The CWEAVE processor

(Version 4.2 [TEX Live])

March 17, 2021 at 13:07

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␣4.2"   ▷ 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 85* ⟩
  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 ≡ 2 ∧ ¬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 bool boolean;
  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 269*.

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.

```
#define _(S) gettext(S)
```

⟨ Include files 4* ⟩ ≡

```
#ifndef HAVE_GETTEXT
#define HAVE_GETTEXT 0
#endif
#if HAVE_GETTEXT
#include <libintl.h>
#else
#define gettext(A) A
#endif
#include <ctype.h>    ▷ definition of isalpha, isdigit and so on ◁
#include <stdbool.h>    ▷ definition of bool, true and false ◁
#include <stddef.h>    ▷ definition of ptrdiff_t ◁
#include <stdint.h>    ▷ definition of uint8_t and uint16_t ◁
#include <stdlib.h>    ▷ definition of getenv and exit ◁
#include <stdio.h>    ▷ definition of printf and friends ◁
#include <string.h>    ▷ definition of strlen, strcmp and so on ◁
```

This code is used in section 1*.

5* Code related to the character set:

```
#define and_and 04    ▷ '&&'; corresponds to MIT's ∧ ◁
#define lt_lt 020    ▷ '<<'; corresponds to MIT's ⊂ ◁
#define gt_gt 021    ▷ '>>'; corresponds to MIT's ⊃ ◁
#define plus_plus 013    ▷ '++'; corresponds to MIT's ↑ ◁
#define minus_minus 01    ▷ '--'; corresponds to MIT's ↓ ◁
#define minus_gt 031    ▷ '->'; corresponds to MIT's → ◁
#define non_eq 032    ▷ '!='; corresponds to MIT's ≠ ◁
#define lt_eq 034    ▷ '<='; corresponds to MIT's ≤ ◁
#define gt_eq 035    ▷ '>='; corresponds to MIT's ≥ ◁
#define eq_eq 036    ▷ '=='; corresponds to MIT's ≡ ◁
#define or_or 037    ▷ '||'; corresponds to MIT's ∨ ◁
#define dot_dot_dot 016    ▷ '...'; corresponds to MIT's ϖ ◁
#define colon_colon 06    ▷ '::'; corresponds to MIT's ∈ ◁
#define period_ast 026    ▷ '.*'; corresponds to MIT's ⊗ ◁
#define minus_gt_ast 027    ▷ '->*'; corresponds to MIT's ↳ ◁
```

⟨ 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((eight_bits) c) ∧ ((eight_bits) c < °200))
#define xisdigit(c) (isdigit((eight_bits) c) ∧ ((eight_bits) c < °200))
#define xisspace(c) (isspace((eight_bits) c) ∧ ((eight_bits) c < °200))
#define xislower(c) (islower((eight_bits) c) ∧ ((eight_bits) c < °200))
#define xisupper(c) (isupper((eight_bits) c) ∧ ((eight_bits) c < °200))
#define xisxdigit(c) (isxdigit((eight_bits) c) ∧ ((eight_bits) c < °200))
```

⟨ 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*, 33, 40, 45, 61, 65, 67, 79, 82, 86, 91, 94, 105, 113*, 116, 119, 173, 181, 186, 193, 202, 206, 220, 227, 236, 240, 251, and 260.

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 *, char);
    ▷ looks up a string in the identifier table ◁
  extern name_pointer section_lookup(char *, char *, int);  ▷ 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_stats flags['s'] ▷ should statistics be printed at end of run? ◁
#define show_happiness flags['h'] ▷ should lack of errors be announced? ◁
#define temporary_output flags['t'] ▷ should temporary output take precedence? ◁
#define make_xrefs flags['x'] ▷ should cross references be output? ◁
⟨ 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 putxchar putchar
#define term_write(a, b) fflush(stdout), fwrite(a, sizeof(char), b, stdout)
#define C_printf(c, a) fprintf(C_file, c, a)
#define C_putc(c) putc(c, C_file) ▷ isn't C wonderfully consistent? ◁
⟨ 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 were sufficient in the original WEB to handle TEX, so they should be sufficient for most applications of CWEB.

