit	394, 395
library unit pragma	408
library unit pragmas	408
library_item	395
library_item on another	410
library_unit_body	395
library_unit_declaration	395
library_unit_renaming_declaration	395
lifetime	164
limited	152
limited interface	152
limited type	293
limited view	396
limited_with_clause	400
line	36
link-time	22, 28
List	46
List pragma	46
literal	41, 189
little	508
little endian	508
local to	305

local_name 481
 logical 205
 logical operator 205
 logical_operator 203
 Long_Float 104, 105
 Long_Integer 97, 98
 loop parameter 249
 loop_parameter_specification 249
 loop_statement 248, 249
 loop_statement with a for iteration_scheme 249
 loop_statement with a while
 iteration_scheme 249
 low line 36
 Low_Order_First 508

M

machine code insertion 518
 machine scalar 487
 Machine_Radix clause 487
 mark_non_spacing 33
 master 300
 matching components 208
 Max 78
 Max attribute 78
 Max_Base_Digits 103
 Max_Binary_Modulus 95
 Max_Digits 103
 Max_Int 96
 Max_Nonbinary_Modulus 95
 Max_Size_In_Storage_Elements 531
 Max_Size_In_Storage_Elements attribute 531
 membership test 206, 210
 membership test simple_expression 206
 mentioned in 400
 metrics 22
 Min 78
 Min attribute 78
 Min_Int 96
 minus 36, 201, 211, 213
 minus operator 201, 211, 213
 Mod 96
 mod 201, 213
 Mod attribute 96
 mod operator 201, 213
 mode 256
 mode conformance 264
 mode conformant 264
 modular type 95
 modular types 98
 modular_type_definition 95
 modular_type_definition expression 95
 Modulus 97
 Modulus attribute 97
 multi-dimensional array 115
 multiply 36, 201, 213
 multiply operator 201, 213
 multiplying 213

multiply operator 213
 multiplying_operator 203

N

n-dimensional array_aggregate 197
 name 179, 180
 name resolution rules 22
 name that has a prefix 180
 name used as a pragma argument 327
 name with a given expected profile 326
 named association 267, 455
 named component association 191
 named discriminant association 128
 named entry index 350
 named in 400
 named number 60
 named number value 66
 named type 54
 named_array_aggregate 196
 Natural 96
 needs finalization 297
 No_Dependence 537
 No_Implementation_Attributes 536
 No_Implementation_Pragmas 537
 No_Obsolescent_Features 537
 No_Return 275
 No_Return pragma 275
 nominal subtype 60, 62
 nominal subtype caused by a name 551
 non-generic subprogram_body 262
 non-returning 275
 nonexistent 533, 534
 nongeneric package_body 282
 nongraphic character 81
 nongraphic characters 94
 nonlimited 152
 nonlimited interface 152
 nonlimited type 293
 nonlimited_with_clause 400
 nonstandard 29
 nonstandard integer type 98
 nonstandard mode 29
 nonstandard real type 102
 normal 299
 normal completion 299
 normal state of an object 449, 522
 normative 21
 not 201, 218
 not equal 201, 206
 not equal operator 201, 206
 not in (membership test) 201, 206
 not operator 201, 218
 notation 25
 notes 23
 notwithstanding 409
 null 277
 null access value 189

null array 118
 null constraint 51
 null extension 143
 null literal 189
 null procedure 277
 null range 76
 null record 131
 null slice 183
 null string literal 43
 null_exclusion 156
 null_procedure_declaration 277
 null_statement 241
 number sign 36
 number_decimal 33
 number_declaration 65, 66
 number_declaration_expression 65
 number_letter 34
 numeral 40
 numeric 39, 220, 223
 numeric literal 189
 numeric type 76
 numeric_literal 40

O

object 58, 62, 272, 301, 533
 object created by an allocator 141
 object_declaration 61, 62, 550
 object_declaration_initialization_expression 61
 object_renaming_declaration 317
 occur immediately within 305
 octal literal 41
 of (a view of) an entity 50, 307
 of a call on an entry or a
 protected subprogram 343
 of a choice and an exception 420
 of a class 72
 of a compilation unit by another 410
 of a complete context 324
 of a component 116, 501, 506
 of a constituent of a complete context 325
 of a construct 27, 304
 of a decimal fixed point subtype 107, 111, 113
 of a decimal fixed point type 108
 of a declaration 306, 307
 of a delay_statement 361
 of a dimension of an array 115
 of a discrete type 101
 of a discrete_range 118
 of a first array subtype 115
 of a fixed point type 106, 107, 113
 of a floating point type 103, 104, 106
 of a formal derived type 463
 of a formal package 476
 of a formal parameter 257
 of a formal_abstract_subprogram_declaration .. 472
 of a function 272
 of a function result 257

