The CWEAVE processor

(Version 4.7 [TEX Live])

March 21, 2022 at 18:48

1*. Introduction. This is the `CWEAVE` program by Silvio Levy and Donald E. Knuth, based on `WEAVE` by Knuth. We are thankful to Steve Avery, Nelson Beebe, Hans-Hermann Bode (to whom the original C++ adaptation is due), Klaus Guntermann, Norman Ramsey, Tomas Rokicki, Joachim Schnitter, Joachim Schrod, Lee Wittenberg, Saroj Mahapatra, Cesar Augusto Rorato Crusius, and others who have contributed improvements.

The "banner line" defined here should be changed whenever `CWEAVE` is modified.

```
#define banner "This␣is␣CWEAVE,␣Version␣4.7"    ▷ will be extended by the TEX Live versionstring ◁
  ⟨ Include files 4* ⟩
  ⟨ Preprocessor definitions ⟩
  ⟨ Common code for CWEAVE and CTANGLE 3* ⟩
  ⟨ Typedef declarations 22 ⟩
  ⟨ Private variables 21 ⟩
  ⟨ Predeclaration of procedures 8* ⟩
```

2*. `CWEAVE` has a fairly straightforward outline. It operates in three phases: First it inputs the source file and stores cross-reference data, then it inputs the source once again and produces the TEX output file, finally it sorts and outputs the index.

Please read the documentation for `COMMON`, the set of routines common to `CTANGLE` and `CWEAVE`, before proceeding further.

```
  int main(int ac,    ▷ argument count ◁
      char **av)    ▷ argument values ◁
  {
    argc ← ac; argv ← av; program ← cweave; ⟨ Set initial values 24 ⟩
    common_init(); ⟨ Start TEX output 89* ⟩
    if (show_banner) cb_show_banner();    ▷ print a "banner line" ◁
    ⟨ Store all the reserved words 34 ⟩
    phase_one();    ▷ read all the user's text and store the cross-references ◁
    phase_two();    ▷ read all the text again and translate it to TEX form ◁
    phase_three();    ▷ output the cross-reference index ◁
    if (tracing ≡ fully ∧ ¬show_progress) new_line;
    return wrap_up();    ▷ and exit gracefully ◁
  }
```

3*. The next few sections contain stuff from the file `"common.w"` that must be included in both `"ctangle.w"` and `"cweave.w"`. It appears in file `"common.h"`, which is also included in `"common.w"` to propagate possible changes from this `COMMON` interface consistently.

First comes general stuff:

```
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ ≡
  typedef uint8_t eight_bits;
  typedef uint16_t sixteen_bits;
  typedef enum {
    ctangle, cweave, ctwill
  } cweb;
  extern cweb program;    ▷ CTANGLE or CWEAVE or CTWILL? ◁
  extern int phase;    ▷ which phase are we in? ◁
```

See also sections 5*, 6*, 7*, 9*, 10*, 12*, 14*, 15*, and 277*.

This code is used in section 1*.

4* You may have noticed that almost all `"strings"` in the `CWEB` sources are placed in the context of the '_' macro. This is just a shortcut for the '*gettext*' function from the "GNU gettext utilities." For systems that do not have this library installed, we wrap things for neutral behavior without internationalization. For backward compatibility with pre-ANSI compilers, we replace the "standard" header file '`stdbool.h`' with the KPATHSEA interface '`simpletypes.h`'.

```
#define _(s) gettext(s)
⟨ Include files 4* ⟩ ≡
#include <ctype.h>    ▷ definition of isalpha, isdigit and so on ◁
#include <kpathsea/simpletypes.h>    ▷ boolean, true and false ◁
#include <stddef.h>    ▷ definition of ptrdiff_t ◁
#include <stdint.h>    ▷ definition of uint8_t and uint16_t ◁
#include <stdio.h>    ▷ definition of printf and friends ◁
#include <stdlib.h>    ▷ definition of getenv and exit ◁
#include <string.h>    ▷ definition of strlen, strcmp and so on ◁
#ifndef HAVE_GETTEXT
#define HAVE_GETTEXT 0
#endif
#if HAVE_GETTEXT
#include <libintl.h>
#else
#define gettext(a) a
#endif
```

This code is used in section 1*.

5* Code related to the character set:

```
#define and_and °4    ▷ '&&'; corresponds to MIT's ∧ ◁
#define lt_lt °20    ▷ '<<'; corresponds to MIT's ⊂ ◁
#define gt_gt °21    ▷ '>>'; corresponds to MIT's ⊃ ◁
#define plus_plus °13    ▷ '++'; corresponds to MIT's ↑ ◁
#define minus_minus °1    ▷ '--'; corresponds to MIT's ↓ ◁
#define minus_gt °31    ▷ '->'; corresponds to MIT's → ◁
#define non_eq °32    ▷ '!='; corresponds to MIT's ≠ ◁
#define lt_eq °34    ▷ '<='; corresponds to MIT's ≤ ◁
#define gt_eq °35    ▷ '>='; corresponds to MIT's ≥ ◁
#define eq_eq °36    ▷ '=='; corresponds to MIT's ≡ ◁
#define or_or °37    ▷ '||'; corresponds to MIT's ∨ ◁
#define dot_dot_dot °16    ▷ '...'; corresponds to MIT's ∞ ◁
#define colon_colon °6    ▷ '::'; corresponds to MIT's ∈ ◁
#define period_ast °26    ▷ '.*'; corresponds to MIT's ⊗ ◁
#define minus_gt_ast °27    ▷ '->*'; corresponds to MIT's ⇆ ◁
#define compress(c) if (loc++ ≤ limit) return c
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern char section_text[];    ▷ text being sought for ◁
  extern char *section_text_end;    ▷ end of section_text ◁
  extern char *id_first;    ▷ where the current identifier begins in the buffer ◁
  extern char *id_loc;    ▷ just after the current identifier in the buffer ◁
```

6* Code related to input routines:

```
#define xisalpha(c) (isalpha((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xisdigit(c) (isdigit((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xisspace(c) (isspace((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xislower(c) (islower((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xisupper(c) (isupper((int)(c)) ∧ ((eight_bits)(c) < °200))
#define xisxdigit(c) (isxdigit((int)(c)) ∧ ((eight_bits)(c) < °200))
#define isxalpha(c) ((c) ≡ '_' ∨ (c) ≡ '$')     ▷ non-alpha characters allowed in identifier ◁
#define ishigh(c) ((eight_bits)(c) > °177)
```

```
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern char buffer[];     ▷ where each line of input goes ◁
  extern char *buffer_end;     ▷ end of buffer ◁
  extern char *loc;     ▷ points to the next character to be read from the buffer ◁
  extern char *limit;     ▷ points to the last character in the buffer ◁
```

7* Code related to file handling:

```
  format line x     ▷ make line an unreserved word ◁
#define max_include_depth 10
            ▷ maximum number of source files open simultaneously, not counting the change file ◁
#define max_file_name_length 1024
#define cur_file file[include_depth]     ▷ current file ◁
#define cur_file_name file_name[include_depth]     ▷ current file name ◁
#define cur_line line[include_depth]     ▷ number of current line in current file ◁
#define web_file file[0]     ▷ main source file ◁
#define web_file_name file_name[0]     ▷ main source file name ◁
```

```
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern int include_depth;     ▷ current level of nesting ◁
  extern FILE *file[];     ▷ stack of non-change files ◁
  extern FILE *change_file;     ▷ change file ◁
  extern char file_name[][max_file_name_length];     ▷ stack of non-change file names ◁
  extern char change_file_name[];     ▷ name of change file ◁
  extern char check_file_name[];     ▷ name of check_file ◁
  extern int line[];     ▷ number of current line in the stacked files ◁
  extern int change_line;     ▷ number of current line in change file ◁
  extern int change_depth;     ▷ where @y originated during a change ◁
  extern boolean input_has_ended;     ▷ if there is no more input ◁
  extern boolean changing;     ▷ if the current line is from change_file ◁
  extern boolean web_file_open;     ▷ if the web file is being read ◁
```

8*

```
⟨ Predeclaration of procedures 8* ⟩ ≡
  extern boolean get_line(void);     ▷ inputs the next line ◁
  extern void check_complete(void);     ▷ checks that all changes were picked up ◁
  extern void reset_input(void);     ▷ initialize to read the web file and change file ◁
```

See also sections 11*, 13*, 16*, 25*, 40, 45, 65, 69, 71, 83, 86, 90, 95, 98, 115*, 118, 122, 181, 189, 194, 201, 210, 214, 228, 235, 244, 248, 259, and 268.

This code is used in section 1*.

9*. Code related to section numbers:

```
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern sixteen_bits section_count;   ▷ the current section number ◁
  extern boolean changed_section[ ];   ▷ is the section changed? ◁
  extern boolean change_pending;   ▷ is a decision about change still unclear? ◁
  extern boolean print_where;   ▷ tells CTANGLE to print line and file info ◁
```

10*. Code related to identifier and section name storage:

```
#define length(c) (size_t)((c + 1)→byte_start − (c)→byte_start)   ▷ the length of a name ◁
#define print_id(c) term_write((c)→byte_start, length(c))   ▷ print identifier ◁
#define llink link   ▷ left link in binary search tree for section names ◁
#define rlink dummy.Rlink   ▷ right link in binary search tree for section names ◁
#define root name_dir→rlink   ▷ the root of the binary search tree for section names ◁
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  typedef struct name_info {
    char *byte_start;   ▷ beginning of the name in byte_mem ◁
    struct name_info *link;
    union {
      struct name_info *Rlink;   ▷ right link in binary search tree for section names ◁
      char Ilk;   ▷ used by identifiers in CWEAVE only ◁
    } dummy;
    void *equiv_or_xref;   ▷ info corresponding to names ◁
  } name_info;   ▷ contains information about an identifier or section name ◁
  typedef name_info *name_pointer;   ▷ pointer into array of name_infos ◁
  typedef name_pointer *hash_pointer;
  extern char byte_mem[ ];   ▷ characters of names ◁
  extern char *byte_mem_end;   ▷ end of byte_mem ◁
  extern char *byte_ptr;   ▷ first unused position in byte_mem ◁
  extern name_info name_dir[ ];   ▷ information about names ◁
  extern name_pointer name_dir_end;   ▷ end of name_dir ◁
  extern name_pointer name_ptr;   ▷ first unused position in name_dir ◁
  extern name_pointer hash[ ];   ▷ heads of hash lists ◁
  extern hash_pointer hash_end;   ▷ end of hash ◁
  extern hash_pointer h;   ▷ index into hash-head array ◁
```

11*.

```
⟨ Predeclaration of procedures 8* ⟩ +≡
  extern boolean names_match(name_pointer, const char *, size_t, eight_bits);
  extern name_pointer id_lookup(const char *, const char *, eight_bits);
    ▷ looks up a string in the identifier table ◁
  extern name_pointer section_lookup(char *, char *, boolean);   ▷ finds section name ◁
  extern void init_node(name_pointer);
  extern void init_p(name_pointer, eight_bits);
  extern void print_prefix_name(name_pointer);
  extern void print_section_name(name_pointer);
  extern void sprint_section_name(char *, name_pointer);
```

12* Code related to error handling:

```
#define spotless 0   ▷ history value for normal jobs ◁
#define harmless_message 1   ▷ history value when non-serious info was printed ◁
#define error_message 2   ▷ history value when an error was noted ◁
#define fatal_message 3   ▷ history value when we had to stop prematurely ◁
#define mark_harmless if (history ≡ spotless) history ← harmless_message
#define mark_error history ← error_message
#define confusion(s) fatal(_("!␣This␣can't␣happen:␣"), s)
```

```
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern int history;   ▷ indicates how bad this run was ◁
```

13* ⟨ Predeclaration of procedures 8* ⟩ +≡

```
  extern int wrap_up(void);   ▷ indicate history and exit ◁
  extern void err_print(const char *);   ▷ print error message and context ◁
  extern void fatal(const char *, const char *);   ▷ issue error message and die ◁
  extern void overflow(const char *);   ▷ succumb because a table has overflowed ◁
```

14* Code related to command line arguments:

```
#define show_banner flags['b']   ▷ should the banner line be printed? ◁
#define show_progress flags['p']   ▷ should progress reports be printed? ◁
#define show_happiness flags['h']   ▷ should lack of errors be announced? ◁
#define show_stats flags['s']   ▷ should statistics be printed at end of run? ◁
#define make_xrefs flags['x']   ▷ should cross references be output? ◁
#define check_for_change flags['c']   ▷ check temporary output for changes ◁
```

```
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern int argc;   ▷ copy of ac parameter to main ◁
  extern char **argv;   ▷ copy of av parameter to main ◁
  extern char C_file_name[];   ▷ name of C_file ◁
  extern char tex_file_name[];   ▷ name of tex_file ◁
  extern char idx_file_name[];   ▷ name of idx_file ◁
  extern char scn_file_name[];   ▷ name of scn_file ◁
  extern boolean flags[];   ▷ an option for each 7-bit code ◁
  extern const char *use_language;   ▷ prefix to cwebmac.tex in TEX output ◁
```