```
#define max_bytes 1000000 ▷ the number of bytes in identifiers, index entries, and section names ◁
#define max_toks 1000000 ▷ number of bytes in compressed C code ◁
#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 ◁
#define max_texts 10239 ▷ number of replacement texts, must be less than 10240 ◁
#define longest_name 10000 ▷ file and section names and section texts shouldn't be longer than this ◁
#define stack_size 500 ▷ number of simultaneous levels of macro expansion ◁
#define buf_size 1000 ▷ maximum length of input line, plus one ◁
#define long_buf_size (buf_size + longest_name) ▷ for CWEAVE ◁
```

18*. End of COMMON interface.

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 100 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) ((p + 1)→byte_start ≡ (p)→byte_start + 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);
```

54* 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 54* ⟩ ≡
  {
    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') {      ▷ wide character constant ◁
      if (delim ≡ 'u' ∧ *loc ≡ '8') {
        *++id_loc ← *loc++;
      }
      delim ← *loc++; *++id_loc ← delim;
    }
    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 55*.

55* 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 55* ⟩ ≡
  {
    c ← *loc++;
    switch (ccode[(eight_bits) c]) {
    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 xref_roman: case xref_wildcard: case xref_typewriter: case noop: case TEX_string:
      c ← ccode[(eight_bits) c]; skip_restricted( ); return c;
    case section_name: ⟨ Scan the section name and make cur_section point to it 56 ⟩
    case verbatim: ⟨ Scan a verbatim string 62* ⟩
    case ord: ⟨ Get a string 54* ⟩
    default: return ccode[(eight_bits) c];
    }
  }
```

This code is used in section 44.

58* ⟨ Put section name into *section_text* 58* ⟩ ≡

```
  k ← section_text;
  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 59* ⟩
    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 56.

59* ⟨ If end of name or erroneous control code, **break** 59* ⟩ ≡

```
if (c ≡ '@') {
  c ← *(loc + 1);
  if (c ≡ '>') {
    loc += 2; break;
  }
  if (ccode[(eight_bits) 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 58*.

60* 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"));
  }
}
```

62* 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 62* ⟩ ≡

```
{
  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 55*.

66* ⟨ Store cross-reference data for the current section 66* ⟩ ≡

```
{
  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", section_count); update_terminal;      ▷ print a progress report ◁
  }
  ⟨ Store cross-references in the TEX part of a section 70* ⟩
  ⟨ Store cross-references in the definition part of a section 73 ⟩
  ⟨ Store cross-references in the C part of a section 76 ⟩
  if (changed_section[section_count]) change_exists ← true;
}
```

This code is used in section 64.

70* 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 70* ⟩ ≡

```
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 "@" 71 ⟩
      new_xref(id_lookup(id_first, id_loc, next_control − identifier));
    }
    break;
  }
  if (next_control ≥ format_code) break;
}
```

This code is used in section 66*.

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

⟨ Process simple format in limbo 75* ⟩ ≡

```
{
  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.

78* 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 (cur_xref→num ≡ file_flag) {
      an_output ← true; cur_xref ← cur_xref→xlink;
    }
    else an_output ← false;
    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);
  }
}
```

85* 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 85* ⟩ ≡
  out_ptr ← out_buf + 1; out_line ← 1; active_file ← tex_file; *out_ptr ← 'c'; tex_puts("\\input␣");
  tex_printf(use_language); tex_puts("cwebma");
```

This code is used in section 2*.

90* 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 90* ⟩ ≡
  {
    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 89.

```
95* static void copy_limbo(void)
  {
    char c;
    while (true) {
      if (loc > limit ∧ (finish_line( ), get_line( ) ≡ false)) return;
      *(limit + 1) ← '@';
      while (*loc ≠ '@') out(*(loc++));
      if (loc++ ≤ limit) {
        c ← *loc++;
        if (ccode[(eight_bits) c] ≡ new_section) break;
        switch (ccode[(eight_bits) c]) {
        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('@');
        }
      }
    }
  }
```

97* 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 98* ⟩
    if (phase ≡ 2) {
      if (ishigh(c)) app_tok(quoted_char);
      app_tok(c);
    }
    ⟨ Copy special things when c ≡ '@', '\\' 99* ⟩
    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 100 ⟩
}
```

98.* ⟨ Check for end of comment 98* ⟩ ≡

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

This code is used in section 97*.

99.* ⟨ Copy special things when c ≡ '@', '\\' 99* ⟩ ≡

```
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 97*.

