## Contents

## Contents of common

## Sections of common

**1. Introduction.** This file contains code common to both CTANGLE and CWEAVE, which roughly concerns the following problems: character uniformity, input routines, error handling and parsing of command line. We have tried to concentrate in this file all the system dependencies, so as to maximize portability.

In the texts below we will sometimes use CWEB to refer to either of the two component programs, if no confusion can arise.

The file begins with a few basic definitions.

⟨ Include files 5 ⟩
⟨ Preprocessor definitions ⟩
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩
⟨ Other definitions 3 ⟩
⟨ Predeclaration of procedures 33 ⟩

**2.** In certain cases CTANGLE and CWEAVE should do almost, but not quite, the same thing. In these cases we've written common code for both, differentiating between the two by means of the global variable *program*.

```
#define ctangle 0
#define cweave 1
```

⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ ≡

```
typedef short boolean;
boolean program;     /* CWEAVE or CTANGLE? */
```

See also sections 7, 10, 20, 27, 29, 32, 56, 67, and 77

This code is used in section 1

**3.** CWEAVE 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, and finally it sorts and outputs the index. Similarly, CTANGLE operates in two phases. The global variable *phase* tells which phase we are in.

⟨ Other definitions 3 ⟩ ≡

```
int phase;     /* which phase are we in? */
```

See also section 11

This code is used in section 1

**4.** There's an initialization procedure that gets both CTANGLE and CWEAVE off to a good start. We will fill in the details of this procedure later.

```
void common_init()
{
  ⟨ Initialize pointers 30 ⟩;
  ⟨ Set the default options common to CTANGLE and CWEAVE 68 ⟩;
  ⟨ Scan arguments and open output files 78 ⟩;
}
```

**5. The character set.** CWEB uses the conventions of C programs found in the standard ctype.h header file.

⟨ Include files 5 ⟩ ≡
#**include** <ctype.h>

See also sections 8 and 22

This code is used in section 1

**6.** A few character pairs are encoded internally as single characters, using the definitions below. These definitions are consistent with an extension of ASCII code originally developed at MIT and explained in Appendix C of *The TEXbook*; thus, users who have such a character set can type things like ≠ and char'4 instead of != and &&. (However, their files will not be too portable until more people adopt the extended code.)

If the character set is not ASCII, the definitions given here may conflict with existing characters; in such cases, other arbitrary codes should be substituted. The indexes to CTANGLE and CWEAVE mention every case where similar codes may have to be changed in order to avoid character conflicts. Look for the entry "ASCII code dependencies" in those indexes.

```
#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 not_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 ϖ */
#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 ↰ */
```

**7. Input routines.** The lowest level of input to the CWEB programs is performed by *input_ln*, which must be told which file to read from. The return value of *input_ln* is 1 if the read is successful and 0 if not (generally this means the file has ended). The conventions of TEX are followed; i.e., the characters of the next line of the file are copied into the *buffer* array, and the global variable *limit* is set to the first unoccupied position. Trailing blanks are ignored. The value of *limit* must be strictly less than *buf_size*, so that *buffer*[*buf_size* − 1] is never filled.

Since *buf_size* is strictly less than *long_buf_size*, some of CWEB's routines use the fact that it is safe to refer to ∗(*limit* + 2) without overstepping the bounds of the array.

```
#define buf_size 100 /* for CWEAVE and CTANGLE */
#define long_buf_size 500 /* for CWEAVE */
#define xisspace(c) (isspace(c) ∧ ((unsigned char) c < °200))
#define xisupper(c) (isupper(c) ∧ ((unsigned char) c < °200))
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  char buffer[long_buf_size]; /* where each line of input goes */
  char *buffer_end ← buffer + buf_size − 2; /* end of buffer */
  char *limit ← buffer; /* points to the last character in the buffer */
  char *loc ← buffer; /* points to the next character to be read from the buffer */
```