15* Code related to output:

```
#define update_terminal fflush(stdout)   ▷ empty the terminal output buffer ◁
#define new_line putchar('\n')
#define term_write(a, b) fflush(stdout), fwrite(a, sizeof(char), b, stdout)
```

```
⟨ Common code for CWEAVE and CTANGLE 3* ⟩ +≡
  extern FILE *C_file;   ▷ where output of CTANGLE goes ◁
  extern FILE *tex_file;   ▷ where output of CWEAVE goes ◁
  extern FILE *idx_file;   ▷ where index from CWEAVE goes ◁
  extern FILE *scn_file;   ▷ where list of sections from CWEAVE goes ◁
  extern FILE *active_file;   ▷ currently active file for CWEAVE output ◁
  extern FILE *check_file;   ▷ temporary output file ◁
```

16* The procedure that gets everything rolling:

```
⟨ Predeclaration of procedures 8* ⟩ +≡
  extern void common_init(void);
  extern void print_stats(void);
  extern void cb_show_banner(void);
```

17* The following parameters are sufficient to handle TEX (converted to `CWEB`), so they should be sufficient for most applications of `CWEB`.

```
#define buf_size 1000    ▷ maximum length of input line, plus one ◁
#define longest_name 10000    ▷ file names, section names, and section texts shouldn't be longer than this ◁
#define long_buf_size (buf_size + longest_name)    ▷ for CWEAVE ◁
#define max_bytes 1000000
          ▷ the number of bytes in identifiers, index entries, and section names; must be less than 2^24 ◁
#define max_names 10239    ▷ number of identifiers, strings, section names; must be less than 10240 ◁
#define max_sections 4000    ▷ greater than the total number of sections ◁
```

18* End of `COMMON` interface.

19* The following parameters are sufficient to handle TEX (converted to `CWEB`), so they should be sufficient for most applications of `CWEAVE`.

```
#define line_length 80    ▷ lines of TEX output have at most this many characters; should be less than 256 ◁
#define max_refs 65535    ▷ number of cross-references; must be less than 65536 ◁
#define max_scraps 5000    ▷ number of tokens in C texts being parsed ◁
```

25* A new cross-reference for an identifier is formed by calling *new_xref*, which discards duplicate entries and ignores non-underlined references to one-letter identifiers or C's reserved words.

If the user has sent the *no_xref* flag (the `-x` option of the command line), it is unnecessary to keep track of cross-references for identifiers. If one were careful, one could probably make more changes around section 115 to avoid a lot of identifier looking up.

```
#define append_xref(c)
          if (xref_ptr ≡ xmem_end) overflow(_("cross-reference"));
          else (++xref_ptr)→num ← c
#define no_xref ¬make_xrefs
#define is_tiny(p) length(p) ≡ 1
#define unindexed(a) ((a) < res_wd_end ∧ (a)→ilk ≥ custom)
          ▷ tells if uses of a name are to be indexed ◁

⟨ Predeclaration of procedures 8* ⟩ +≡
  static void new_xref(name_pointer);
  static void new_section_xref(name_pointer);
  static void set_file_flag(name_pointer);
```

30* The first position of *tok_mem* that is unoccupied by replacement text is called *tok_ptr*, and the first unused location of *tok_start* is called *text_ptr*. Thus, we usually have ∗*text_ptr* ≡ *tok_ptr*.

```
#define max_toks 65535    ▷ number of symbols in C texts being parsed; must be less than 65536 ◁
#define max_texts 10239    ▷ number of phrases in C texts being parsed; must be less than 10240 ◁

⟨ Private variables 21 ⟩ +≡
  static token tok_mem[max_toks];    ▷ tokens ◁
  static token_pointer tok_mem_end ← tok_mem + max_toks − 1;    ▷ end of tok_mem ◁
  static token_pointer tok_ptr;    ▷ first unused position in tok_mem ◁
  static token_pointer max_tok_ptr;    ▷ largest value of tok_ptr ◁
  static token_pointer tok_start[max_texts];    ▷ directory into tok_mem ◁
  static text_pointer tok_start_end ← tok_start + max_texts − 1;    ▷ end of tok_start ◁
  static text_pointer text_ptr;    ▷ first unused position in tok_start ◁
  static text_pointer max_text_ptr;    ▷ largest value of text_ptr ◁
```

57* C strings and character constants, delimited by double and single quotes, respectively, can contain newlines or instances of their own delimiters if they are protected by a backslash. We follow this convention, but do not allow the string to be longer than *longest_name*.

```
⟨ Get a string 57* ⟩ ≡
  { char delim ← c;      ▷ what started the string ◁
    id_first ← section_text + 1;  id_loc ← section_text;
    if (delim ≡ '\'' ∧ *(loc − 2) ≡ '@') {
      *++id_loc ← '@';  *++id_loc ← '@';
    }
    *++id_loc ← delim;
    if (delim ≡ 'L' ∨ delim ≡ 'u' ∨ delim ≡ 'U') ⟨ Get a wide character constant 58 ⟩
    if (delim ≡ '<') delim ← '>';      ▷ for file names in #include lines ◁
    while (true) {
      if (loc ≥ limit) {
        if (*(limit − 1) ≠ '\\') {
          err_print(_("!␣String␣didn't␣end"));  loc ← limit;  break;
        }
        if (get_line() ≡ false) {
          err_print(_("!␣Input␣ended␣in␣middle␣of␣string"));  loc ← buffer;  break;
        }
      }
      if ((c ← *loc++) ≡ delim) {
        if (++id_loc ≤ section_text_end) *id_loc ← c;
        break;
      }
      if (c ≡ '\\') {
        if (loc ≥ limit) continue;
        else {
          if (++id_loc ≤ section_text_end) {
            *id_loc ← '\\';  c ← *loc++;
          }
        }
      }
      if (++id_loc ≤ section_text_end) *id_loc ← c;
    }
    if (id_loc ≥ section_text_end) {
      fputs(_("\n!␣String␣too␣long:␣"), stdout);  term_write(section_text + 1, 25);  printf("...");
      mark_error;
    }
    id_loc++;  return string;
  }
```

This code is used in sections 44 and 59*.

59* After an `@` sign has been scanned, the next character tells us whether there is more work to do.

```
⟨ Get control code and possible section name 59* ⟩ ≡
  switch (ccode[c ← *loc++]) {
  case translit_code: err_print(_("!␣Use␣@l␣in␣limbo␣only")); continue;
  case underline: xref_switch ← def_flag; continue;
  case trace: tracing ← c − '0'; continue;
  case section_name: ⟨ Scan the section name and make cur_section point to it 60 ⟩
  case verbatim: ⟨ Scan a verbatim string 66* ⟩
  case ord: ⟨ Get a string 57* ⟩
  case xref_roman: case xref_wildcard: case xref_typewriter: case noop: case TEX_string:
    skip_restricted( ); /*␣fall␣through␣*/
  default: return ccode[c];
  }
```

This code is used in section 44.

62* ⟨ Put section name into *section_text* 62* ⟩ ≡

```
  while (true) {
    if (loc > limit ∧ get_line( ) ≡ false) {
      err_print(_("!␣Input␣ended␣in␣section␣name")); loc ← buffer + 1; break;
    }
    c ← *loc; ⟨ If end of name or erroneous control code, break 63* ⟩
    loc++;
    if (k < section_text_end) k++;
    if (xisspace(c)) {
      c ← '␣';
      if (*(k − 1) ≡ '␣') k−−;
    }
    *k ← c;
  }
  if (k ≥ section_text_end) {
    fputs(_("\n!␣Section␣name␣too␣long:␣"), stdout); term_write(section_text + 1, 25); printf("...");
    mark_harmless;
  }
  if (*k ≡ '␣' ∧ k > section_text) k−−;
```

This code is used in section 60.

63* ⟨ If end of name or erroneous control code, **break** 63* ⟩ ≡

```
  if (c ≡ '@') {
    c ← *(loc + 1);
    if (c ≡ '>') {
      loc += 2; break;
    }
    if (ccode[c] ≡ new_section) {
      err_print(_("!␣Section␣name␣didn't␣end")); break;
    }
    if (c ≠ '@') {
      err_print(_("!␣Control␣codes␣are␣forbidden␣in␣section␣name")); break;
    }
    *(++k) ← '@'; loc++;      ▷ now c ≡ *loc again ◁
  }
```

This code is used in section 62*.

64* This function skips over a restricted context at relatively high speed.

```
static void skip_restricted(void)
{
   id_first ← loc;  ∗(limit + 1) ← '@';
false_alarm:
   while (∗loc ≠ '@') loc++;
   id_loc ← loc;
   if (loc++ > limit) {
      err_print(_("!␣Control␣text␣didn't␣end")); loc ← limit;
   }
   else {
      if (∗loc ≡ '@' ∧ loc ≤ limit) {
         loc++; goto false_alarm;
      }
      if (∗loc++ ≠ '>') err_print(_("!␣Control␣codes␣are␣forbidden␣in␣control␣text"));
   }
}
```

66* At the present point in the program we have $*(loc-1) \equiv verbatim$; we set *id_first* to the beginning of the string itself, and *id_loc* to its ending-plus-one location in the buffer. We also set *loc* to the position just after the ending delimiter.

⟨ Scan a verbatim string 66* ⟩ ≡

```
   id_first ← loc++;  ∗(limit + 1) ← '@';  ∗(limit + 2) ← '>';
   while (∗loc ≠ '@' ∨ ∗(loc + 1) ≠ '>') loc++;
   if (loc ≥ limit) err_print(_("!␣Verbatim␣string␣didn't␣end"));
   id_loc ← loc;  loc += 2;  return verbatim;
```

This code is used in section 59*.

70* ⟨ Store cross-reference data for the current section 70* ⟩ ≡

```
{
   if (++section_count ≡ max_sections) overflow(_("section␣number"));
   changed_section[section_count] ← changing;      ▷ it will become true if any line changes ◁
   if (∗(loc − 1) ≡ '∗' ∧ show_progress) {
      printf("∗%d", (int) section_count); update_terminal;      ▷ print a progress report ◁
   }
   ⟨ Store cross-references in the TEX part of a section 74∗ ⟩
   ⟨ Store cross-references in the definition part of a section 77 ⟩
   ⟨ Store cross-references in the C part of a section 80 ⟩
   if (changed_section[section_count]) change_exists ← true;
}
```

This code is used in section 68.

74*. In the TEX part of a section, cross-reference entries are made only for the identifiers in C texts enclosed in `| ... |`, or for control texts enclosed in `@^ ... @>` or `@. ... @>` or `@: ... @>`.

```
⟨ Store cross-references in the TEX part of a section 74* ⟩ ≡
  while (true) {
    switch (next_control ← skip_TEX()) {
    case translit_code: err_print(_("!␣Use␣@l␣in␣limbo␣only")); continue;
    case underline: xref_switch ← def_flag; continue;
    case trace: tracing ← *(loc − 1) − '0'; continue;
    case '|': C_xref(section_name); break;
    case xref_roman: case xref_wildcard: case xref_typewriter: case noop: case section_name:
      loc −= 2; next_control ← get_next();     ▷ scan to @> ◁
      if (next_control ≥ xref_roman ∧ next_control ≤ xref_typewriter) {
        ⟨ Replace '@@' by '@' 75 ⟩
        new_xref(id_lookup(id_first, id_loc, next_control − identifier));
      }
      break;
    }
    if (next_control ≥ format_code) break;
  }
```

This code is used in section 70*.

79*. A much simpler processing of format definitions occurs when the definition is found in limbo.

```
⟨ Process simple format in limbo 79* ⟩ ≡
  if (get_next() ≠ identifier) err_print(_("!␣Missing␣left␣identifier␣of␣@s"));
  else {
    lhs ← id_lookup(id_first, id_loc, normal);
    if (get_next() ≠ identifier) err_print(_("!␣Missing␣right␣identifier␣of␣@s"));
    else {
      rhs ← id_lookup(id_first, id_loc, normal); lhs→ilk ← rhs→ilk;
    }
  }
```

This code is used in section 41.