112* 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 '| ... |' ◁
#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/id_flag) {
        case 1: printf("\\\\{"); print_id((name_dir + r)); printf("}"); break; ▷ id_flag ◁
        case 2: printf("\\&{"); print_id((name_dir + r)); printf("}"); break; ▷ res_flag ◁
        case 3: printf("<"); print_section_name((name_dir + r)); printf(">"); break;
            ▷ section_flag ◁
        case 4: printf("[[%d]]", r); break; ▷ tok_flag ◁
        case 5: printf("|[[%d]]|", r); break; ▷ inner_tok_flag ◁
        default: ⟨ Print token r in symbolic form 114 ⟩
        }
      }
    update_terminal;
  }
#endif
```

113* ⟨ Predeclaration of procedures 8* ⟩ +≡

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

125*. 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* macro will cause the appropriate action to be performed, followed by **goto** *found*.

```
⟨ Cases for exp 125* ⟩ ≡
  if (cat1 ≡ lbrace ∨ cat1 ≡ int_like ∨ cat1 ≡ decl) {
    make_underlined(pp); big_app1(pp);
    if (indent_param_decl) {
      big_app(indent); app(indent);
    }
    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) squash(pp + 3, 1, base, 0, 5);
  else if (cat1 ≡ cast ∧ cat2 ≡ colon) squash(pp + 2, 1, 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) squash(pp, 1, stmt, −1, 8);
  else if (cat1 ≡ lpar ∧ cat2 ≡ rpar ∧ (cat3 ≡ const_like ∨ cat3 ≡ case_like)) {
    big_app1(pp + 2); big_app(' '); big_app1(pp + 3); reduce(pp + 2, 2, rpar, 0, 9);
  }
  else if (cat1 ≡ cast ∧ (cat2 ≡ const_like ∨ cat2 ≡ case_like)) {
    big_app1(pp + 1); big_app(' '); big_app1(pp + 2); reduce(pp + 1, 2, cast, 0, 9);
  }
  else if (cat1 ≡ exp ∨ cat1 ≡ cast) squash(pp, 2, exp, −2, 10);
```

This code is used in section 118.

135*. ⟨ Cases for *decl_head* 135* ⟩ ≡

```
if (cat1 ≡ comma) {
  big_app2(pp); big_app('␣'); reduce(pp, 2, decl_head, −1, 33);
}
else if (cat1 ≡ ubinop) {
  big_app1(pp); big_app('{'); big_app1(pp + 1); big_app('}'); reduce(pp, 2, decl_head, −1, 34);
}
else if (cat1 ≡ exp ∧ cat2 ≠ lpar ∧ 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) {
  big_app1(pp);
  if (indent_param_decl) {
    big_app(indent); app(indent);
  }
  reduce(pp, 1, fn_decl, 0, 38);
}
else if (cat1 ≡ semi) squash(pp, 2, decl, −1, 39);
```

This code is used in section 118.

136*. ⟨ Cases for *decl* 136* ⟩ ≡

```
if (cat1 ≡ decl) {
  big_app1(pp); big_app(force); big_app1(pp + 1); reduce(pp, 2, decl, −1, 40);
}
else if (cat1 ≡ stmt ∨ cat1 ≡ function) {
  big_app1(pp);
  if (order_decl_stmt) big_app(big_force);
  else big_app(force);
  big_app1(pp + 1); reduce(pp, 2, cat1, −1, 41);
}
```

This code is used in section 118.

140*. ⟨ Cases for *fn_decl* 140* ⟩ ≡

```
if (cat1 ≡ decl) {
  big_app1(pp); big_app(force); big_app1(pp + 1); 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);
}
```

This code is used in section 118.

176* 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 176* ⟩ ≡
  while (true) {
    ⟨ Make sure the entries pp through pp + 3 of cat are defined 177 ⟩
    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 118 ⟩
  }
```

This code is used in section 180.