**8.** ⟨ Include files 5 ⟩ +≡

```
#include <stdio.h>
```

**9.** In the unlikely event that your standard I/O library does not support *feof*, *getc* and *ungetc* you may have to change things here.

```
int input_ln(fp) /* copies a line into buffer or returns 0 */
    FILE *fp; /* what file to read from */
{
  register int c ← EOF; /* character read; initialized so some compilers won't complain */
  register char *k; /* where next character goes */
  if (feof(fp)) return (0); /* we have hit end-of-file */
  limit ← k ← buffer; /* beginning of buffer */
  while (k ≤ buffer_end ∧ (c ← getc(fp)) ≠ EOF ∧ c ≠ '\n')
    if ((*(k++) ← c) ≠ '␣') limit ← k;
  if (k > buffer_end)
    if ((c ← getc(fp)) ≠ EOF ∧ c ≠ '\n') {
      ungetc(c, fp);
      loc ← buffer;
      err_print("!␣Input␣line␣too␣long");
    }
  if (c ≡ EOF ∧ limit ≡ buffer) return (0); /* there was nothing after the last newline */
  return (1);
}
```

**10.** Now comes the problem of deciding which file to read from next. Recall that the actual text that CWEB should process comes from two streams: a *web_file*, which can contain possibly nested include commands @i, and a *change_file*, which might also contain includes. The *web_file* together with the currently open include files form a stack *file*, whose names are stored in a parallel stack *file_name*. The boolean *changing* tells whether or not we're reading from the *change_file*.

The line number of each open file is also kept for error reporting and for the benefit of CTANGLE.

```
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 60
#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 */
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  int include_depth; /* current level of nesting */
  FILE *file[max_include_depth]; /* stack of non-change files */
  FILE *change_file; /* change file */
  char file_name[max_include_depth][max_file_name_length]; /* stack of non-change file names */
  char change_file_name[max_file_name_length]; /* name of change file */
  char alt_web_file_name[max_file_name_length]; /* alternate name to try */
  int line[max_include_depth]; /* number of current line in the stacked files */
  int change_line; /* number of current line in change file */
  int change_depth; /* where @y originated during a change */
  boolean input_has_ended; /* if there is no more input */
  boolean changing; /* if the current line is from change_file */
  boolean web_file_open ← 0; /* if the web file is being read */
```

**11.** When *changing* ≡ 0, the next line of *change_file* is kept in *change_buffer*, for purposes of comparison with the next line of *cur_file*. After the change file has been completely input, we set *change_limit* ← *change_buffer*, so that no further matches will be made.

Here's a shorthand expression for inequality between the two lines:

```
#define lines_dont_match
        (change_limit − change_buffer ≠ limit − buffer ∨ strncmp(buffer, change_buffer, limit − buffer))
⟨ Other definitions 3 ⟩ +≡
  char change_buffer[buf_size]; /* next line of change_file */
  char *change_limit; /* points to the last character in change_buffer */
```

**12.** Procedure *prime_the_change_buffer* sets *change_buffer* in preparation for the next matching operation. Since blank lines in the change file are not used for matching, we have (*change_limit* ≡ *change_buffer* ∧ ¬*changing*) if and only if the change file is exhausted. This procedure is called only when *changing* is 1; hence error messages will be reported correctly.

```
void prime_the_change_buffer( )
{
  change_limit ← change_buffer; /* this value is used if the change file ends */
  ⟨ Skip over comment lines in the change file; return if end of file 13 ⟩;
  ⟨ Skip to the next nonblank line; return if end of file 14 ⟩;
  ⟨ Move buffer and limit to change_buffer and change_limit 15 ⟩;
}
```

**13.** While looking for a line that begins with `@x` in the change file, we allow lines that begin with `@`, as long as they don't begin with `@y`, `@z` or `@i` (which would probably mean that the change file is fouled up).

```
⟨ Skip over comment lines in the change file; return if end of file 13 ⟩ ≡
  while (1) {
    change_line++;
    if (¬input_ln(change_file)) return;
    if (limit < buffer + 2) continue;
    if (buffer[0] ≠ '@') continue;
    if (xisupper(buffer[1])) buffer[1] ← tolower(buffer[1]);
    if (buffer[1] ≡ 'x') break;
    if (buffer[1] ≡ 'y' ∨ buffer[1] ≡ 'z' ∨ buffer[1] ≡ 'i') {
      loc ← buffer + 2;
      err_print("!␣Missing␣@x␣in␣change␣file");
    }
  }
```

This code is used in section 12

**14.** Here we are looking at lines following the `@x`.

```
⟨ Skip to the next nonblank line; return if end of file 14 ⟩ ≡
  do {
    change_line++;
    if (¬input_ln(change_file)) {
      err_print("!␣Change␣file␣ended␣after␣@x");
      return;
    }
  } while (limit ≡ buffer);
```

This code is used in section 12

**15.** ⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 15 ⟩ ≡

```
  {
    change_limit ← change_buffer − buffer + limit;
    strncpy(change_buffer, buffer, limit − buffer + 1);
  }
```

This code is used in sections 12 and 16

**16.** The following procedure is used to see if the next change entry should go into effect; it is called only when *changing* is 0. The idea is to test whether or not the current contents of *buffer* matches the current contents of *change_buffer*. If not, there's nothing more to do; but if so, a change is called for: All of the text down to the `@y` is supposed to match. An error message is issued if any discrepancy is found. Then the procedure prepares to read the next line from *change_file*.