of a generic formal object 460
 of a generic function 456
 of a generic package 456
 of a generic procedure 456
 of a generic subprogram 456
 of a generic unit 307, 325, 454
 of a goto_statement 253
 of a language-defined check 431
 of a library unit 395, 396
 of a library_item 396
 of a master 300
 of a membership test 206
 of a modular type 95, 96
 of a named access type 157
 of a named discriminant_association 128
 of a one-dimensional array 115
 of a package 279
 of a package (other than a generic
 formal package) 279
 of a positional discriminant_association 128
 of a pragma 45
 of a prefixed view 185
 of a private type 137, 143
 of a private_extension_declaration 284
 of a profile 258
 of a program unit 404
 of a protected object 340, 341, 353
 of a protected unit 339
 of a qualified_expression 230
 of a range 76
 of a record component 131
 of a record type 130, 133, 137, 143
 of a record_component_association 192
 of a representation item 482
 of a requeue_statement 344, 356
 of a return object 274
 of a scalar subtype 77
 of a scalar type 77
 of a selective_accept 379
 of a signed integer type 96
 of a storage pool 527
 of a string_literal 42
 of a subprogram 257
 of a subtype 52
 of a subunit 404
 of a tag 552
 of a task 333, 334, 353, 386
 of a task object 331
 of a task on a master 335
 of a task on another task 335
 of a task unit 330
 of a type 54, 57, 58, 75, 78, 137, 143, 284, 325
 of a type_conversion 219
 of a use_clause 314
 of a value 81, 83
 of a value of a by-reference type 261
 of a variant_part 134
 of a view of a callable entity 307

- of a view of a composite type 307
- of a with_clause 400
- of Address for stand-alone objects and
 - for program units 488
- of Alignment for first subtypes 490
- of Alignment for objects 490
- of an access type 158, 174
- of an access-to-subprogram type 157
- of an anonymous access type 157, 158
- of an array 115
- of an array type 121
- of an array_aggregate 197
- of an aspect 482
- of an aspect of
 - representation of an entity 482, 484
- of an assignment operation 242
- of an assignment_statement 242
- of an attribute_definition_clause 307
- of an elaborable construct 414
- of an entry call 354
- of an entry caller 350
- of an enumeration value 92
- of an implementation with the Standard 23
- of an integer value 96
- of an object 58, 62, 63, 167, 300, 482
- of an object or value 51
- of an operational aspect of an entity 482, 484
- of an ordinary fixed point type 107
- of Bit_Order for record types and
 - record extensions 508
- of Component_Size for array types 497
- of External_Tag for a tagged type 498
- of Input for a type 547
- of one compilation unit upon another 398
- of one execution by another 422
- of Output for a type 547
- of Read for a type 547
- of Size for first subtypes 494
- of Size for stand-alone objects 492
- of Small for fixed point types 110
- of Storage_Pool for a non-derived
 - access-to-object type 528
- of Storage_Size for a non-derived
 - access-to-object type 528
- of the execution of a construct 386
- of the execution of an Ada program 24
- of the index range of an array_aggregate 199
- of the result of a function_call 268
- of the target type of a conversion 223
- of types 51, 66
- of Write for a type 547
- on a delay_statement 361
- on a dispatching operation 145
- on an accept_statement 350
- one range in another 76
- one-dimensional array 115
- open 353
- open alternative 379

- open entry 353
- operand of concatenation 212
- operates on a type 57
- operational aspect 482
- operational item 481, 482
- operator 276
- operator precedence 203
- operator_symbol 256
- optimization 448
- Optimize 46
- Optimize pragma 46
- or 201, 205
- or else (short-circuit control form) 201, 204
- or operator 201, 205
- ordering 206
- ordering operator 206
- ordinary fixed point type 106, 107
- ordinary_fixed_point_definition 106
- other_control 35
- other_format 34
- other_private_use 35
- other_surrogate 35
- Output 543, 545
- Output attribute 543, 545
- Output clause 487, 547
- Overflow_Check 97, 202, 247, 438
- overload resolution 324, 325
- overloadable 308
- overloaded 308
- overloading rules 22, 324
- overridable 308
- override 308, 457
- overriding_indicator 312