182* 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 182* ⟩ ≡
  {
    ⟨ If semi-tracing, show the irreducible scraps 183* ⟩
    for (j ← scrap_base; j ≤ lo_ptr; j++) {
      if (j ≠ scrap_base) app('␣');
      if (j→mathness % 4 ≡ yes_math) app('$');
      app1(j);
      if (j→mathness/4 ≡ yes_math) app('$');
      if (tok_ptr + 6 > tok_mem_end) overflow(_("token"));
    }
    freeze_text; return text_ptr − 1;
  }
```

This code is used in section 180.

183* ⟨ If semi-tracing, show the irreducible scraps 183* ⟩ ≡

```
if (lo_ptr > scrap_base ∧ tracing ≡ 1) {
  printf(_("\nIrreducible␣scrap␣sequence␣in␣section␣%d:"), section_count); mark_harmless;
  for (j ← scrap_base; j ≤ lo_ptr; j++) {
    printf("␣"); print_cat(j→cat);
  }
}
```

This code is used in section 182*.

184.* ⟨ If tracing, print an indication of where we are 184* ⟩ ≡

```
if (tracing ≡ 2) {
  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 180.

189.* ⟨ Make sure that there is room for the new scraps, tokens, and texts 189* ⟩ ≡

```
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 188 and 197.

191.* The following code must use *app_tok* instead of *app* in order to protect against overflow. Note that *tok_ptr* + 1 ≤ *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 191* ⟩ ≡

```
count ← −1;
if (next_control ≡ constant) app_str("\\T{");
else if (next_control ≡ string) {
  count ← 20; app_str("\\.{");
}
else app_str("\\vb{");
while (id_first < id_loc) {
  if (count ≡ 0) {      ▷ insert a discretionary break in a long string ◁
    app_str("}\\)\\.{"); count ← 20;
  }
  if ((eight_bits)(∗id_first) > °177) {
    app_tok(quoted_char); app_tok((eight_bits)(∗id_first++));
  }
  else {
    switch (∗id_first) {
    case '␣': case '\\': case '#': case '%': case '$': case '^': case '{': case '}': case '~':
      case '&': case '_': app('\\'); break;
    case '@':
      if (∗(id_first + 1) ≡ '@') id_first++;
      else err_print(_("!␣Double␣@␣should␣be␣used␣in␣strings"));
    }
    app_tok(∗id_first++);
  }
  count−−;
}
app('}'); app_scrap(exp, maybe_math);
```

This code is used in section 188.

195* 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;  ▷ holds original value of scrap_base ◁
  save_base ← 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;
}
```

203*
```
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);
}
```

216* ⟨ Skip next character, give error if not '@' 216* ⟩ ≡
```
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 215.

217*. 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 217* ⟩ ≡
  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 218* ⟩
    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 215.

218*. ⟨ Copy a quoted character into the buffer 218* ⟩ ≡

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

This code is used in section 217*.

219* 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)
{
  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 222 ⟩
}
```

224* 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 224* ⟩ ≡
  do {
    next_control ← copy_TEX( );
    switch (next_control) {
    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 222.

228*. 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 228* ⟩ ≡
  {
    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('$'); app_cur_id(false);
      if (*loc ≡ '(')
      reswitch:
        switch (next_control ← get_next()) {
        case '(': case ',': app(next_control); goto reswitch;
        case identifier: app_cur_id(false); goto reswitch;
        case ')': app(next_control); next_control ← get_next(); break;
        default: err_print(_("! Improper macro definition")); break;
        }
      else next_control ← get_next();
      app_str("$ "); app(break_space); app_scrap(dead, no_math);
        ▷ scrap won't take part in the parsing ◁
    }
  }
```

This code is used in section 225.

229*. ⟨ Start a format definition 229* ⟩ ≡

```
  {
    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(' '); 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 225.

232*. 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 232* ⟩ ≡
  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 231.

233*. ⟨ Emit the scrap for a section name if present 233* ⟩ ≡

```
  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 231.

239* 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"); active_file ← tex_file;
  }
  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 242 ⟩
      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 244 ⟩
    ⟨ Sort and output the index 252 ⟩
    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 261 ⟩
    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 ← Λ; ⟨ Update the result when it has changed 266* ⟩
  if (show_happiness) {
    if (show_progress) new_line;
    fputs(_("Done."), stdout);
  }
  check_complete();      ▷ was all of the change file used? ◁
}
```