When a match is found, the current section is marked as changed unless the first line after the `@x` and after the `@y` both start with either `'@*'` or `'@␣'` (possibly preceded by whitespace).

This procedure is called only when *buffer* < *limit*, i.e., when the current line is nonempty.

```
#define if_section_start_make_pending(b)
        { *limit ← '!';
          for (loc ← buffer; xisspace(*loc); loc++) ;
          *limit ← '␣';
          if (*loc ≡ '@' ∧ (xisspace(*(loc + 1)) ∨ *(loc + 1) ≡ '*')) change_pending ← b;
        }
void check_change( )      /* switches to change_file if the buffers match */
{
  int n ← 0;      /* the number of discrepancies found */
  if (lines_dont_match) return;
  change_pending ← 0;
  if (¬changed_section[section_count]) {
    if_section_start_make_pending(1);
    if (¬change_pending) changed_section[section_count] ← 1;
  }
  while (1) {
    changing ← 1;
    print_where ← 1;
    change_line++;
    if (¬input_ln(change_file)) {
      err_print("! Change file ended before @y");
      change_limit ← change_buffer;
      changing ← 0;
      return;
    }
    if (limit > buffer + 1 ∧ buffer[0] ≡ '@') {
      if (xisupper(buffer[1])) buffer[1] ← tolower(buffer[1]);
      ⟨ If the current line starts with @y, report any discrepancies and return 17 ⟩;
    }
    ⟨ Move buffer and limit to change_buffer and change_limit 15 ⟩;
    changing ← 0;
    cur_line++;
    while (¬input_ln(cur_file)) {      /* pop the stack or quit */
      if (include_depth ≡ 0) {
        err_print("! CWEB file ended during a change");
        input_has_ended ← 1;
        return;
      }
      include_depth−−;
      cur_line++;
    }
    if (lines_dont_match) n++;
  }
}
```

**17.** ⟨ If the current line starts with `@y`, report any discrepancies and **return** 17 ⟩ ≡

```
if (buffer[1] ≡ 'x' ∨ buffer[1] ≡ 'z') {
  loc ← buffer + 2;
  err_print("! Where is the matching @y?");
}
else if (buffer[1] ≡ 'y') {
  if (n > 0) {
    loc ← buffer + 2;
    printf("\n! Hmm... %d ", n);
    err_print("of the preceding lines failed to match");
  }
  change_depth ← include_depth;
  return;
}
```

This code is used in section 16

**18.** The *reset_input* procedure, which gets `CWEB` ready to read the user's `CWEB` input, is used at the beginning of phase one of `CTANGLE`, phases one and two of `CWEAVE`.

```
void reset_input( )
{
  limit ← buffer;
  loc ← buffer + 1;
  buffer[0] ← ' ';
  ⟨ Open input files 19 ⟩;
  include_depth ← 0;
  cur_line ← 0;
  change_line ← 0;
  change_depth ← include_depth;
  changing ← 1;
  prime_the_change_buffer( );
  changing ← ¬changing;
  limit ← buffer;
  loc ← buffer + 1;
  buffer[0] ← ' ';
  input_has_ended ← 0;
}
```

**19.** The following code opens the input files.

⟨ Open input files 19 ⟩ ≡

```
if ((web_file ← fopen(web_file_name, "r")) ≡ Λ) {
  strcpy(web_file_name, alt_web_file_name);
  if ((web_file ← fopen(web_file_name, "r")) ≡ Λ)
    fatal("! Cannot open input file ", web_file_name);
}
web_file_open ← 1;
if ((change_file ← fopen(change_file_name, "r")) ≡ Λ)
  fatal("! Cannot open change file ", change_file_name);
```

This code is used in section 18

**20.** The *get_line* procedure is called when $loc > limit$; it puts the next line of merged input into the buffer and updates the other variables appropriately. A space is placed at the right end of the line. This procedure returns $\neg input\_has\_ended$ because we often want to check the value of that variable after calling the procedure.