P

- Pack 486
- Pack pragma 486
- Package 279
- package instance 456
- package_body 281
- package_body_stub 403
- package_declaration 279, 280
- package_renaming_declaration 319
- package_specification 279
- packed 486
- packing 486
- padding bits 482
- Page 46
- Page pragma 46
- parameter assigning back 272
- parameter copy back 272
- parameter default_expression 257
- parameter mode 257
- parameter passing 270, 271, 272
- parameter_and_result_profile 256
- parameter_association 267, 271
- parameter_profile 256

parameter_specification	256
parent subtype	66
parent type	66
parent_unit_name	395
partition	22, 410, 412
partition building	410
pass by copy	260
pass by reference	260
per-object constraint	131, 132
per-object expression	131
percent sign	36
plus	201, 211, 213
plus operator	201, 211, 213
plus sign	36
point	36
polymorphism	136, 145
pool element	157, 527
pool type	527
pool-specific access type	157
Pos	99
Pos attribute	99
Position	506
position	502
Position attribute	506
position number	76
positional association	267, 455
positional component association	191
positional discriminant association	128
positional_array_aggregate	196
Positive	96
possible interpretation	325
post-compilation error	22
post-compilation rules	22
potentially blocking operation	345
potentially use-visible	315
pragma	45
Pragma	44
pragma argument	45
pragma name	45
pragma Pack	486
pragma, configuration	408
pragma, identifier specific to	45
pragma, library unit	408
pragma, program unit	407
pragma, representation	481
pragma_argument_association	44
precedence of operators	203
Pred	79
Pred attribute	80
predefined	203
predefined exception	419
predefined operator	203
predefined type	54
prelaborable initialization	415
Prelaborable_Initialization	414
Prelaborable_Initialization pragma	414
Prelaborate	413, 414
Prelaborate pragma	413
preelaborated	415
prefix	179, 180
prefixed view	185
prefixed view profile	265
preserved by type conversion and parameter passing	141
prevention via accessibility rules	164
primary	201
primary that is a name	202
private declaration of a library unit	396
private extension	51, 137, 143
private library unit	396
private operations	287
private part	306
private type	51
private types and private extensions	283
private_extension_declaration	283, 286
private_type_declaration	283, 286
procedure	255
procedure instance	456
procedure_call_statement	267
procedure_or_entry_call	381
procedure_specification	255
profile	257
progenitor subtype	153
progenitor type	153
program	409, 412
program execution	409
Program unit	394
program unit pragma	407
propagate	422
proper_body	176
protected	152, 264
protected action	345
protected calling convention	264
protected declaration	337, 340
protected entry	337
protected entry_declaration	349
protected function	344
protected interface	152
protected object	328, 337
protected operation	337
protected procedure	344
protected subprogram	337, 344
protected subprogram call	345
protected tagged type	152
protected unit	337
protected_body	338, 341
protected_body_stub	403
protected_declaration}	339
protected_definition	338, 340
protected_element_declaration	338
protected_operation_declaration	338
protected_operation_item	338
protected_type_declaration	338
protection of	389
pseudo-names of anonymous types	54
public declaration of a library unit	396

public library unit 396
punctuation_connector 34
Pure 416
pure 416
Pure pragma 416

Q

qualified_expression 229, 230
quotation mark 36

R

raise_statement 421
raise_statement with an exception_name 422
raised by failure of run-time check 25, 29, 56, 79, 80, 86, 87, 88, 89, 90, 97, 100, 108, 148, 172, 177, 180, 181, 183, 186, 190, 191, 195, 199, 202, 203, 204, 205, 212, 216, 218, 219, 223, 228, 230, 232, 233, 243, 247, 261, 268, 274, 276, 301, 302, 321, 334, 341, 346, 353, 355, 380, 387, 412, 419, 426, 432, 443, 446, 465, 497, 517, 523, 528, 533, 547
Range 77, 120
range 76, 77, 236
Range attribute 78, 120
range constraint 236
range simple_expressions 77
range with a scalar subtype 77
Range(N) 120
Range(N) attribute 120
range_attribute_designator 187
range_attribute_designator expression 187
range_attribute_reference 187, 188
Range_Check 56, 79, 80, 86, 87, 88, 89, 90, 100, 108, 189, 199, 205, 218, 219, 223, 225, 226, 227, 230, 439, 546
range_constraint 76, 77
range_constraint range 76
range_constraint with a scalar subtype 77
re-raise statement 421, 422
Read 541, 543
Read attribute 541, 543
Read clause 487, 547
reading a view conversion 228
ready 328
real literal 39
real literals 102
real type 51, 102
real_range_specification 103
real_range_specification bounds 103
real_type_definition 102
reclamation of storage 532
recommended level of support 484
record 130
record extension 67, 143
record layout 501
record type 130
record_aggregate 191, 193