250* 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;
      }
  }
```

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

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

263* 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 "**263. Index**."

264* Formatting alternatives. `CWEAVE` indents declarations after old-style function definitions. With the `-i` option they will come out flush left. You won't see any difference if you use ANSI-style function definitions.

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

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

265* 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*;

266* 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 is suitable 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. 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 266* ⟩ ≡
  if ((tex_file ← fopen(tex_file_name, "r")) ≠ Λ) {
    char x[BUFSIZ], y[BUFSIZ];
    int x_size, y_size, comparison ← false;
    if ((check_file ← fopen(check_file_name, "r")) ≡ Λ)
      fatal(_("!␣Cannot␣open␣output␣file␣"), check_file_name);
    if (temporary_output) ⟨ Compare the temporary output to the previous output 267* ⟩
    fclose(tex_file); tex_file ← Λ; fclose(check_file); check_file ← Λ;
    ⟨ Take appropriate action depending on the comparison 268* ⟩
  }
  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 239*.

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

```
⟨ Compare the temporary output to the previous output 267* ⟩ ≡
  do {
    x_size ← fread(x, 1, BUFSIZ, tex_file); y_size ← fread(y, 1, BUFSIZ, check_file);
    comparison ← (x_size ≡ y_size);      ▷ Do not merge these statements! ◁
    if (comparison) comparison ← ¬memcmp(x, y, x_size);
  } while (comparison ∧ ¬feof(tex_file) ∧ ¬feof(check_file));
```

This code is used in section 266*.

268* 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 268* ⟩ ≡
  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 266*.

269* Put "version" information in a single spot. 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[];
```

270* ⟨ Set initial values 24 ⟩ +≡

```
strncpy(cb_banner, banner, max_banner − 1);
```

271* 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, 25, 54, 55, 58, 59, 60, 62, 66, 70, 75, 78, 85, 90, 95, 97, 98, 99, 112, 113, 125, 135, 136, 140, 176, 182, 183, 184, 189, 191, 195, 203, 216, 217, 218, 219, 224, 228, 229, 232, 233, 239, 250, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271.

⟨ Check if next token is **include** 49 ⟩ Used in section 47.
⟨ Check if we're at the end of a preprocessor command 50 ⟩ Used in section 44.
⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition 232* ⟩ Used in section 231.
⟨ Clear *bal* and **return** 100 ⟩ Used in section 97*.
⟨ Combine the irreducible scraps that remain 182* ⟩ Used in section 180.
⟨ Common code for `CWEAVE` and `CTANGLE` 3*, 5*, 6*, 7*, 9*, 10*, 12*, 14*, 15*, 269* ⟩ Used in section 1*.
⟨ Compare the temporary output to the previous output 267* ⟩ Used in section 266*.
⟨ Compress two-symbol operator 51 ⟩ Used in section 44.
⟨ Copy a quoted character into the buffer 218* ⟩ Used in section 217*.
⟨ Copy special things when $c \equiv$ `'@'`, `'\\'` 99* ⟩ Used in section 97*.
⟨ Copy the C text into the *buffer* array 217* ⟩ Used in section 215.
⟨ Do the first pass of sorting 244 ⟩ Used in section 239*.
⟨ Emit the scrap for a section name if present 233* ⟩ Used in section 231.
⟨ Get a constant 53 ⟩ Used in section 44.
⟨ Get a string 54* ⟩ Used in sections 44 and 55*.
⟨ Get an identifier 52 ⟩ Used in section 44.
⟨ Get control code and possible section name 55* ⟩ Used in section 44.
⟨ If end of name or erroneous control code, **break** 59* ⟩ Used in section 58*.
⟨ If semi-tracing, show the irreducible scraps 183* ⟩ Used in section 182*.
⟨ If tracing, print an indication of where we are 184* ⟩ Used in section 180.
⟨ Include files 4* ⟩ Used in section 1*.
⟨ Insert new cross-reference at q, not at beginning of list 124 ⟩ Used in section 123.
⟨ Invert the cross-reference list at *cur_name*, making *cur_xref* the head 258 ⟩ Used in section 256.
⟨ Look ahead for strongest line break, **goto** *reswitch* 212 ⟩ Used in section 211.
⟨ Make sure that there is room for the new scraps, tokens, and texts 189* ⟩ Used in sections 188 and 197.
⟨ Make sure the entries *pp* through $pp + 3$ of *cat* are defined 177 ⟩ Used in section 176*.
⟨ Match a production at *pp*, or increase *pp* if there is no match 118 ⟩ Used in section 176*.
⟨ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 211 ⟩ Used in section 209.
⟨ Output a section name 214 ⟩ Used in section 209.
⟨ Output all the section names 261 ⟩ Used in section 239*.
⟨ Output all the section numbers on the reference list *cur_xref* 237 ⟩ Used in section 235.
⟨ Output an identifier 210 ⟩ Used in section 209.
⟨ Output index entries for the list at *sort_ptr* 254 ⟩ Used in section 252.
⟨ Output saved *indent* or *outdent* tokens 213 ⟩ Used in sections 209 and 212.
⟨ Output the code for the beginning of a new section 223 ⟩ Used in section 222.
⟨ Output the code for the end of a section 238 ⟩ Used in section 222.
⟨ Output the cross-references at *cur_name* 256 ⟩ Used in section 254.
⟨ Output the name at *cur_name* 255 ⟩ Used in section 254.
⟨ Output the text of the section name 215 ⟩ Used in section 214.
⟨ Predeclaration of procedures 8*, 11*, 13*, 16*, 25*, 33, 40, 45, 61, 65, 67, 79, 82, 86, 91, 94, 105, 113*, 116, 119, 173, 181, 186, 193, 202, 206, 220, 227, 236, 240, 251, 260 ⟩ Used in section 1*.
⟨ Print a snapshot of the scrap list if debugging 179 ⟩ Used in sections 174 and 175.
⟨ Print error messages about unused or undefined section names 80 ⟩ Used in section 64.
⟨ Print token r in symbolic form 114 ⟩ Used in section 112*.
⟨ Print warning message, break the line, **return** 90* ⟩ Used in section 89.
⟨ Private variables 21, 23, 30, 37, 43, 46, 48, 63, 72, 77, 81, 102, 110, 115, 178, 200, 205, 221, 230, 241, 243, 246, 248, 257 ⟩ Used in section 1*.
⟨ Process a format definition 74 ⟩ Used in section 73.
⟨ Process simple format in limbo 75* ⟩ Used in section 41.
⟨ Put section name into *section_text* 58* ⟩ Used in section 56.
⟨ Raise preprocessor flag 47 ⟩ Used in section 44.