If we've just changed from the *cur_file* to the *change_file*, or if the *cur_file* has changed, we tell CTANGLE to print this information in the C file by means of the *print_where* flag.

```
#define max_sections 2000
        /* number of identifiers, strings, section names; must be less than 10240 */
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  typedef unsigned short sixteen_bits;
  sixteen_bits section_count;      /* the current section number */
  boolean changed_section[max_sections];      /* is the section changed? */
  boolean change_pending;
    /* if the current change is not yet recorded in changed_section[section_count] */
  boolean print_where ← 0;      /* should CTANGLE print line and file info? */
```

**21.**

```
int get_line( )      /* inputs the next line */
{
restart:
  if (changing ∧ include_depth ≡ change_depth)
    ⟨ Read from change_file and maybe turn off changing 25 ⟩;
  if (¬changing ∨ include_depth > change_depth) {
    ⟨ Read from cur_file and maybe turn on changing 24 ⟩;
    if (changing ∧ include_depth ≡ change_depth) goto restart;
  }
  loc ← buffer;
  *limit ← ' ';
  if (*buffer ≡ '@' ∧ (*(buffer + 1) ≡ 'i' ∨ *(buffer + 1) ≡ 'I')) {
    loc ← buffer + 2;
    while (loc ≤ limit ∧ (*loc ≡ ' ' ∨ *loc ≡ '\t' ∨ *loc ≡ '"')) loc++;
    if (loc ≥ limit) {
      err_print("! Include file name not given");
      goto restart;
    }
    if (include_depth ≥ max_include_depth − 1) {
      err_print("! Too many nested includes");
      goto restart;
    }
    include_depth++;      /* push input stack */
    ⟨ Try to open include file, abort push if unsuccessful, go to restart 23 ⟩;
  }
  return (¬input_has_ended);
}
```

**22.** When an `@i` line is found in the *cur_file*, we must temporarily stop reading it and start reading from the named include file. The `@i` line should give a complete file name with or without double quotes. If the environment variable `CWEBINPUTS` is set, or if the compiler flag of the same name was defined at compile time, `CWEB` will look for include files in the directory thus named, if it cannot find them in the current directory. (Colon-separated paths are not supported.) The remainder of the `@i` line after the file name is ignored.

```
#define too_long()
        {
          include_depth--;
          err_print("! Include file name too long");
          goto restart;
        }
⟨ Include files 5 ⟩ +≡
#include <stdlib.h>      /* declaration of getenv and exit */
```

**23.** ⟨ Try to open include file, abort push if unsuccessful, go to *restart* 23 ⟩ ≡

```
{
  char temp_file_name[max_file_name_length];
  char *cur_file_name_end ← cur_file_name + max_file_name_length − 1;
  char *k ← cur_file_name, *kk;
  int l;      /* length of file name */
  while (*loc ≠ '␣' ∧ *loc ≠ '\t' ∧ *loc ≠ '"' ∧ k ≤ cur_file_name_end) *k++ ← *loc++;
  if (k > cur_file_name_end) too_long( );
  *k ← '\0';
  if ((cur_file ← fopen(cur_file_name, "r")) ≠ Λ) {
    cur_line ← 0;
    print_where ← 1;
    goto restart;      /* success */
  }
  kk ← getenv("CWEBINPUTS");
  if (kk ≠ Λ) {
    if ((l ← strlen(kk)) > max_file_name_length − 2) too_long( );
    strcpy(temp_file_name, kk);
  }
  else {
#ifdef CWEBINPUTS
    if ((l ← strlen(CWEBINPUTS)) > max_file_name_length − 2) too_long( );
    strcpy(temp_file_name, CWEBINPUTS);
#else
    l ← 0;
#endif      /* CWEBINPUTS */
  }
  if (l > 0) {
    if (k + l + 2 ≥ cur_file_name_end) too_long( );
    for ( ; k ≥ cur_file_name; k−−) *(k + l + 1) ← *k;
    strcpy(cur_file_name, temp_file_name);
    cur_file_name[l] ← '/';      /* UNIX pathname separator */
    if ((cur_file ← fopen(cur_file_name, "r")) ≠ Λ) {
      cur_line ← 0;
      print_where ← 1;
      goto restart;      /* success */
    }
  }
  include_depth−−;
  err_print("!␣Cannot␣open␣include␣file");
  goto restart;
}
```