82*. The following recursive procedure walks through the tree of section names and prints out anomalies.

```
  static void section_check(name_pointer p)     ▷ print anomalies in subtree p ◁
  {
    if (p) {
      section_check(p→llink); cur_xref ← (xref_pointer) p→xref;
      if ((an_output ← (cur_xref→num ≡ file_flag)) ≡ true) cur_xref ← cur_xref→xlink;
      if (cur_xref→num < def_flag) {
        fputs(_("\n!␣Never␣defined:␣<"), stdout); print_section_name(p); putchar('>');
        mark_harmless;
      }
      while (cur_xref→num ≥ cite_flag) cur_xref ← cur_xref→xlink;
      if (cur_xref ≡ xmem ∧ ¬an_output) {
        fputs(_("\n!␣Never␣used:␣<"), stdout); print_section_name(p); putchar('>'); mark_harmless;
      }
      section_check(p→rlink);
    }
  }
```

89* In particular, the *finish_line* procedure is called near the very beginning of phase two. We initialize the output variables in a slightly tricky way so that the first line of the output file will be dependent of the user language set by the '`+l`' option and its argument. If you call `CWEAVE` with '`+lX`' (or '`-lX`' as well), where '`X`' is the (possibly empty) string of characters to the right of '`l`', '`X`' will be prepended to '`cwebmac.tex`', e.g., if you call `CWEAVE` with '`+ldeutsch`', you will receive the line '`\input deutschcwebmac`'. Without this option the first line of the output file will be '`\input cwebmac`'.

```
⟨ Start TEX output 89* ⟩ ≡
  out_ptr ← out_buf + 1;  out_line ← 1;  active_file ← tex_file;  tex_puts("\\input␣");
  tex_printf(use_language);  tex_puts("cwebma");  ∗out_ptr ← 'c';
```

This code is used in section 2*.

94* We get to this section only in the unusual case that the entire output line consists of a string of backslashes followed by a string of nonblank non-backslashes. In such cases it is almost always safe to break the line by putting a `'%'` just before the last character.

```
⟨ Print warning message, break the line, return 94* ⟩ ≡
  {
    printf(_("\n!␣Line␣had␣to␣be␣broken␣(output␣l.␣%d):\n"), out_line);
    term_write(out_buf + 1, out_ptr − out_buf − 1);  new_line;  mark_harmless;
    flush_buffer(out_ptr − 1, true, true);  return;
  }
```

This code is used in section 93.

99*

```
static void copy_limbo(void)
  {
    while (true) {
      if (loc > limit ∧ (finish_line( ), get_line( ) ≡ false)) return;
      ∗(limit + 1) ← '@';
      while (∗loc ≠ '@') out(∗(loc++));
      if (loc++ ≤ limit) {
        switch (ccode[(eight_bits) ∗loc++]) {
        case new_section: return;
        case translit_code: out_str("\\ATL"); break;
        case '@': out('@'); break;
        case noop: skip_restricted( ); break;
        case format_code:
          if (get_next( ) ≡ identifier) get_next( );
          if (loc ≥ limit) get_line( );      ▷ avoid blank lines in output ◁
          break;      ▷ the operands of @s are ignored on this pass ◁
        default: err_print(_("!␣Double␣@␣should␣be␣used␣in␣limbo")); out('@');
        }
      }
    }
  }
```

101*. The *copy_comment* function issues a warning if more braces are opened than closed, and in the case of a more serious error it supplies enough braces to keep TEX from complaining about unbalanced braces. Instead of copying the TEX material into the output buffer, this function copies it into the token memory (in phase two only). The abbreviation *app_tok*(*t*) is used to append token *t* to the current token list, and it also makes sure that it is possible to append at least one further token without overflow.

```
#define app_tok(c)
        {
          if (tok_ptr + 2 > tok_mem_end) overflow(_("token"));
          *(tok_ptr++) ← c;
        }

  static int copy_comment(      ▷ copies TEX code in comments ◁
      boolean is_long_comment,      ▷ is this a traditional C comment? ◁
      int bal)      ▷ brace balance ◁
  {
    char c;      ▷ current character being copied ◁
    while (true) {
      if (loc > limit) {
        if (is_long_comment) {
          if (get_line() ≡ false) {
            err_print(_("!␣Input␣ended␣in␣mid-comment")); loc ← buffer + 1; goto done;
          }
        }
        else {
          if (bal > 1) err_print(_("!␣Missing␣}␣in␣comment"));
          goto done;
        }
      }
      c ← *(loc++);
      if (c ≡ '|') return bal;
      if (is_long_comment) ⟨Check for end of comment 102*⟩
      if (phase ≡ 2) {
        if (ishigh(c)) app_tok(quoted_char);
        app_tok(c);
      }
      ⟨Copy special things when c ≡ '@', '\\' 103*⟩
      if (c ≡ '{') bal++;
      else if (c ≡ '}') {
        if (bal > 1) bal--;
        else {
          err_print(_("!␣Extra␣}␣in␣comment"));
          if (phase ≡ 2) tok_ptr--;
        }
      }
    }
  done: ⟨Clear bal and return 104⟩
  }
```

102* ⟨ Check for end of comment 102* ⟩ ≡

```
if (c ≡ '*' ∧ *loc ≡ '/') {
  loc++;
  if (bal > 1) err_print(_("!␣Missing␣}␣in␣comment"));
  goto done;
}
```

This code is used in section 101*.

103* ⟨ Copy special things when c ≡ '@', '\\' 103* ⟩ ≡

```
if (c ≡ '@') {
  if (*(loc++) ≠ '@') {
    err_print(_("!␣Illegal␣use␣of␣@␣in␣comment")); loc −= 2;
    if (phase ≡ 2) *(tok_ptr − 1) ← '␣';
    goto done;
  }
}
else {
  if (c ≡ '\\' ∧ *loc ≠ '@') {
    if (phase ≡ 2) app_tok(*(loc++));
    else loc++;
  }
}
```

This code is used in section 101*.

110.* The raw input is converted into scraps according to the following table, which gives category codes followed by the translations. The symbol '`**`' stands for '`\&{`identifier`}`', i.e., the identifier itself treated as a reserved word. The right-hand column is the so-called *mathness*, which is explained further below.

An identifier *c* of length 1 is translated as `\|c` instead of as `\\{c}`. An identifier `CAPS` in all caps is translated as `\.{CAPS}` instead of as `\\{CAPS}`. An identifier that has become a reserved word via **typedef** is translated with `\&` replacing `\\` and *raw_int* replacing *exp*.

A string of length greater than 20 is broken into pieces of size at most 20 with discretionary breaks in between.

`!=`	*binop*: `\I`	yes
`<=`	*binop*: `\Z`	yes
`>=`	*binop*: `\G`	yes
`==`	*binop*: `\E`	yes
`&&`	*binop*: `\W`	yes
`||`	*binop*: `\V`	yes
`++`	*unop*: `\PP`	yes
`--`	*unop*: `\MM`	yes
`->`	*binop*: `\MG`	yes
`>>`	*binop*: `\GG`	yes
`<<`	*binop*: `\LL`	yes
`::`	*colcol*: `\DC`	maybe
`.*`	*binop*: `\PA`	yes
`->*`	*binop*: `\MGA`	yes
`...`	*raw_int*: `\,\ldots\,`	yes
"string"	*exp*: `\.{`string with special characters quoted`}`	maybe
`@=`string`@>`	*exp*: `\vb{`string with special characters quoted`}`	maybe
`@'7'`	*exp*: `\.{@'7'}`	maybe
`077` or `\77`	*exp*: `\T{\~77}`	maybe
`0x7f`	*exp*: `\T{\^7f}`	maybe
`0b10111`	*exp*: `\T{\\10111}`	maybe
`77`	*exp*: `\T{77}`	maybe
`77L`	*exp*: `\T{77\$L}`	maybe
`0.1E5`	*exp*: `\T{0.1\_5}`	maybe
`0x10p3`	*exp*: `\T{\^10}\p{3}`	maybe
`1'000'000`	*exp*: `\T{1\␣000\␣000}`	maybe
`+`	*ubinop*: `+`	yes
`-`	*ubinop*: `-`	yes
`*`	*raw_ubin*: `*`	yes
`/`	*binop*: `/`	yes
`<`	*prelangle*: `\langle`	yes
`=`	*binop*: `\K`	yes
`>`	*prerangle*: `\rangle`	yes
`.`	*binop*: `.`	yes
`|`	*binop*: `\OR`	yes
`^`	*binop*: `\XOR`	yes
`%`	*binop*: `\MOD`	yes
`?`	*question*: `\?`	yes
`!`	*unop*: `\R`	yes
`~`	*unop*: `\CM`	yes
`&`	*raw_ubin*: `\AND`	yes
`(`	*lpar*: `(`	maybe
`)`	*rpar*: `)`	maybe
`[`	*lbrack*: `[`	maybe

`]`	*rbrack*: `]`	maybe
`{`	*lbrace*: `{`	yes
`}`	*lbrace*: `}`	yes
`,`	*comma*: `,`	yes
`;`	*semi*: `;`	maybe
`:`	*colon*: `:`	no
`#` (within line)	*ubinop*: `\#`	yes
`#` (at beginning)	*lproc*: *force* *preproc_line* `\#`	no
end of `#` line	*rproc*: *force*	no
identifier	*exp*: `\\{`identifier with underlines and dollar signs quoted`}`	maybe
`alignas`	*alignas_like*: `**`	maybe
`alignof`	*sizeof_like*: `**`	maybe
`and`	*alfop*: `**`	yes
`and_eq`	*alfop*: `**`	yes
`asm`	*sizeof_like*: `**`	maybe
`auto`	*int_like*: `**`	maybe
`bitand`	*alfop*: `**`	yes
`bitor`	*alfop*: `**`	yes
`bool`	*raw_int*: `**`	maybe
`break`	*case_like*: `**`	maybe
`case`	*case_like*: `**`	maybe
`catch`	*catch_like*: `**`	maybe
`char`	*raw_int*: `**`	maybe
`char8_t`	*raw_int*: `**`	maybe
`char16_t`	*raw_int*: `**`	maybe
`char32_t`	*raw_int*: `**`	maybe
`class`	*struct_like*: `**`	maybe
`clock_t`	*raw_int*: `**`	maybe
`compl`	*alfop*: `**`	yes
`complex`	*int_like*: `**`	yes
`concept`	*int_like*: `**`	maybe
`const`	*const_like*: `**`	maybe
`consteval`	*const_like*: `**`	maybe
`constexpr`	*const_like*: `**`	maybe
`constinit`	*const_like*: `**`	maybe
`const_cast`	*raw_int*: `**`	maybe
`continue`	*case_like*: `**`	maybe
`co_await`	*case_like*: `**`	maybe
`co_return`	*case_like*: `**`	maybe
`co_yield`	*case_like*: `**`	maybe
`decltype`	*sizeof_like*: `**`	maybe
`default`	*default_like*: `**`	maybe
`define`	*define_like*: `**`	maybe
`defined`	*sizeof_like*: `**`	maybe
`delete`	*delete_like*: `**`	maybe
`div_t`	*raw_int*: `**`	maybe
`do`	*do_like*: `**`	maybe
`double`	*raw_int*: `**`	maybe
`dynamic_cast`	*raw_int*: `**`	maybe
`elif`	*if_like*: `**`	maybe
`else`	*else_like*: `**`	maybe
`endif`	*if_like*: `**`	maybe

enum	*struct_like*: `**`	maybe
error	*if_like*: `**`	maybe
explicit	*int_like*: `**`	maybe
export	*int_like*: `**`	maybe
extern	*int_like*: `**`	maybe
FILE	*raw_int*: `**`	maybe
false	*normal*: `**`	maybe
float	*raw_int*: `**`	maybe
for	*for_like*: `**`	maybe
fpos_t	*raw_int*: `**`	maybe
friend	*int_like*: `**`	maybe
goto	*case_like*: `**`	maybe
if	*if_like*: `**`	maybe
ifdef	*if_like*: `**`	maybe
ifndef	*if_like*: `**`	maybe
imaginary	*int_like*: `**`	maybe
include	*if_like*: `**`	maybe
inline	*int_like*: `**`	maybe
int	*raw_int*: `**`	maybe
jmp_buf	*raw_int*: `**`	maybe
ldiv_t	*raw_int*: `**`	maybe
line	*if_like*: `**`	maybe
long	*raw_int*: `**`	maybe
make_pair	*ftemplate*: `\\{make\_pair}`	maybe
mutable	*int_like*: `**`	maybe
namespace	*struct_like*: `**`	maybe
new	*new_like*: `**`	maybe
noexcept	*attr*: `**`	maybe
not	*alfop*: `**`	yes
not_eq	*alfop*: `**`	yes
NULL	*exp*: `\NULL`	yes
nullptr	*exp*: `\NULL`	yes
offsetof	*raw_int*: `**`	maybe
operator	*operator_like*: `**`	maybe
or	*alfop*: `**`	yes
or_eq	*alfop*: `**`	yes
pragma	*if_like*: `**`	maybe
private	*public_like*: `**`	maybe
protected	*public_like*: `**`	maybe
ptrdiff_t	*raw_int*: `**`	maybe
public	*public_like*: `**`	maybe
register	*int_like*: `**`	maybe
reinterpret_cast	*raw_int*: `**`	maybe
requires	*int_like*: `**`	maybe
restrict	*int_like*: `**`	maybe
return	*case_like*: `**`	maybe
short	*raw_int*: `**`	maybe
sig_atomic_t	*raw_int*: `**`	maybe
signed	*raw_int*: `**`	maybe
size_t	*raw_int*: `**`	maybe
sizeof	*sizeof_like*: `**`	maybe
static	*int_like*: `**`	maybe