record_aggregate
 record_component_association_list 192
record_component_association 191
record_component_association expression 192
record_component_association_list 191, 193
record_definition 130, 131
record_extension_part 143, 144
record_representation_clause 502, 504
record_type_definition 130, 131
reference parameter passing 260
references 30
relation 201
relational 206
relational operator 206
relational_operator 203
relationship with scope 306
rem 201, 213
rem operator 201, 213
renamed entity 316
renamed view 316
renaming-as-body 319
renaming-as-declaration 319
renaming_declaration 316
rendezvous 351
representation 486
representation aspect 482
representation attribute 486
representation item 481, 482, 484
representation of an object 482
representation pragma 481
requested decimal precision 103
requeue 356
requeue protected entry 357
requeue statement 357
requeue task entry 357
requeue-with-abort 358
requeue_statement 356, 357
require overriding 150
required 124, 128, 147, 161, 171, 173, 177, 188, 220, 221, 222, 223, 262, 264, 265, 270, 273, 285, 309, 317, 320, 326, 330, 331, 339, 340, 349, 350, 356, 357, 404, 459, 464, 467, 469, 471, 476, 487
required for a task to run 329
requires a completion 177
requires late initialization 62
reserved word 47
resolution of ambiguity 324
resolution rules 22
restriction 535
restriction parameter expression 535
restriction_parameter_argument 535
Restrictions 535
Restrictions pragma 535
result of a function_call 268
result of inherited function 70
return statement 272
return_subtype_indication 272
returned by a function 141

right parenthesis 36
 root library unit 396
 root_integer 96
 root_real 102
 rooted at a type 72
 Round 113
 Round attribute 113
 run-time 22, 28, 431, 448
 run-time concept 299
 run-time error 22, 28, 431, 448
 run-time polymorphism 145
 run-time semantics 22

S

scalar subtype 236
 scalar type 51, 76
 scalar_constraint 56
 Scale 112
 Scale attribute 112
 select_alternative 378
 select_statement 377
 selected_component 183, 185
 selective_accept 378, 379
 selector_name 184
 semantic 398
 semicolon 36
 separate 394
 separate compilation 394
 separator 37
 separator_line 34
 separator_paragraph 34
 separator_space 34
 sequence_of_statements 240, 241
 short-circuit control form 204, 205
 short-circuit control form relation 204
 Short_Float 104
 Short_Integer 97
 signed integer type 95
 signed_integer_type_definition 95
 signed_integer_type_definition
 simple_expression 95
 simple 352
 simple entry call 352
 simple_expression 201
 simple_return_statement 272, 274
 simple_statement 240
 single 350
 single entry 350
 single_protected_declaration 338, 340
 single_task_declaration 329, 331
 Size 492, 493
 Size attribute 492, 493, 495
 Size clause 487, 492, 494
 slice 182, 183
 slice discrete_range 182
 Small 109
 Small attribute 109

Small clause 110, 487
 solidus 36
 special inheritance rule for tagged types ... 68, 208
 Specialized Needs 20
 Specialized Needs Annexes 20
 specifiable 487
 specifiable (of an attribute and for an entity) .. 487
 specifiable attributes 487
 specific type 72
 specific type caused by the freezing of
 the class-wide type 552
 specified (not!) 25
 specified discriminant 125
 specifying 486
 stand-alone constant 64
 stand-alone object 61
 stand-alone object,
 component, or aggregate 141
 stand-alone variable 64
 standard 29
 standard mode 29
 standard storage pool 528
 start 345
 statement 240
 statement_identifier 241
 static 234
 static semantics 22
 statically 164, 168
 statically deeper 164, 168
 statically determined tag 145
 statically tagged 146
 storage element 487
 storage place 501
 storage pool 157
 storage pool element 527
 storage pool type 527
 Storage_Check 419, 446, 497, 528
 Storage_Pool 527
 Storage_Pool attribute 527
 Storage_Pool clause 487, 528
 Storage_Size 496, 527
 Storage_Size attribute 496, 527
 Storage_Size clause 487, 528
 Storage_Size pragma 496, 497
 Storage_Size pragma argument 497
 stream 538
 stream attribute 547
 stream type 538
 Stream_Size 540
 Stream_Size attribute 540, 541
 Stream_Size clause 487
 string subtype 236
 string type 122
 string_element 42
 string_literal 42, 189
 subcomponent 51
 subprogram 255
 subprogram call 266, 268