This code is used in section 21

**24.** ⟨ Read from *cur_file* and maybe turn on *changing* 24 ⟩ ≡

```
{
  cur_line++;
  while (¬input_ln(cur_file)) {      /* pop the stack or quit */
    print_where ← 1;
    if (include_depth ≡ 0) {
      input_has_ended ← 1;
      break;
    }
    else {
      fclose(cur_file);
      include_depth--;
      if (changing ∧ include_depth ≡ change_depth) break;
      cur_line++;
    }
  }
  if (¬changing ∧ ¬input_has_ended)
    if (limit − buffer ≡ change_limit − change_buffer)
      if (buffer[0] ≡ change_buffer[0])
        if (change_limit > change_buffer) check_change( );
}
```

This code is used in section 21

**25.** ⟨ Read from *change_file* and maybe turn off *changing* 25 ⟩ ≡

```
{
  change_line++;
  if (¬input_ln(change_file)) {
    err_print("! Change file ended without @z");
    buffer[0] ← '@';
    buffer[1] ← 'z';
    limit ← buffer + 2;
  }
  if (limit > buffer) {      /* check if the change has ended */
    if (change_pending) {
      if_section_start_make_pending(0);
      if (change_pending) {
        changed_section[section_count] ← 1;
        change_pending ← 0;
      }
    }
    *limit ← ' ';
    if (buffer[0] ≡ '@') {
      if (xisupper(buffer[1])) buffer[1] ← tolower(buffer[1]);
      if (buffer[1] ≡ 'x' ∨ buffer[1] ≡ 'y') {
        loc ← buffer + 2;
        err_print("! Where is the matching @z?");
      }
      else if (buffer[1] ≡ 'z') {
        prime_the_change_buffer( );
        changing ← ¬changing;
        print_where ← 1;
      }
    }
  }
}
```

This code is used in section 21

**26.** At the end of the program, we will tell the user if the change file had a line that didn't match any relevant line in *web_file*.

```
void check_complete( )
{
  if (change_limit ≠ change_buffer) {      /* changing is 0 */
    strncpy(buffer, change_buffer, change_limit − change_buffer + 1);
    limit ← buffer + (int) (change_limit − change_buffer);
    changing ← 1;
    change_depth ← include_depth;
    loc ← buffer;
    err_print("! Change file entry did not match");
  }
}
```

**27. Storage of names and strings.** Both CWEAVE and CTANGLE store identifiers, section names and other strings in a large array of **char**s, called *byte_mem*. Information about the names is kept in the array *name_dir*, whose elements are structures of type *name_info*, containing a pointer into the *byte_mem* array (the address where the name begins) and other data. A *name_pointer* variable is a pointer into *name_dir*.

```
#define max_bytes 90000
        /* the number of bytes in identifiers, index entries, and section names; must be less than 2^24 */
#define max_names 4000
        /* number of identifiers, strings, section names; must be less than 10240 */
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  typedef struct name_info {
    char *byte_start;    /* beginning of the name in byte_mem */
    ⟨ More elements of name_info structure 31 ⟩
  } name_info;    /* contains information about an identifier or section name */
  typedef name_info *name_pointer;    /* pointer into array of name_infos */
  char byte_mem[max_bytes];    /* characters of names */
  char *byte_mem_end ← byte_mem + max_bytes − 1;    /* end of byte_mem */
  name_info name_dir[max_names];    /* information about names */
  name_pointer name_dir_end ← name_dir + max_names − 1;    /* end of name_dir */
```

**28.** The actual sequence of characters in the name pointed to by a **name_pointer** $p$ appears in positions $p$→*byte_start* to $(p+1)$→*byte_start* $-1$, inclusive. The *print_id* macro prints this text on the user's terminal.

```
#define length(c) (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 */
```

**29.** The first unused position in *byte_mem* and *name_dir* is kept in *byte_ptr* and *name_ptr*, respectively. Thus we usually have *name_ptr*→*byte_start* ≡ *byte_ptr*, and certainly we want to keep *name_ptr* ≤ *name_dir_end* and *byte_ptr* ≤ *byte_mem_end*.

```
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  name_pointer name_ptr;    /* first unused position in byte_start */
  char *byte_ptr;    /* first unused position in byte_mem */
```