`static_assert`	*sizeof_like*: `**`	maybe
`static_cast`	*raw_int*: `**`	maybe
`struct`	*struct_like*: `**`	maybe
`switch`	*for_like*: `**`	maybe
`template`	*template_like*: `**`	maybe
`TeX`	*exp*: `\TeX`	yes
`this`	*exp*: `\this`	yes
`thread_local`	*raw_int*: `**`	maybe
`throw`	*case_like*: `**`	maybe
`time_t`	*raw_int*: `**`	maybe
`try`	*else_like*: `**`	maybe
`typedef`	*typedef_like*: `**`	maybe
`typeid`	*sizeof_like*: `**`	maybe
`typename`	*struct_like*: `**`	maybe
`undef`	*if_like*: `**`	maybe
`union`	*struct_like*: `**`	maybe
`unsigned`	*raw_int*: `**`	maybe
`using`	*using_like*: `**`	maybe
`va_dcl`	*decl*: `**`	maybe
`va_list`	*raw_int*: `**`	maybe
`virtual`	*int_like*: `**`	maybe
`void`	*raw_int*: `**`	maybe
`volatile`	*const_like*: `**`	maybe
`wchar_t`	*raw_int*: `**`	maybe
`while`	*for_like*: `**`	maybe
`xor`	*alfop*: `**`	yes
`xor_eq`	*alfop*: `**`	yes
`@,`	*insert*: `\,`	maybe
`@|`	*insert*: *opt* `0`	maybe
`@/`	*insert*: *force*	no
`@#`	*insert*: *big_force*	no
`@+`	*insert*: *big_cancel* `{}` *break_space* `{}` *big_cancel*	no
`@;`	*semi*:	maybe
`@[`	*begin_arg*:	maybe
`@]`	*end_arg*:	maybe
`@&`	*insert*: `\J`	maybe
`@h`	*insert*: *force* `\ATH` *force*	no
`@<` section name `@>`	*section_scrap*: `\X`*n*`:`translated section name`\X`	maybe
`@(` section name `@>`	*section_scrap*: `\X`*n*`:\.{`section name with special characters quoted␣`}\X`	maybe
`/*` comment `*/`	*insert*: *cancel* `\C{`translated comment`}` *force*	no
`//` comment	*insert*: *cancel* `\SHC{`translated comment`}` *force*	no

The construction `@t` stuff `@>` contributes `\hbox{` stuff `}` to the following scrap.

111* Here is a table of all the productions. Each production that combines two or more consecutive scraps implicitly inserts a `$` where necessary, that is, between scraps whose abutting boundaries have different *mathness*. In this way we never get double `$$`.

A translation is provided when the resulting scrap is not merely a juxtaposition of the scraps it comes from. An asterisk* next to a scrap means that its first identifier gets an underlined entry in the index, via the function *make_underlined*. Two asterisks** means that both *make_underlined* and *make_reserved* are called; that is, the identifier's ilk becomes *raw_int*. A dagger † before the production number refers to the notes at the end of this section, which deal with various exceptional cases.

We use *in*, *out*, *back*, *bsp*, and *din* as shorthands for *indent*, *outdent*, *backup*, *break_space*, and *dindent*, respectively.

	LHS	→ RHS	Translation	Example
0	$\left\{\begin{matrix}\mathit{any}\\ \mathit{any}\ \mathit{any}\\ \mathit{any}\ \mathit{any}\ \mathit{any}\end{matrix}\right\}\mathit{insert}$	$\to\left\{\begin{matrix}\mathit{any}\\ \mathit{any}\ \mathit{any}\\ \mathit{any}\ \mathit{any}\ \mathit{any}\end{matrix}\right\}$		stmt; ▷comment◁
†1	$\mathit{exp}\left\{\begin{matrix}\mathit{lbrace}\\ \mathit{int\_like}\\ \mathit{decl}\end{matrix}\right\}$	$\to\mathit{fn\_decl}\left\{\begin{matrix}\mathit{lbrace}\\ \mathit{int\_like}\\ \mathit{decl}\end{matrix}\right\}$	$F=E^*\ \mathit{din}$	*main*( ) { *main*(*ac*, *av*) **int** *ac*;
2	*exp unop*	→ *exp*		$x{+}{+}$
3	$\mathit{exp}\left\{\begin{matrix}\mathit{binop}\\ \mathit{ubinop}\end{matrix}\right\}\mathit{exp}$	→ *exp*		x/y $x+y$
4	*exp comma exp*	→ *exp*	$E_1C\ \mathit{opt}9\ E_2$	$f(x,y)$

5	$\mathit{exp}\left\{\begin{matrix}\mathit{lpar}\ \mathit{rpar}\\ \mathit{cast}\end{matrix}\right\}\mathit{colon}$	$\to\mathit{exp}\left\{\begin{matrix}\mathit{lpar}\ \mathit{rpar}\\ \mathit{cast}\end{matrix}\right\}\mathit{base}$		**C**( ) :
				C(**int** *i*) :

6	*exp semi*	→ *stmt*		$x=0$;
7	*exp colon*	→ *tag*	E^*C	*found*:
8	*exp rbrace*	→ *stmt rbrace*		end of **enum** list
9	$\mathit{exp}\left\{\begin{matrix}\mathit{lpar}\ \mathit{rpar}\\ \mathit{cast}\end{matrix}\right\}\left\{\begin{matrix}\mathit{const\_like}\\ \mathit{case\_like}\end{matrix}\right\}$	$\to\mathit{exp}\left\{\begin{matrix}\mathit{lpar}\ \mathit{rpar}\\ \mathit{cast}\end{matrix}\right\}$	$\left\{\begin{matrix}R=R_\sqcup C\\ C_1=C_{1\sqcup}C_2\end{matrix}\right\}$	*f*( ) **const** *f*(**int**) **throw**
10	$\mathit{exp}\left\{\begin{matrix}\mathit{exp}\\ \mathit{cast}\end{matrix}\right\}$	→ *exp*		*time*( )
11	$\mathit{lpar}\left\{\begin{matrix}\mathit{exp}\\ \mathit{ubinop}\end{matrix}\right\}\mathit{rpar}$	→ *exp*		(x) $(*)$
12	*lpar rpar*	→ *exp*	$L\backslash,R$	functions, declarations
13	$\mathit{lpar}\left\{\begin{matrix}\mathit{decl\_head}\\ \mathit{int\_like}\\ \mathit{cast}\end{matrix}\right\}\mathit{rpar}$	→ *cast*		(**char** ∗)
14	$\mathit{lpar}\left\{\begin{matrix}\mathit{decl\_head}\\ \mathit{int\_like}\\ \mathit{exp}\end{matrix}\right\}\mathit{comma}$	→ *lpar*	$L\left\{\begin{matrix}D\\ I\\ E\end{matrix}\right\}C\ \mathit{opt}9$	(**int**,
15	$\mathit{lpar}\left\{\begin{matrix}\mathit{stmt}\\ \mathit{decl}\end{matrix}\right\}$	→ *lpar*	$\left\{\begin{matrix}LS_\sqcup\\ LD_\sqcup\end{matrix}\right\}$	$(k=5;$ (**int** $k=5;$
16	$\mathit{unop}\left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	→ *exp*		$\neg x$ ~**C**
17	*ubinop cast rpar*	→ *cast rpar*	$C=\{U\}C$	∗**CPtr**)
18	$\mathit{ubinop}\left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	$\to\left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	$\{U\}\left\{\begin{matrix}E\\ I\end{matrix}\right\}$	$*x$ ∗**CPtr**
19	*ubinop binop*	→ *binop*	$\mathit{math\_rel}\ U\{B\}\}$	$*=$
20	*binop binop*	→ *binop*	$\mathit{math\_rel}\ \{B_1\}\{B_2\}\}$	$\gg=$

21	$\mathit{cast} \left\{\begin{matrix}\mathit{lpar}\\ \mathit{exp}\end{matrix}\right\}$	$\to \left\{\begin{matrix}\mathit{lpar}\\ \mathit{exp}\end{matrix}\right\}$	$\left\{\begin{matrix}CL\\ C\sqcup E\end{matrix}\right\}$	(**double**)$(x+2)$ (**double**) x
22	$\mathit{cast}\ \mathit{semi}$	$\to \mathit{exp}\ \mathit{semi}$		(**int**);
23	$\mathit{sizeof\_like}\ \mathit{cast}$	$\to \mathit{exp}$		**sizeof**(**double**)
24	$\mathit{sizeof\_like}\ \mathit{exp}$	$\to \mathit{exp}$	$S\sqcup E$	**sizeof** x
25	$\mathit{int\_like} \left\{\begin{matrix}\mathit{int\_like}\\ \mathit{struct\_like}\end{matrix}\right\}$	$\to \left\{\begin{matrix}\mathit{int\_like}\\ \mathit{struct\_like}\end{matrix}\right\}$	$I\sqcup\left\{\begin{matrix}I\\ S\end{matrix}\right\}$	**extern char**
26	$\mathit{int\_like}\ \mathit{exp} \left\{\begin{matrix}\mathit{raw\_int}\\ \mathit{struct\_like}\end{matrix}\right\}$	$\to \mathit{int\_like} \left\{\begin{matrix}\mathit{raw\_int}\\ \mathit{struct\_like}\end{matrix}\right\}$		**extern**`"Ada"` **int**
27	$\mathit{int\_like} \left\{\begin{matrix}\mathit{exp}\\ \mathit{ubinop}\\ \mathit{colon}\end{matrix}\right\}$	$\to \mathit{decl\_head} \left\{\begin{matrix}\mathit{exp}\\ \mathit{ubinop}\\ \mathit{colon}\end{matrix}\right\}$	$D=I\sqcup$	**int** x **int** $*x$ **unsigned** :
28	$\mathit{int\_like} \left\{\begin{matrix}\mathit{semi}\\ \mathit{binop}\end{matrix}\right\}$	$\to \mathit{decl\_head} \left\{\begin{matrix}\mathit{semi}\\ \mathit{binop}\end{matrix}\right\}$		**int** x; **int** f(**int** $=4$)
29	$\mathit{public\_like}\ \mathit{colon}$	$\to \mathit{tag}$		**private**:
30	$\mathit{public\_like}$	$\to \mathit{int\_like}$		**private**
31	$\mathit{colcol} \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	$\to \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	$\mathit{qualifier}\ C\left\{\begin{matrix}E\\ I\end{matrix}\right\}$	**C**$::x$
32	$\mathit{colcol}\ \mathit{colcol}$	$\to \mathit{colcol}$		**C**::**B**::
33	$\mathit{decl\_head}\ \mathit{comma}$	$\to \mathit{decl\_head}$	$DC\sqcup$	**int** x,
34	$\mathit{decl\_head}\ \mathit{ubinop}$	$\to \mathit{decl\_head}$	$D\{U\}$	**int** $*$
†35	$\mathit{decl\_head}\ \mathit{exp}$	$\to \mathit{decl\_head}$	DE^*	**int** x
36	$\mathit{decl\_head} \left\{\begin{matrix}\mathit{binop}\\ \mathit{colon}\end{matrix}\right\} \mathit{exp} \left\{\begin{matrix}\mathit{comma}\\ \mathit{semi}\\ \mathit{rpar}\end{matrix}\right\}$	$\to \mathit{decl\_head} \left\{\begin{matrix}\mathit{comma}\\ \mathit{semi}\\ \mathit{rpar}\end{matrix}\right\}$	$D=D\left\{\begin{matrix}B\\ C\end{matrix}\right\}E$	**int** f(**int** $x=2$) **int** b : 1
37	$\mathit{decl\_head}\ \mathit{cast}$	$\to \mathit{decl\_head}$		**int** f(**int**)
†38	$\mathit{decl\_head} \left\{\begin{matrix}\mathit{int\_like}\\ \mathit{lbrace}\\ \mathit{decl}\end{matrix}\right\}$	$\to \mathit{fn\_decl} \left\{\begin{matrix}\mathit{int\_like}\\ \mathit{lbrace}\\ \mathit{decl}\end{matrix}\right\}$	$F=D\ \mathit{din}$	**long** $\mathit{time}()$ {
39	$\mathit{decl\_head}\ \mathit{semi}$	$\to \mathit{decl}$		**int** n;
40	$\mathit{decl}\ \mathit{decl}$	$\to \mathit{decl}$	$D_1\ \mathit{force}\ D_2$	**int** n; **double** x;
†41	$\mathit{decl} \left\{\begin{matrix}\mathit{stmt}\\ \mathit{function}\end{matrix}\right\}$	$\to \left\{\begin{matrix}\mathit{stmt}\\ \mathit{function}\end{matrix}\right\}$	$D\ \mathit{big\_force}\left\{\begin{matrix}S\\ F\end{matrix}\right\}$	**extern** n; $\mathit{main}()$ { }
42	$\mathit{base} \left\{\begin{matrix}\mathit{int\_like}\\ \mathit{exp}\end{matrix}\right\} \mathit{comma}$	$\to \mathit{base}$	$B\sqcup\left\{\begin{matrix}I\\ E\end{matrix}\right\}C\ \mathit{opt}9$	: **public A**, : $i(5)$,
43	$\mathit{base} \left\{\begin{matrix}\mathit{int\_like}\\ \mathit{exp}\end{matrix}\right\} \mathit{lbrace}$	$\to \mathit{lbrace}$	$B\sqcup\left\{\begin{matrix}I\\ E\end{matrix}\right\}\sqcup L$	**D** : **public A** {
44	$\mathit{struct\_like}\ \mathit{lbrace}$	$\to \mathit{struct\_head}$	$S\sqcup L$	**struct** {
45	$\mathit{struct\_like} \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\} \mathit{semi}$	$\to \mathit{decl\_head}\ \mathit{semi}$	$S\sqcup\left\{\begin{matrix}E^{**}\\ I^{**}\end{matrix}\right\}$	**struct forward**;
46	$\mathit{struct\_like} \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\} \mathit{lbrace}$	$\to \mathit{struct\_head}$	$S\sqcup\left\{\begin{matrix}E^{**}\\ I^{**}\end{matrix}\right\}\sqcup L$	**struct name_info** {
47	$\mathit{struct\_like} \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\} \mathit{colon}$	$\to \mathit{struct\_like} \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\} \mathit{base}$		**class C** :
†48	$\mathit{struct\_like} \left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	$\to \mathit{int\_like}$	$S\sqcup\left\{\begin{matrix}E\\ I\end{matrix}\right\}$	**struct name_info** z;