⟨ Reduce the scraps using the productions until no more rules apply 176* ⟩ Used in section 180.
⟨ Replace "@@" by "@" 71 ⟩ Used in sections 68 and 70*.
⟨ Rest of *trans_plus* union 245 ⟩ Used in section 109.
⟨ Scan a verbatim string 62* ⟩ Used in section 55*.
⟨ Scan the section name and make *cur_section* point to it 56 ⟩ Used in section 55*.
⟨ Set initial values 24, 31, 38, 57, 88, 103, 111, 152, 196, 201, 247, 249, 264*, 265*, 270* ⟩ Used in section 2*.
⟨ Show cross-references to this section 234 ⟩ Used in section 222.
⟨ Skip next character, give error if not '@' 216* ⟩ Used in section 215.
⟨ Sort and output the index 252 ⟩ Used in section 239*.
⟨ Special control codes for debugging 39 ⟩ Used in section 38.
⟨ Split the list at *sort_ptr* into further lists 253 ⟩ Used in section 252.
⟨ Start TEX output 85* ⟩ Used in section 2*.
⟨ Start a format definition 229* ⟩ Used in section 225.
⟨ Start a macro definition 228* ⟩ Used in section 225.
⟨ Store all the reserved words 34 ⟩ Used in section 2*.
⟨ Store cross-reference data for the current section 66* ⟩ Used in section 64.
⟨ Store cross-references in the C part of a section 76 ⟩ Used in section 66*.
⟨ Store cross-references in the TEX part of a section 70* ⟩ Used in section 66*.
⟨ Store cross-references in the definition part of a section 73 ⟩ Used in section 66*.
⟨ Take appropriate action depending on the comparison 268* ⟩ Used in section 266*.
⟨ Tell about changed sections 242 ⟩ Used in section 239*.
⟨ Translate the C part of the current section 231 ⟩ Used in section 222.
⟨ Translate the TEX part of the current section 224* ⟩ Used in section 222.
⟨ Translate the current section 222 ⟩ Used in section 219*.
⟨ Translate the definition part of the current section 225 ⟩ Used in section 222.
⟨ Typedef declarations 22, 29, 109, 199 ⟩ Used in section 1*.
⟨ Update the result when it has changed 266* ⟩ Used in section 239*.