**30.**

```
⟨ Initialize pointers 30 ⟩ ≡
  name_dir→byte_start ← byte_ptr ← byte_mem;    /* position zero in both arrays */
  name_ptr ← name_dir + 1;    /* name_dir[0] will be used only for error recovery */
  name_ptr→byte_start ← byte_mem;    /* this makes name 0 of length zero */
```

See also sections 34 and 41

This code is used in section 4

**31.** The names of identifiers are found by computing a hash address $h$ and then looking at strings of bytes signified by the **name_pointer**s *hash*[$h$], *hash*[$h$]→*link*, *hash*[$h$]→*link*→*link*, ..., until either finding the desired name or encountering the null pointer.

```
⟨ More elements of name_info structure 31 ⟩ ≡
  struct name_info *link;
```

See also sections 40 and 55

This code is used in section 27

**32.** The hash table itself consists of *hash_size* entries of type **name_pointer**, and is updated by the *id_lookup* procedure, which finds a given identifier and returns the appropriate **name_pointer**. The matching is done by the function *names_match*, which is slightly different in CWEAVE and CTANGLE. If there is no match for the identifier, it is inserted into the table.

```
#define hash_size 353 /* should be prime */
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  typedef name_pointer *hash_pointer;
  name_pointer hash[hash_size]; /* heads of hash lists */
  hash_pointer hash_end ← hash + hash_size − 1; /* end of hash */
  hash_pointer h; /* index into hash-head array */
```

**33.**
```
⟨ Predeclaration of procedures 33 ⟩ ≡
  extern int names_match( );
```

See also sections 38, 46, 53, 57, 60, 63, 69, and 81

This code is used in section 1

**34.** Initially all the hash lists are empty.

```
⟨ Initialize pointers 30 ⟩ +≡
  for (h ← hash; h ≤ hash_end; *h++ ← Λ) ;
```

**35.** Here is the main procedure for finding identifiers:

```
name_pointer id_lookup(first, last, t) /* looks up a string in the identifier table */
    char *first; /* first character of string */
    char *last; /* last character of string plus one */
    char t; /* the ilk; used by CWEAVE only */
{
  char *i ← first; /* position in buffer */
  int h; /* hash code */
  int l; /* length of the given identifier */
  name_pointer p; /* where the identifier is being sought */
  if (last ≡ Λ)
    for (last ← first; *last ≠ '\0'; last++) ;
  l ← last − first; /* compute the length */
  ⟨ Compute the hash code h 36 ⟩;
  ⟨ Compute the name location p 37 ⟩;
  if (p ≡ name_ptr) ⟨ Enter a new name into the table at position p 39 ⟩;
  return (p);
}
```

**36.** A simple hash code is used: If the sequence of character codes is $c_1 c_2 \ldots c_n$, its hash value will be

$$(2^{n-1}c_1 + 2^{n-2}c_2 + \cdots + c_n) \bmod hash\_size.$$

```
⟨ Compute the hash code h 36 ⟩ ≡
  h ← (unsigned char) *i;
  while (++i < last) h ← (h + h + (int) ((unsigned char) *i)) % hash_size;
```

This code is used in section 35

**37.** If the identifier is new, it will be placed in position $p \leftarrow name\_ptr$, otherwise $p$ will point to its existing location.

```
⟨ Compute the name location p 37 ⟩ ≡
  p ← hash[h];
  while (p ∧ ¬names_match(p, first, l, t)) p ← p→link;
  if (p ≡ Λ) {
    p ← name_ptr;      /* the current identifier is new */
    p→link ← hash[h];
    hash[h] ← p;      /* insert p at beginning of hash list */
  }
```

This code is used in section 35

**38.** The information associated with a new identifier must be initialized in a slightly different way in `CWEAVE` than in `CTANGLE`; hence the *init_p* procedure.

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  void init_p( );
```

**39.**

```
⟨ Enter a new name into the table at position p 39 ⟩ ≡
  {
    if (byte_ptr + l > byte_mem_end) overflow("byte␣memory");
    if (name_ptr ≥ name_dir_end) overflow("name");
    strncpy(byte_ptr, first, l);
    (++name_ptr)→byte_start ← byte_ptr += l;
    if (program ≡ cweave) init_p(p, t);
  }
```

This code is used in section 35

**40.** The names of sections are stored in *byte_mem* together with the identifier names, but a hash table is not used for them because `CTANGLE` needs to be able to recognize a section name when given a prefix of that name. A conventional binary search tree is used to retrieve section names, with fields called *llink* and *rlink* (where *llink* takes the place of *link*). The root of this tree is stored in *name_dir→rlink*; this will be the only information in *name_dir*[0].

Since the space used by *rlink* has a different function for identifiers than for section names, we declare it as a **union**.