subprogram instance	456
subprogram_body	261, 262
subprogram_body_stub	403
subprogram_declaration	255, 257, 258
subprogram_default	471
subprogram_renaming_declaration	320
subprogram_specification	255
subsystem	394
subtype	51, 52, 67, 77, 92, 96, 104, 108, 115, 125, 140, 158, 236, 460
subtype caused by a record extension	550
subtype caused by an implicit conversion	551
subtype caused by an implicit dereference	551
subtype conformance	265
subtype conformant	265
subtype_declaration	55, 56
subtype_indication	56
subtype_mark	56
subtypes of the profile of a callable entity	551
subunit	404
Succ	79
Succ attribute	79
support external streaming	548
Suppress	431
Suppress pragma	431
suppressed check	432
synchronization	328
synchronized	152
synchronized interface	152
synchronized tagged	152
synchronized tagged type	152
syntactic category	27
syntax rules	26
System	510
System.Address.To.Access.Conversions	517
System.Machine.Code	519
System.Storage.Elements	516
System.Storage.Pools	526

T

Tag	140
Tag attribute	140
tag indeterminate	146
tag of an object	137
Tag_Check	148, 225, 227, 243, 441
tagged	160
tagged incomplete view	160
tagged type	136
task	152, 328, 333
task declaration	329, 331
task interface	152
task tagged type	152
task unit	328
task_body	330, 331, 333
task_body_stub	403
task_declaration}	330
task_definition	329, 331

task_item	329
task_type_declaration	329
template	450
term	201
Term=[mentioned],Sec=[in a with_clause]	400
terminate_alternative	379
Terminated	388
terminated	328
Terminated attribute	388
termination	335
terms introduced or defined	30
text of a program	36
the value of an object	59
tick	36
time base	359
time type	359
timed_entry_call	381, 382
times	201, 213
times operator	201, 213
to a callable construct by a return statement	272
to a loop_statement by an exit_statement	252
to a program unit by a program unit pragma	407
to a range	76
to a subtype	52
transfer of control	241
triggering_alternative	384
triggering_statement	384
True	95
type	50, 152, 301
type caused by a range	551
type caused by an expression	551
type caused by the freezing of a subtype	551
type conformance	264
type conformant	264
type conversion	219, 222, 223
type conversion, array components	221
type extension	137, 143
type of	52
type of a discrete_range	117
type of a range	76
type resolution rules	326
type supports	548
type_conversion	219
type_conversion operand	219
type_declaration	53
type_definition	54

U

ultimate	75
unary	203
unary adding	213
unary adding operator	213
unary operator	203
unary_adding_operator	203
unchecked	520, 532
unchecked conversion	521
unchecked storage deallocation	532
unchecked type conversion	520
Unchecked_Access	525
Unchecked_Access attribute	525
unconstrained	52
unconstrained_array_definition	114
undefined result	448
under Syntax heading	22
underline	36
uninitialized allocator	231, 232
uninitialized variables	522
universal type	74
universal_integer	98
unknown	125
unknown discriminants	125
unknown_discriminant_part	123
unpolluted	538
unspecified	25
Unsuppress	431
Unsuppress pragma	431
usage name	50
use clause	308, 315
use in a record definition	131
use-visible	308, 315
use_clause	314, 315
use_package_clause	314
use_type_clause	314
user-defined	276, 295, 526
user-defined assignment	295
user-defined heap management	526
user-defined operator	276
user-defined storage management	526

V

Val	100
Val attribute	100
Valid	524
Valid attribute	524
Value	89, 90
value	219
Value attribute	89
value conversion	219, 223
values belonging to	52

variable	59
variable object	59
variable view	59
variant	134
variant_part	134, 136
variant_part discrete_choice	134
vertical line	36
view	50, 219
view conversion	219, 227
visibility rules	308
visible	308, 310
visible part	306

W

waiting for activations to complete	334
waiting for dependents to terminate	335
where hidden from all visibility	311
Wide_Character	94
Wide_Image	83
Wide_Image attribute	83
Wide_Value	88, 89
Wide_Value attribute	88
Wide_Wide_Character	94
Wide_Wide_Image	80
Wide_Wide_Image attribute	81
Wide_Wide_Value	86
Wide_Wide_Value attribute	86
Wide_Wide_Width	85
Wide_Wide_Width attribute	85
Wide_Width	85
Wide_Width attribute	85
Width	85
Width attribute	85
with respect to nonstatic expressions	484
with_clause	400
within a pragma in a context_clause	409
within a pragma that appears at the place of	
a compilation unit	409
within a use_clause in a context_clause	409
within a with_clause	409
within the declaration itself	310
within the parent_unit_name of	
a library unit	409
within the parent_unit_name of a subunit	409
word	487
Write	541, 542
Write attribute	541, 542
Write clause	487, 547

X

xor	201, 205
xor operator	201, 205