49	*struct_head* $\begin{Bmatrix}\textit{decl}\\ \textit{stmt}\\ \textit{function}\end{Bmatrix}$ *rbrace*	→ *int_like*	$S\ \textit{in}\ \textit{force}\begin{Bmatrix}D\\ S\\ F\end{Bmatrix}\textit{out}\ \textit{force}\ R$	**struct** { declaration }
50	*struct_head rbrace*	→ *int_like*	S\,R	**class C** { }
51	*fn_decl decl*	→ *fn_decl*	*F force D*	$f(z)$ **double** z;
†52	*fn_decl stmt*	→ *function*	*F out out force S*	*main*() ...
53	*function* $\begin{Bmatrix}\textit{stmt}\\ \textit{decl}\\ \textit{function}\end{Bmatrix}$	→ $\begin{Bmatrix}\textit{stmt}\\ \textit{decl}\\ \textit{function}\end{Bmatrix}$	$F\ \textit{big\_force}\begin{Bmatrix}S\\ D\\ F\end{Bmatrix}$	outer block
54	*lbrace rbrace*	→ *stmt*	L\,R	empty statement
55	*lbrace* $\begin{Bmatrix}\textit{stmt}\\ \textit{decl}\\ \textit{function}\end{Bmatrix}$ *rbrace*	→ *stmt*	*force L in force S force back R out force*	compound statement
56	*lbrace exp* [*comma*] *rbrace*	→ *exp*		initializer
57	*if_like exp*	→ *if_clause*	I␣E	**if** (z)
58	*else_like colon*	→ *else_like base*		**try** :
59	*else_like lbrace*	→ *else_head lbrace*		**else** {
60	*else_like stmt*	→ *stmt*	*force E in bsp S out force*	**else** $x = 0$;
61	*else_head* $\begin{Bmatrix}\textit{stmt}\\ \textit{exp}\end{Bmatrix}$	→ *stmt*	*force E bsp noop cancel S bsp*	**else** {$x = 0$; }
62	*if_clause lbrace*	→ *if_head lbrace*		**if** (x) {
63	*if_clause stmt else_like if_like*	→ *if_like*	*force I in bsp S out force E*␣*I*	**if** (x) y; **else if**
64	*if_clause stmt else_like*	→ *else_like*	*force I in bsp S out force E*	**if** (x) y; **else**
65	*if_clause stmt*	→ *else_like stmt*		**if** (x) y;
66	*if_head* $\begin{Bmatrix}\textit{stmt}\\ \textit{exp}\end{Bmatrix}$ *else_like if_like*	→ *if_like*	*force I bsp noop cancel S force E*␣*I*	**if** (x) { y; } **else if**
67	*if_head* $\begin{Bmatrix}\textit{stmt}\\ \textit{exp}\end{Bmatrix}$ *else_like*	→ *else_like*	*force I bsp noop cancel S force E*	**if** (x) { y; } **else**
68	*if_head* $\begin{Bmatrix}\textit{stmt}\\ \textit{exp}\end{Bmatrix}$	→ *else_head* $\begin{Bmatrix}\textit{stmt}\\ \textit{exp}\end{Bmatrix}$		**if** (x) { y }
69	*do_like stmt else_like semi*	→ *stmt*	*D bsp noop cancel S cancel noop bsp ES*	**do** $f(x)$; **while** ($g(x)$);
70	*case_like semi*	→ *stmt*		**return**;
71	*case_like colon*	→ *tag*		**default**:
72	*case_like exp*	→ *exp*	C␣E	**return** 0
†73	*catch_like* $\begin{Bmatrix}\textit{cast}\\ \textit{exp}\end{Bmatrix}$	→ *fn_decl*	$C\begin{Bmatrix}C\\ E\end{Bmatrix}\textit{din}$	**catch**(...)
74	*tag tag*	→ *tag*	T_1 *bsp* T_2	**case** 0: **case** 1:
75	*tag* $\begin{Bmatrix}\textit{stmt}\\ \textit{decl}\\ \textit{function}\end{Bmatrix}$	→ $\begin{Bmatrix}\textit{stmt}\\ \textit{decl}\\ \textit{function}\end{Bmatrix}$	*force back T bsp S*	**case** 0: $z = 0$;
†76	*stmt* $\begin{Bmatrix}\textit{stmt}\\ \textit{decl}\\ \textit{function}\end{Bmatrix}$	→ $\begin{Bmatrix}\textit{stmt}\\ \textit{decl}\\ \textit{function}\end{Bmatrix}$	$S\begin{Bmatrix}\textit{force}\ S\\ \textit{big\_force}\ D\\ \textit{big\_force}\ F\end{Bmatrix}$	$x = 1$; $y = 2$;
77	*semi*	→ *stmt*	␣S	empty statement
†78	*lproc* $\begin{Bmatrix}\textit{if\_like}\\ \textit{else\_like}\\ \textit{define\_like}\end{Bmatrix}$	→ *lproc*		**#include** **#else** **#define**
79	*lproc rproc*	→ *insert*		**#endif**

80	*lproc* $\left\{\begin{matrix}\mathit{exp}\ [\mathit{exp}]\\ \mathit{function}\end{matrix}\right\}$ *rproc*	→ *insert*	$I\text{␣}\left\{\begin{matrix}E[\text{␣}\backslash\texttt{5}E]\\ F\end{matrix}\right\}$	**#define** *a* 1 **#define** *a* { *b*; }
81	*section_scrap semi*	→ *stmt*	*MS force*	⟨ section name ⟩;
82	*section_scrap*	→ *exp*		⟨ section name ⟩
83	*insert any*	→ *any*		`|#include|`
84	*prelangle*	→ *binop*	`<`	< not in template
85	*prerangle*	→ *binop*	`>`	> not in template
86	*langle prerangle*	→ *cast*	$L\backslash,P$	⟨ ⟩
87	*langle* $\left\{\begin{matrix}\mathit{decl\_head}\\ \mathit{int\_like}\\ \mathit{exp}\end{matrix}\right\}$ *prerangle*	→ *cast*		⟨**class C**⟩
88	*langle* $\left\{\begin{matrix}\mathit{decl\_head}\\ \mathit{int\_like}\\ \mathit{exp}\end{matrix}\right\}$ *comma*	→ *langle*	$L\left\{\begin{matrix}D\\ I\\ E\end{matrix}\right\}C\ \mathit{opt}9$	⟨**class C**,
89	*template_like exp prelangle*	→ *template_like exp langle*		**template** *a*⟨100⟩
90	*template_like* $\left\{\begin{matrix}\mathit{exp}\\ \mathit{raw\_int}\end{matrix}\right\}$	→ $\left\{\begin{matrix}\mathit{exp}\\ \mathit{raw\_int}\end{matrix}\right\}$	$T\text{␣}\left\{\begin{matrix}E\\ R\end{matrix}\right\}$	**C** :: **template** *a*( )
91	*template_like*	→ *raw_int*		**template**⟨**class T**⟩
92	*new_like lpar exp rpar*	→ *new_like*		**new**(*nothrow*)
93	*new_like cast*	→ *exp*	$N\text{␣}C$	**new** (**int** ∗)
†94	*new_like*	→ *new_exp*		**new C**( )
95	*new_exp* $\left\{\begin{matrix}\mathit{int\_like}\\ \mathit{const\_like}\end{matrix}\right\}$	→ *new_exp*	$N\text{␣}\left\{\begin{matrix}I\\ C\end{matrix}\right\}$	**new const int**
96	*new_exp struct_like* $\left\{\begin{matrix}\mathit{exp}\\ \mathit{int\_like}\end{matrix}\right\}$	→ *new_exp*	$N\text{␣}S\text{␣}\left\{\begin{matrix}E\\ I\end{matrix}\right\}$	**new struct S**
97	*new_exp raw_ubin*	→ *new_exp*	$N\texttt{\{}R\texttt{\}}$	**new int**∗[2]
98	*new_exp* $\left\{\begin{matrix}\mathit{lpar}\\ \mathit{exp}\end{matrix}\right\}$	→ *exp* $\left\{\begin{matrix}\mathit{lpar}\\ \mathit{exp}\end{matrix}\right\}$	$E=N\left\{\begin{matrix}\\ \text{␣}\end{matrix}\right\}$	**operator**[ ](**int**) **new int**(2)
†99	*new_exp*	→ *exp*		**new int**;
100	*ftemplate prelangle*	→ *ftemplate langle*		*make_pair*⟨**int**, **int**⟩
101	*ftemplate*	→ *exp*		*make_pair*(1, 2)
102	*for_like exp*	→ *else_like*	$F\text{␣}E$	**while** (1)
103	*raw_ubin const_like*	→ *raw_ubin*	$RC\backslash\text{␣}$	∗**const** *x*
104	*raw_ubin*	→ *ubinop*		∗ *x*
105	*const_like*	→ *int_like*		**const** *x*
106	*raw_int prelangle*	→ *raw_int langle*		**C**⟨
107	*raw_int colcol*	→ *colcol*		**C** ::
108	*raw_int cast*	→ *raw_int*		**C**⟨**class T**⟩
109	*raw_int lpar*	→ *exp lpar*		**complex**(*x*, *y*)
†110	*raw_int*	→ *int_like*		**complex** *z*
†111	*operator_like* $\left\{\begin{matrix}\mathit{binop}\\ \mathit{unop}\\ \mathit{ubinop}\end{matrix}\right\}$	→ *exp*	$O\texttt{\{}\left\{\begin{matrix}B\\ U\\ U\end{matrix}\right\}\texttt{\}}$	**operator**+
112	*operator_like* $\left\{\begin{matrix}\mathit{new\_like}\\ \mathit{delete\_like}\end{matrix}\right\}$	→ *exp*	$O\text{␣}\left\{\begin{matrix}N\\ S\end{matrix}\right\}$	**operator delete**
113	*operator_like comma*	→ *exp*		**operator**,
†114	*operator_like*	→ *new_exp*		**operator char**∗
115	*typedef_like* $\left\{\begin{matrix}\mathit{int\_like}\\ \mathit{cast}\end{matrix}\right\}\left\{\begin{matrix}\mathit{comma}\\ \mathit{semi}\end{matrix}\right\}$	→ *typedef_like exp* $\left\{\begin{matrix}\mathit{comma}\\ \mathit{semi}\end{matrix}\right\}$		**typedef int I**,