```
#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 */
⟨ More elements of name_info structure 31 ⟩ +≡
  union {
    struct name_info *Rlink;      /* right link in binary search tree for section names */
    char Ilk;      /* used by identifiers in CWEAVE only */
  } dummy;
```

**41.**

```
⟨ Initialize pointers 30 ⟩ +≡
  root ← Λ;      /* the binary search tree starts out with nothing in it */
```

**42.** If $p$ is a **name_pointer** variable, as we have seen, $p\rightarrow byte\_start$ is the beginning of the area where the name corresponding to $p$ is stored. However, if $p$ refers to a section name, the name may need to be stored in chunks, because it may "grow": a prefix of the section name may be encountered before the full name. Furthermore we need to know the length of the shortest prefix of the name that was ever encountered.

We solve this problem by inserting two extra bytes at $p\rightarrow byte\_start$, representing the length of the shortest prefix, when $p$ is a section name. Furthermore, the last byte of the name will be a blank space if $p$ is a prefix. In the latter case, the name pointer $p+1$ will allow us to access additional chunks of the name: The second chunk will begin at the name pointer $(p+1)\rightarrow link$, and if it too is a prefix (ending with blank) its *link* will point to additional chunks in the same way. Null links are represented by *name_dir*.

```
#define first_chunk(p) ((p)→byte_start + 2)
#define prefix_length(p)
        (int) ((unsigned char) *((p)→byte_start) * 256 + (unsigned char) *((p)→byte_start + 1))
#define set_prefix_length(p, m) (*((p)→byte_start) ← (m)/256, *((p)→byte_start + 1) ← (m) % 256)
  void print_section_name(p)
      name_pointer p;
  {
    char *ss, *s ← first_chunk(p);
    name_pointer q ← p + 1;
    while (p ≠ name_dir) {
      ss ← (p + 1)→byte_start − 1;
      if (*ss ≡ '␣' ∧ ss ≥ s) {
        term_write(s, ss − s);
        p ← q→link;
        q ← p;
      }
      else {
        term_write(s, ss + 1 − s);
        p ← name_dir;
        q ← Λ;
      }
      s ← p→byte_start;
    }
    if (q) term_write("...", 3);      /* complete name not yet known */
  }
```

```
43.  void sprint_section_name(dest, p)
       char *dest;
       name_pointer p;
  {
    char *ss, *s ← first_chunk(p);
    name_pointer q ← p + 1;
    while (p ≠ name_dir) {
      ss ← (p + 1)→byte_start − 1;
      if (*ss ≡ '␣' ∧ ss ≥ s) {
        p ← q→link;
        q ← p;
      }
      else {
        ss++;
        p ← name_dir;
      }
      strncpy(dest, s, ss − s), dest += ss − s;
      s ← p→byte_start;
    }
    *dest ← '\0';
  }

44.  void print_prefix_name(p)
       name_pointer p;
  {
    char *s ← first_chunk(p);
    int l ← prefix_length(p);
    term_write(s, l);
    if (s + l < (p + 1)→byte_start) term_write("...", 3);
  }
```

**45.** When we compare two section names, we'll need a function analogous to *strcmp*. But we do not assume the strings are null-terminated, and we keep an eye open for prefixes and extensions.