116	*typedef_like int_like*	→ *typedef_like*	$T_\sqcup I$	**typedef char**
†117	*typedef_like exp*	→ *typedef_like*	$T_\sqcup E^{**}$	**typedef I** `@[@]` (∗**P**)
118	*typedef_like comma*	→ *typedef_like*	$TC_\sqcup$	**typedef int x**,
119	*typedef_like semi*	→ *decl*		**typedef int x**, **y**;
120	*typedef_like ubinop* $\left\{\begin{matrix}\mathit{cast}\\\mathit{ubinop}\end{matrix}\right\}$	→ *typedef_like* $\left\{\begin{matrix}\mathit{cast}\\\mathit{ubinop}\end{matrix}\right\}$	$\left\{\begin{matrix}C=\{U\}C\\U_2=\{U_1\}U_2\end{matrix}\right\}$	**typedef** ∗∗(**CPtr**)
121	*delete_like lpar rpar*	→ *delete_like*	$DL\backslash{,}R$	**delete**[ ]
122	*delete_like exp*	→ *exp*	$D_\sqcup E$	**delete** *p*
†123	*question exp* $\left\{\begin{matrix}\mathit{colon}\\\mathit{base}\end{matrix}\right\}$	→ *binop*		? *x* : ? *f*( ) :
124	*begin_arg end_arg*	→ *exp*		`@[`**char**∗`@]`
125	*any_other end_arg*	→ *end_arg*		**char**∗`@]`
126	*alignas_like decl_head*	→ *attr*		**alignas**(**struct** *s* ∗)
127	*alignas_like exp*	→ *attr*		**alignas**(8)
128	*lbrack lbrack*	→ *attr_head*		attribute begins
129	*lbrack*	→ *lpar*		[elsewhere
130	*rbrack*	→ *rpar*		] elsewhere
131	*attr_head rbrack rbrack*	→ *attr*		[[. . .]]
132	*attr_head exp*	→ *attr_head*		[[*deprecated*
133	*attr_head using_like exp colon*	→ *attr_head*		[[**using** `NS`:
134	*attr* $\left\{\begin{matrix}\mathit{lbrace}\\\mathit{stmt}\end{matrix}\right\}$	→ $\left\{\begin{matrix}\mathit{lbrace}\\\mathit{stmt}\end{matrix}\right\}$	$A_\sqcup\left\{\begin{matrix}S\\L\end{matrix}\right\}$	[[*likely*]] {
135	*attr tag*	→ *tag*	$A_\sqcup T$	[[*likely*]] **case** 0:
136	*attr semi*	→ *stmt*		[[*fallthrough*]];
137	*attr attr*	→ *attr*	$A_{1\sqcup}A_2$	**alignas**(*x*) [[. . .]]
138	*attr decl_head*	→ *decl_head*		[[*nodiscard*]] *f*( )
139	*decl_head attr*	→ *decl_head*		(**int** *x* [[*deprecated*]])
140	*using_like*	→ *int_like*		**using** not in attributes
141	*struct_like attr*	→ *struct_like*	$S_\sqcup A$	**struct** [[*deprecated*]]
142	*exp attr*	→ *attr*	$E_\sqcup A$	**enum** {*x* [[. . .]]}
143	*attr typedef_like*	→ *typedef_like*	$A_\sqcup T$	[[*deprecated*]] **typedef**
144	*raw_int lbrack*	→ *exp*		**int**[3]
145	*attr_head comma*	→ *attr_head*		[[*x*, *y*
146	*if_head attr*	→ *if_head*	$I_\sqcup A$	**if** (*x*) [[*unlikely*]] {
147	*lbrack lbrack rbrack rbrack*	→ *exp*		[[]]
148	*attr function*	→ *function*	$A_\sqcup F$	attribute and function
149	*default_like colon*	→ *case_like colon*		**default**:
150	*default_like*	→ *exp*		*f*( ) = **default**;
151	*struct_like struct_like*	→ *struct_like*	$S_{1\sqcup}S_2$	**enum class**
152	*exp colcol int_like*	→ *int_like*		*std* :: **atomic**
†153	*langle struct_like* $\left\{\begin{matrix}\mathit{exp}\\\mathit{int\_like}\end{matrix}\right\}$ *comma*	→ *langle*	$LS\left\{\begin{matrix}E^{**}\\I^{**}\end{matrix}\right\}C$	⟨**typename** *t*,
†154	*langle struct_like* $\left\{\begin{matrix}\mathit{exp}\\\mathit{int\_like}\end{matrix}\right\}$ *prerangle*	→ *cast*	$LS\left\{\begin{matrix}E^{**}\\I^{**}\end{matrix}\right\}P$	⟨**typename** *t*⟩
155	*template_like cast struct_like*	→ *struct_like*	$T_\sqcup CS$	**template**⟨ . . . ⟩ **class**
156	*tag rbrace*	→ *decl rbrace*		**public**: }
157	*fn_decl attr*	→ *fn_decl*	$F_\sqcup A$	**void** *f*( ) **noexcept**
158	*alignas_like cast*	→ *attr*		**alignas**(**int**)

†Notes

Rules 1, 38, 52, and 73: The *din*s and *out*s are suppressed if `CWEAVE` has been invoked with the `-i` option.
Rule 35: The *exp* must not be immediately followed by *lpar*, *lbrack*, *exp*, or *cast*.
Rule 41: The *big_force* becomes *force* if `CWEAVE` has been invoked with the `-o` option.
Rule 48: The *exp* or *int_like* must not be immediately followed by *base*.
Rule 76: The *force* in the *stmt* line becomes *bsp* if `CWEAVE` has been invoked with the `-f` option, and the *big_force* in the *decl* and *function* lines becomes *force* if `CWEAVE` has been invoked with the `-o` option.
Rule 78: The *define_like* case calls *make_underlined* on the following scrap.
Rule 94: The *new_like* must not be immediately followed by *lpar*.
Rule 99: The *new_exp* must not be immediately followed by *raw_int*, *struct_like*, or *colcol*.
Rule 110: The *raw_int* must not be immediately followed by *langle*.
Rule 111: The operator after *operator_like* must not be immediately followed by a *binop*.
Rule 114: The *operator_like* must not be immediately followed by *raw_ubin*.
Rule 117: The *exp* must not be immediately followed by *lpar*, *exp*, or *cast*.
Rule 123: The mathness of the *colon* or *base* changes to ‘yes’.
Rules 153, 154: *make_reserved* is called only if `CWEAVE` has been invoked with the `+t` option.

115* Token lists in *tok_mem* are composed of the following kinds of items for TEX output.

- Character codes and special codes like *force* and *math_rel* represent themselves;
- $id\_flag + p$ represents `\\{`identifier p`}`;
- $res\_flag + p$ represents `\&{`identifier p`}`;
- $section\_flag + p$ represents section name p;
- $tok\_flag + p$ represents token list number p;
- $inner\_tok\_flag + p$ represents token list number p, to be translated without line-break controls.

```
#define id_flag 10240   ▷ signifies an identifier ◁
#define res_flag (2 * id_flag)   ▷ signifies a reserved word ◁
#define section_flag (3 * id_flag)   ▷ signifies a section name ◁
#define tok_flag (4 * id_flag)   ▷ signifies a token list ◁
#define inner_tok_flag (5 * id_flag)   ▷ signifies a token list in '| ... |' ◁
```

⟨ Predeclaration of procedures 8* ⟩ +≡

```
#if 0
  static void print_text(text_pointer p);
#endif
```

116*.

```
#if 0
  static void print_text(      ▷ prints a token list for debugging; not used in main ◁
      text_pointer p)
  {
    token_pointer j;      ▷ index into tok_mem ◁
    sixteen_bits r;      ▷ remainder of token after the flag has been stripped off ◁
    if (p ≥ text_ptr) printf("BAD");
    else
      for (j ← *p; j < *(p + 1); j++) {
        r ← *j % id_flag;
        switch (*j) {
        case id_flag: printf("\\\\{"); print_id((name_dir + r)); putchar('}'); break;
        case res_flag: printf("\\&{"); print_id((name_dir + r)); putchar('}'); break;
        case section_flag: putchar('<'); print_section_name((name_dir + r)); putchar('>'); break;
        case tok_flag: printf("[[%d]]", (int) r); break;
        case inner_tok_flag: printf("|[[%d]]|", (int) r); break;
        default: ⟨Print token r in symbolic form 117⟩
        }
      }
    update_terminal;
  }
#endif
```

128* Now comes the code that tries to match each production starting with a particular type of scrap. Whenever a match is discovered, the *squash* or *reduce* function will cause the appropriate action to be performed.

⟨ Cases for *exp* 128* ⟩ ≡

```
if (cat1 ≡ lbrace ∨ cat1 ≡ int_like ∨ cat1 ≡ decl) {
  make_underlined(pp); big_app1(pp);
  if (indent_param_decl) big_app(dindent);
  reduce(pp, 1, fn_decl, 0, 1);
}
else if (cat1 ≡ unop) squash(pp, 2, exp, −2, 2);
else if ((cat1 ≡ binop ∨ cat1 ≡ ubinop) ∧ cat2 ≡ exp) squash(pp, 3, exp, −2, 3);
else if (cat1 ≡ comma ∧ cat2 ≡ exp) {
  big_app2(pp); app(opt); app('9'); big_app1(pp + 2); reduce(pp, 3, exp, −2, 4);
}
else if (cat1 ≡ lpar ∧ cat2 ≡ rpar ∧ cat3 ≡ colon) reduce(pp + 3, 0, base, 0, 5);
else if (cat1 ≡ cast ∧ cat2 ≡ colon) reduce(pp + 2, 0, base, 0, 5);
else if (cat1 ≡ semi) squash(pp, 2, stmt, −1, 6);
else if (cat1 ≡ colon) {
  make_underlined(pp); squash(pp, 2, tag, −1, 7);
}
else if (cat1 ≡ rbrace) reduce(pp, 0, stmt, −1, 8);
else if (cat1 ≡ lpar ∧ cat2 ≡ rpar ∧ (cat3 ≡ const_like ∨ cat3 ≡ case_like)) {
  big_app1_insert(pp + 2, '␣'); reduce(pp + 2, 2, rpar, 0, 9);
}
else if (cat1 ≡ cast ∧ (cat2 ≡ const_like ∨ cat2 ≡ case_like)) {
  big_app1_insert(pp + 1, '␣'); reduce(pp + 1, 2, cast, 0, 9);
}
else if (cat1 ≡ exp ∨ cat1 ≡ cast) squash(pp, 2, exp, −2, 10);
else if (cat1 ≡ attr) {
  big_app1_insert(pp, '␣'); reduce(pp, 2, exp, −2, 142);
}
else if (cat1 ≡ colcol ∧ cat2 ≡ int_like) squash(pp, 3, int_like, −2, 152);
```

This code is used in section 121.

138*. ⟨ Cases for *decl_head* 138* ⟩ ≡

```
if (cat1 ≡ comma) {
  big_app2(pp); big_app('␣'); reduce(pp, 2, decl_head, −1, 33);
}
else if (cat1 ≡ ubinop) {
  big_app1_insert(pp, '{'); big_app('}'); reduce(pp, 2, decl_head, −1, 34);
}
else if (cat1 ≡ exp ∧ cat2 ≠ lpar ∧ cat2 ≠ lbrack ∧ cat2 ≠ exp ∧ cat2 ≠ cast) {
  make_underlined(pp + 1); squash(pp, 2, decl_head, −1, 35);
}
else if ((cat1 ≡ binop ∨ cat1 ≡ colon) ∧ cat2 ≡ exp ∧ (cat3 ≡ comma ∨ cat3 ≡ semi ∨ cat3 ≡ rpar))
  squash(pp, 3, decl_head, −1, 36);
else if (cat1 ≡ cast) squash(pp, 2, decl_head, −1, 37);
else if (cat1 ≡ lbrace ∨ cat1 ≡ int_like ∨ cat1 ≡ decl) {
  if (indent_param_decl) big_app(dindent);
  squash(pp, 1, fn_decl, 0, 38);
}
else if (cat1 ≡ semi) squash(pp, 2, decl, −1, 39);
else if (cat1 ≡ attr) {
  big_app1_insert(pp, '␣'); reduce(pp, 2, decl_head, −1, 139);
}
```

This code is used in section 121.

139*. ⟨ Cases for *decl* 139* ⟩ ≡

```
if (cat1 ≡ decl) {
  big_app1_insert(pp, force); reduce(pp, 2, decl, −1, 40);
}
else if (cat1 ≡ stmt ∨ cat1 ≡ function) {
  big_app1_insert(pp, order_decl_stmt ? big_force : force); reduce(pp, 2, cat1, −1, 41);
}
```

This code is used in section 121.

143*. ⟨ Cases for *fn_decl* 143* ⟩ ≡

```
if (cat1 ≡ decl) {
  big_app1_insert(pp, force); reduce(pp, 2, fn_decl, 0, 51);
}
else if (cat1 ≡ stmt) {
  big_app1(pp);
  if (indent_param_decl) {
    app(outdent); app(outdent);
  }
  big_app(force); big_app1(pp + 1); reduce(pp, 2, function, −1, 52);
}
else if (cat1 ≡ attr) {
  big_app1_insert(pp, '␣'); reduce(pp, 2, fn_decl, 0, 157);
}
```

This code is used in section 121.

153* ⟨ Cases for *catch_like* 153* ⟩ ≡

```
if (cat1 ≡ cast ∨ cat1 ≡ exp) {
  big_app1(pp);
  if (indent_param_decl) big_app(dindent);
  big_app1(pp + 1); reduce(pp, 2, fn_decl, 0, 73);
}
```

This code is used in section 121.

156* ⟨ Cases for *stmt* 156* ⟩ ≡

```
if (cat1 ≡ stmt ∨ cat1 ≡ decl ∨ cat1 ≡ function) {
  big_app1_insert(pp, (cat1 ≡ function ∨ cat1 ≡ decl) ? (order_decl_stmt ? big_force : force) :
      (force_lines ? force : break_space)); reduce(pp, 2, cat1, −1, 76);
}
```

This code is used in section 121.

184* And here now is the code that applies productions as long as possible. Before applying the production mechanism, we must make sure it has good input (at least four scraps, the length of the lhs of the longest rules), and that there is enough room in the memory arrays to hold the appended tokens and texts. Here we use a very conservative test; it's more important to make sure the program will still work if we change the production rules (within reason) than to squeeze the last bit of space from the memory arrays.

```
#define safe_tok_incr 20
#define safe_text_incr 10
#define safe_scrap_incr 10
```

⟨ Reduce the scraps using the productions until no more rules apply 184* ⟩ ≡

```
while (true) {
  ⟨ Make sure the entries pp through pp + 3 of cat are defined 185 ⟩
  if (tok_ptr + safe_tok_incr > tok_mem_end) {
    if (tok_ptr > max_tok_ptr) max_tok_ptr ← tok_ptr;
    overflow(_("token"));
  }
  if (text_ptr + safe_text_incr > tok_start_end) {
    if (text_ptr > max_text_ptr) max_text_ptr ← text_ptr;
    overflow(_("text"));
  }
  if (pp > lo_ptr) break;
  init_mathness ← cur_mathness ← maybe_math;
  ⟨ Match a production at pp, or increase pp if there is no match 121 ⟩
}
```

This code is used in section 188.

190* If the initial sequence of scraps does not reduce to a single scrap, we concatenate the translations of all remaining scraps, separated by blank spaces, with dollar signs surrounding the translations of scraps where appropriate.

```
⟨ Combine the irreducible scraps that remain 190* ⟩ ≡
  ⟨ If semi-tracing, show the irreducible scraps 191* ⟩
  for (j ← scrap_base; j ≤ lo_ptr; j++) {
    if (j ≠ scrap_base) app('␣');
    if (j→mathness % 4 ≡ yes_math) app('$');
    app(tok_flag + (int)(j→trans − tok_start));
    if (j→mathness/4 ≡ yes_math) app('$');
    if (tok_ptr + 6 > tok_mem_end) overflow(_("token"));
  }
  freeze_text; return text_ptr − 1;
```

This code is used in section 188.

191*

```
⟨ If semi-tracing, show the irreducible scraps 191* ⟩ ≡
  if (lo_ptr > scrap_base ∧ tracing ≡ partly) {
    printf(_("\nIrreducible␣scrap␣sequence␣in␣section␣%d:"), (int) section_count); mark_harmless;
    for (j ← scrap_base; j ≤ lo_ptr; j++) {
      putchar('␣'); print_cat(j→cat);
    }
  }
```

This code is used in section 190*.

192*

```
⟨ If tracing, print an indication of where we are 192* ⟩ ≡
  if (tracing ≡ fully) {
    printf(_("\nTracing␣after␣l.␣%d:\n"), cur_line); mark_harmless;
    if (loc > buffer + 50) {
      printf("..."); term_write(loc − 51, 51);
    }
    else term_write(buffer, loc − buffer);
  }
```

This code is used in section 188.

197*

```
⟨ Make sure that there is room for the new scraps, tokens, and texts 197* ⟩ ≡
  if (scrap_ptr + safe_scrap_incr > scrap_info_end ∨ tok_ptr + safe_tok_incr > tok_mem_end
        ∨ text_ptr + safe_text_incr > tok_start_end) {
    if (scrap_ptr > max_scr_ptr) max_scr_ptr ← scrap_ptr;
    if (tok_ptr > max_tok_ptr) max_tok_ptr ← tok_ptr;
    if (text_ptr > max_text_ptr) max_text_ptr ← text_ptr;
    overflow(_("scrap/token/text"));
  }
```

This code is used in sections 196 and 205.

199* The following code must use *app_tok* instead of *app* in order to protect against overflow. Note that $tok\_ptr + 1 \le max\_toks$ after *app_tok* has been used, so another *app* is legitimate before testing again.

Many of the special characters in a string must be prefixed by '\' so that TEX will print them properly.

```
⟨ Append a string or constant 199* ⟩ ≡
  { int count ← −1;      ▷ characters remaining before string break ◁
    switch (next_control) {
    case constant: app_str("\\T{"); break;
    case string: count ← 20; app_str("\\.{"); break;
    default: app_str("\\vb{");
    }
    while (id_first < id_loc) {
      if (count ≡ 0) {      ▷ insert a discretionary break in a long string ◁
        app_str("}\\)\\.{"); count ← 20;
      }
      switch (∗id_first) {
      case '␣': case '\\': case '#': case '$': case '^': case '{': case '}': case '~': case '&':
        case '_': app('\\'); break;
      case '%':
        if (next_control ≡ constant) {
          app_str("}\\p{");      ▷ special macro for 'hex exponent' ◁
          id_first++;      ▷ skip '%' ◁
        }
        else app('\\');
        break;
      case '@':
        if (∗(id_first + 1) ≡ '@') id_first++;
        else err_print(_("!␣Double␣@␣should␣be␣used␣in␣strings"));
        break;
      default:      ▷ high-bit character handling ◁
        if ((eight_bits)(∗id_first) > °177) app_tok(quoted_char);
      }
      app_tok(∗id_first++); count−−;
    }
    app('}'); app_scrap(exp, maybe_math);
  }
```

This code is used in section 196.

203* When the '|' that introduces C text is sensed, a call on *C_translate* will return a pointer to the TEX translation of that text. If scraps exist in *scrap_info*, they are unaffected by this translation process.

```
  static text_pointer C_translate(void)
  {
    text_pointer p;      ▷ points to the translation ◁
    scrap_pointer save_base ← scrap_base;      ▷ holds original value of scrap_base ◁

    scrap_base ← scrap_ptr + 1; C_parse(section_name);      ▷ get the scraps together ◁
    if (next_control ≠ '|') err_print(_("!␣Missing␣'|'␣after␣C␣text"));
    app_tok(cancel); app_scrap(insert, maybe_math);      ▷ place a cancel token as a final "comment" ◁
    p ← translate();      ▷ make the translation ◁
    if (scrap_ptr > max_scr_ptr) max_scr_ptr ← scrap_ptr;
    scrap_ptr ← scrap_base − 1; scrap_base ← save_base;      ▷ scrap the scraps ◁
    return p;
  }
```

211*.

```
static void push_level(    ▷ suspends the current level ◁
    text_pointer p)
{
  if (stack_ptr ≡ stack_end) overflow(_("stack"));
  if (stack_ptr > stack) {    ▷ save current state ◁
    stack_ptr→end_field ← cur_end; stack_ptr→tok_field ← cur_tok; stack_ptr→mode_field ← cur_mode;
  }
  stack_ptr++;
  if (stack_ptr > max_stack_ptr) max_stack_ptr ← stack_ptr;
  cur_tok ← *p; cur_end ← *(p + 1);
}
```

224*. ⟨ Skip next character, give error if not '@' 224* ⟩ ≡

```
if (*k++ ≠ '@') {
  fputs(_("\n!␣Illegal␣control␣code␣in␣section␣name:␣<"), stdout);
  print_section_name(cur_section_name); printf(">␣"); mark_error;
}
```

This code is used in section 223.

225*. The C text enclosed in | ... | should not contain '|' characters, except within strings. We put a '|' at the front of the buffer, so that an error message that displays the whole buffer will look a little bit sensible. The variable *delim* is zero outside of strings, otherwise it equals the delimiter that began the string being copied.

⟨ Copy the C text into the *buffer* array 225* ⟩ ≡

```
j ← limit + 1; *j ← '|'; delim ← 0;
while (true) {
  if (k ≥ k_limit) {
    fputs(_("\n!␣C␣text␣in␣section␣name␣didn't␣end:␣<"), stdout);
    print_section_name(cur_section_name); printf(">␣"); mark_error; break;
  }
  b ← *(k++);
  if (b ≡ '@' ∨ (b ≡ '\\' ∧ delim ≠ 0)) ⟨ Copy a quoted character into the buffer 226* ⟩
  else {
    if (b ≡ '\'' ∨ b ≡ '"') {
      if (delim ≡ 0) delim ← b;
      else if (delim ≡ b) delim ← 0;
    }
    if (b ≠ '|' ∨ delim ≠ 0) {
      if (j > buffer + long_buf_size − 3) overflow(_("buffer"));
      *(++j) ← b;
    }
    else break;
  }
}
```

This code is used in section 223.

226*. ⟨ Copy a quoted character into the buffer 226* ⟩ ≡

```
{
  if (j > buffer + long_buf_size − 4) overflow(_("buffer"));
  *(++j) ← b; *(++j) ← *(k++);
}
```

This code is used in section 225*.

227*. Phase two processing. We have assembled enough pieces of the puzzle in order to be ready to specify the processing in `CWEAVE`'s main pass over the source file. Phase two is analogous to phase one, except that more work is involved because we must actually output the TEX material instead of merely looking at the `CWEB` specifications.

```
static void phase_two(void)
{
  phase ← 2;  reset_input();
  if (show_progress) fputs(_("\nWriting␣the␣output␣file..."), stdout);
  section_count ← 0;  format_visible ← true;  copy_limbo();  finish_line();
  flush_buffer(out_buf, false, false);     ▷ insert a blank line, it looks nice ◁
  while (¬input_has_ended) ⟨Translate the current section 230⟩
}
```

232*. In the TEX part of a section, we simply copy the source text, except that index entries are not copied and C text within `| ... |` is translated.

```
⟨Translate the TEX part of the current section 232*⟩ ≡
  do switch (next_control ← copy_TEX()) {
  case '|': init_stack; output_C(); break;
  case '@': out('@'); break;
  case TEX_string: case noop: case xref_roman: case xref_wildcard: case xref_typewriter:
    case section_name: loc −= 2;  next_control ← get_next();     ▷ skip to @> ◁
    if (next_control ≡ TEX_string) err_print(_("!␣TeX␣string␣should␣be␣in␣C␣text␣only"));
    break;
  case thin_space: case math_break: case ord: case line_break: case big_line_break: case no_line_break:
    case join: case pseudo_semi: case macro_arg_open: case macro_arg_close: case output_defs_code:
    err_print(_("!␣You␣can't␣do␣that␣in␣TeX␣text")); break;
  } while (next_control < format_code);
```

This code is used in section 230.

236* Keeping in line with the conventions of the C preprocessor (and otherwise contrary to the rules of `CWEB`) we distinguish here between the case that '`(`' immediately follows an identifier and the case that the two are separated by a space. In the latter case, and if the identifier is not followed by '`(`' at all, the replacement text starts immediately after the identifier. In the former case, it starts after we scan the matching '`)`'.

⟨ Start a macro definition 236* ⟩ ≡

```
{
  if (save_line ≠ out_line ∨ save_place ≠ out_ptr ∨ space_checked) app(backup);
  if (¬space_checked) {
    emit_space_if_needed; save_position;
  }
  app_str("\\D");     ▷ this will produce '#define ' ◁
  if ((next_control ← get_next()) ≠ identifier) err_print(_("!␣Improper␣macro␣definition"));
  else {
    app_cur_id(false);
    if (∗loc ≡ '(') {
      app('$');
    reswitch:
      switch (next_control ← get_next()) {
      case '(': case ',': app(next_control); goto reswitch;
      case identifier: app_cur_id(false); goto reswitch;
      case ')': app(next_control); next_control ← get_next(); break;
      case dot_dot_dot: app_str("\\,\\ldots\\,"); app_scrap(raw_int, no_math);
        if ((next_control ← get_next()) ≡ ')') {
          app(next_control); next_control ← get_next(); break;
        }
        /*␣otherwise␣fall␣through␣*/
      default: err_print(_("!␣Improper␣macro␣definition")); break;
      }
      app('$');
    }
    else next_control ← get_next();
    app(break_space); app_scrap(dead, no_math);     ▷ scrap won't take part in the parsing ◁
  }
}
```

This code is used in section 233.

237*. ⟨ Start a format definition 237* ⟩ ≡

```
{
  doing_format ← true;
  if (∗(loc − 1) ≡ 's' ∨ ∗(loc − 1) ≡ 'S') format_visible ← false;
  if (¬space_checked) {
    emit_space_if_needed; save_position;
  }
  app_str("\\F");    ▷ this will produce 'format ' ◁
  next_control ← get_next();
  if (next_control ≡ identifier) {
    app(id_flag + (int)(id_lookup(id_first, id_loc, normal) − name_dir)); app(break_space);
      ▷ this is syntactically separate from what follows ◁
    next_control ← get_next();
    if (next_control ≡ identifier) {
      app(id_flag + (int)(id_lookup(id_first, id_loc, normal) − name_dir)); app_scrap(exp, maybe_math);
      app_scrap(semi, maybe_math); next_control ← get_next();
    }
  }
  if (scrap_ptr ≠ scrap_info + 2) err_print(_("!␣Improper␣format␣definition"));
}
```

This code is used in section 233.

240*. The title of the section and an ≡ or +≡ are made into a scrap that should not take part in the parsing.

⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition 240* ⟩ ≡

```
do next_control ← get_next(); while (next_control ≡ '+');    ▷ allow optional '+=' ◁
if (next_control ≠ '=' ∧ next_control ≠ eq_eq)
  err_print(_("!␣You␣need␣an␣=␣sign␣after␣the␣section␣name"));
else next_control ← get_next();
if (out_ptr > out_buf + 1 ∧ ∗out_ptr ≡ 'Y' ∧ ∗(out_ptr − 1) ≡ '\\') app(backup);
    ▷ the section name will be flush left ◁
app(section_flag + (int)(this_section − name_dir)); cur_xref ← (xref_pointer) this_section→xref;
if (cur_xref→num ≡ file_flag) cur_xref ← cur_xref→xlink;
app_str("${}");
if (cur_xref→num ≠ section_count + def_flag) {
  app_str("\\mathrel+");    ▷ section name is multiply defined ◁
  this_section ← name_dir;    ▷ so we won't give cross-reference info here ◁
}
app_str("\\E");    ▷ output an equivalence sign ◁
app_str("{}$"); app(force); app_scrap(dead, no_math);    ▷ this forces a line break unless '@+' follows ◁
```

This code is used in section 239.

241*. ⟨ Emit the scrap for a section name if present 241* ⟩ ≡

```
if (next_control < section_name) {
  err_print(_("!␣You␣can't␣do␣that␣in␣C␣text")); next_control ← get_next();
}
else if (next_control ≡ section_name) {
  app(section_flag + (int)(cur_section − name_dir)); app_scrap(section_scrap, maybe_math);
  next_control ← get_next();
}
```

This code is used in section 239.

247*. Phase three processing. We are nearly finished! `CWEAVE`'s only remaining task is to write out the index, after sorting the identifiers and index entries.

If the user has set the *no_xref* flag (the `-x` option on the command line), just finish off the page, omitting the index, section name list, and table of contents.

```
static void phase_three(void)
{
  if (no_xref) {
    finish_line(); out_str("\\end");
  }
  else {
    phase ← 3;
    if (show_progress) fputs(_("\nWriting␣the␣index..."), stdout);
    finish_line();
    if ((idx_file ← fopen(idx_file_name, "wb")) ≡ Λ)
      fatal(_("!␣Cannot␣open␣index␣file␣"), idx_file_name);
    if (change_exists) {
      ⟨ Tell about changed sections 250 ⟩
      finish_line(); finish_line();
    }
    out_str("\\inx"); finish_line(); active_file ← idx_file;    ▷ change active file to the index file ◁
    ⟨ Do the first pass of sorting 252 ⟩
    ⟨ Sort and output the index 260 ⟩
    finish_line(); fclose(active_file);    ▷ finished with idx_file ◁
    active_file ← tex_file;    ▷ switch back to tex_file for a tic ◁
    out_str("\\fin"); finish_line();
    if ((scn_file ← fopen(scn_file_name, "wb")) ≡ Λ)
      fatal(_("!␣Cannot␣open␣section␣file␣"), scn_file_name);
    active_file ← scn_file;    ▷ change active file to section listing file ◁
    ⟨ Output all the section names 269 ⟩
    finish_line(); fclose(active_file);    ▷ finished with scn_file ◁
    active_file ← tex_file;
    if (group_found) out_str("\\con"); else out_str("\\end");
  }
  finish_line(); fclose(active_file); active_file ← tex_file ← Λ;
  if (check_for_change) ⟨ Update the result when it has changed 274* ⟩
  if (show_happiness) {
    if (show_progress) new_line;
    fputs(_("Done."), stdout);
  }
  check_complete();    ▷ was all of the change file used? ◁
}
```

258* Procedure *unbucket* goes through the buckets and adds nonempty lists to the stack, using the collating sequence specified in the *collate* array. The parameter to *unbucket* tells the current depth in the buckets. Any two sequences that agree in their first 255 character positions are regarded as identical.

```
#define infinity 255      ▷ ∞ (approximately) ◁
  static void unbucket(      ▷ empties buckets having depth d ◁
      eight_bits d)
  {
    int c;      ▷ index into bucket; cannot be a simple char because of sign comparison below ◁
    for (c ← 100 + 128; c ≥ 0; c−−)
      if (bucket[collate[c]]) {
        if (sort_ptr ≥ scrap_info_end) overflow(_("sorting"));
        sort_ptr++;
        if (sort_ptr > max_sort_ptr) max_sort_ptr ← sort_ptr;
        if (c ≡ 0) sort_ptr→depth ← infinity;
        else sort_ptr→depth ← d;
        sort_ptr→head ← bucket[collate[c]]; bucket[collate[c]] ← Λ;
      }
  }
```

270* Because on some systems the difference between two pointers is a **ptrdiff_t** rather than an **int**, we use `%td` to print these quantities.

```
  void print_stats(void)
  {
    puts(_("\nMemory␣usage␣statistics:"));
    printf(_("%td␣names␣(out␣of␣%ld)\n"), (ptrdiff_t)(name_ptr − name_dir), (long) max_names);
    printf(_("%td␣cross-references␣(out␣of␣%ld)\n"), (ptrdiff_t)(xref_ptr − xmem), (long) max_refs);
    printf(_("%td␣bytes␣(out␣of␣%ld)\n"), (ptrdiff_t)(byte_ptr − byte_mem), (long) max_bytes);
    puts(_("Parsing:"));
    printf(_("%td␣scraps␣(out␣of␣%ld)\n"), (ptrdiff_t)(max_scr_ptr − scrap_info), (long) max_scraps);
    printf(_("%td␣texts␣(out␣of␣%ld)\n"), (ptrdiff_t)(max_text_ptr − tok_start), (long) max_texts);
    printf(_("%td␣tokens␣(out␣of␣%ld)\n"), (ptrdiff_t)(max_tok_ptr − tok_mem), (long) max_toks);
    printf(_("%td␣levels␣(out␣of␣%ld)\n"), (ptrdiff_t)(max_stack_ptr − stack), (long) stack_size);
    puts(_("Sorting:"));
    printf(_("%td␣levels␣(out␣of␣%ld)\n"), (ptrdiff_t)(max_sort_ptr − scrap_info), (long) max_scraps);
  }
```

271*. Extensions to `CWEB`. The following sections introduce new or improved features that have been created by numerous contributors over the course of a quarter century.

Care has been taken to keep the original section numbering intact, so this new material should nicely integrate with the original "**271. Index**."

272.* Formatting alternatives. `CWEAVE` indents declarations after old-style function definitions and long parameter lists of modern function definitions. With the `-i` option they will come out flush left.

#define *indent_param_decl* *flags*[`'i'`] ▷ should formal parameter declarations be indented? ◁

⟨ Set initial values 24 ⟩ +≡
  indent_param_decl ← *true*;

273.* The original manual described the `-o` option for `CWEAVE`, but this was not yet present. Here is a simple implementation. The purpose is to suppress the extra space between local variable declarations and the first statement in a function block.

#define *order_decl_stmt* *flags*[`'o'`] ▷ should declarations and statements be separated? ◁

⟨ Set initial values 24 ⟩ +≡
  order_decl_stmt ← *true*;

274.* Output file update. Most C projects are controlled by a `Makefile` that automatically takes care of the temporal dependecies between the different source modules. It may be convenient that `CWEB` doesn't create new output for all existing files, when there are only changes to some of them. Thus the `make` process will only recompile those modules where necessary. You can activate this feature with the '`+c`' command-line option. The idea and basic implementation of this mechanism can be found in the program `NUWEB` by Preston Briggs, to whom credit is due.

```
⟨ Update the result when it has changed 274* ⟩ ≡
  {
    if ((tex_file ← fopen(tex_file_name, "r")) ≠ Λ) {
      boolean comparison ← false;
      if ((check_file ← fopen(check_file_name, "r")) ≡ Λ)
        fatal(_("!␣Cannot␣open␣output␣file␣"), check_file_name);
      ⟨ Compare the temporary output to the previous output 275* ⟩
      fclose(tex_file); tex_file ← Λ; fclose(check_file); check_file ← Λ;
      ⟨ Take appropriate action depending on the comparison 276* ⟩
    }
    else rename(check_file_name, tex_file_name);      ▷ This was the first run ◁
    strcpy(check_file_name, "");      ▷ We want to get rid of the temporary file ◁
  }
```

This code is used in section 247*.

275.* We hope that this runs fast on most systems.

```
⟨ Compare the temporary output to the previous output 275* ⟩ ≡
  do {
    char x[BUFSIZ], y[BUFSIZ];
    int x_size ← fread(x, sizeof(char), BUFSIZ, tex_file);
    int y_size ← fread(y, sizeof(char), BUFSIZ, check_file);
    comparison ← (x_size ≡ y_size) ∧ ¬memcmp(x, y, x_size);
  } while (comparison ∧ ¬feof(tex_file) ∧ ¬feof(check_file));
```

This code is used in section 274*.

276.* Note the superfluous call to *remove* before *rename*. We're using it to get around a bug in some implementations of *rename*.

```
⟨ Take appropriate action depending on the comparison 276* ⟩ ≡
  if (comparison) remove(check_file_name);      ▷ The output remains untouched ◁
  else {
    remove(tex_file_name); rename(check_file_name, tex_file_name);
  }
```

This code is used in section 274*.

277*. Print "version" information. Don't do this at home, kids! Push our local macro to the variable in `COMMON` for printing the *banner* and the *versionstring* from there.

#define *max_banner* 50

⟨ Common code for `CWEAVE` and `CTANGLE` 3* ⟩ +≡
  extern char *cb_banner*[ ];

278*. ⟨ Set initial values 24 ⟩ +≡
  strncpy(*cb_banner*, *banner*, *max_banner* − 1);

279*. Index. If you have read and understood the code for Phase III above, you know what is in this index and how it got here. All sections in which an identifier is used are listed with that identifier, except that reserved words are indexed only when they appear in format definitions, and the appearances of identifiers in section names are not indexed. Underlined entries correspond to where the identifier was declared. Error messages, control sequences put into the output, and a few other things like "recursion" are indexed here too.

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 25, 30, 57, 59, 62, 63, 64, 66, 70, 74, 79, 82, 89, 94, 99, 101, 102, 103, 110, 111, 115, 116, 128, 138, 139, 143, 153, 156, 184, 190, 191, 192, 197, 199, 203, 211, 224, 225, 226, 227, 232, 236, 237, 240, 241, 247, 258, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279.