```
#define less 0      /* the first name is lexicographically less than the second */
#define equal 1     /* the first name is equal to the second */
#define greater 2   /* the first name is lexicographically greater than the second */
#define prefix 3    /* the first name is a proper prefix of the second */
#define extension 4 /* the first name is a proper extension of the second */
  int web_strcmp(j, j_len, k, k_len)     /* fuller comparison than strcmp */
      char *j, *k;     /* beginning of first and second strings */
      int j_len, k_len;     /* length of strings */
  {
    char *j1 ← j + j_len, *k1 ← k + k_len;
    while (k < k1 ∧ j < j1 ∧ *j ≡ *k) k++, j++;
    if (k ≡ k1)
      if (j ≡ j1) return equal;
      else return extension;
    else if (j ≡ j1) return prefix;
    else if (*j < *k) return less;
    else return greater;
  }
```

**46.** Adding a section name to the tree is straightforward if we know its parent and whether it's the *rlink* or *llink* of the parent. As a special case, when the name is the first section being added, we set the "parent" to Λ. When a section name is created, it has only one chunk, which however may be just a prefix: the full name will hopefully be unveiled later. Obviously, *prefix_length* starts out as the length of the first chunk, though it may decrease later.

The information associated with a new node must be initialized differently in `CWEAVE` and `CTANGLE`; hence the *init_node* procedure, which is defined differently in `cweave.w` and `ctangle.w`.

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  extern void init_node( );
```

**47.**
```
name_pointer add_section_name(par, c, first, last, ispref)      /* install a new node in the tree */
    name_pointer par;      /* parent of new node */
    int c;      /* right or left? */
    char *first;      /* first character of section name */
    char *last;      /* last character of section name, plus one */
    int ispref;      /* are we adding a prefix or a full name? */
{
  name_pointer p ← name_ptr;      /* new node */
  char *s ← first_chunk(p);
  int name_len ← last − first + ispref;      /* length of section name */
  if (s + name_len > byte_mem_end) overflow("byte␣memory");
  if (name_ptr + 1 ≥ name_dir_end) overflow("name");
  (++name_ptr)→byte_start ← byte_ptr ← s + name_len;
  if (ispref) {
    *(byte_ptr − 1) ← '␣';
    name_len−−;
    name_ptr→link ← name_dir;
    (++name_ptr)→byte_start ← byte_ptr;
  }
  set_prefix_length(p, name_len);
  strncpy(s, first, name_len);
  p→llink ← Λ;
  p→rlink ← Λ;
  init_node(p);
  return par ≡ Λ ? (root ← p) : c ≡ less ? (par→llink ← p) : (par→rlink ← p);
}
```

**48.**
```
void extend_section_name(p, first, last, ispref)
    name_pointer p;      /* name to be extended */
    char *first;         /* beginning of extension text */
    char *last;          /* one beyond end of extension text */
    int ispref;          /* are we adding a prefix or a full name? */
{
  char *s;
  name_pointer q ← p + 1;
  int name_len ← last − first + ispref;
  if (name_ptr ≥ name_dir_end) overflow("name");
  while (q→link ≠ name_dir) q ← q→link;
  q→link ← name_ptr;
  s ← name_ptr→byte_start;
  name_ptr→link ← name_dir;
  if (s + name_len > byte_mem_end) overflow("byte␣memory");
  (++name_ptr)→byte_start ← byte_ptr ← s + name_len;
  strncpy(s, first, name_len);
  if (ispref) *(byte_ptr − 1) ← '␣';
}
```

**49.** The *section_lookup* procedure is supposed to find a section name that matches a new name, installing the new name if its doesn't match an existing one. The new name is the string between *first* and *last*; a "match" means that the new name exactly equals or is a prefix or extension of a name in the tree.

```
name_pointer section_lookup(first, last, ispref)    /* find or install section name in tree */
    char *first, *last;    /* first and last characters of new name */
    int ispref;            /* is the new name a prefix or a full name? */
{
  int c ← 0;      /* comparison between two names; initialized so some compilers won't complain */
  name_pointer p ← root;      /* current node of the search tree */
  name_pointer q ← Λ;         /* another place to look in the tree */
  name_pointer r ← Λ;         /* where a match has been found */
  name_pointer par ← Λ;       /* parent of p, if r is Λ; otherwise parent of r */
  int name_len ← last − first + 1;
  ⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 50 ⟩;
  ⟨ If no match found, add new name to tree 51 ⟩;
  ⟨ If one match found, check for compatibility and return match 52 ⟩;
}
```

**50.** A legal new name matches an existing section name if and only if it matches the shortest prefix of that section name. Therefore we can limit our search for matches to shortest prefixes, which eliminates the need for chunk-chasing at this stage.

⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 50 ⟩ ≡

```
while (p) {      /* compare shortest prefix of p with new name */
  c ← web_strcmp(first, name_len, first_chunk(p), prefix_length(p));
  if (c ≡ less ∨ c ≡ greater) {      /* new name does not match p */
    if (r ≡ Λ)      /* no previous matches have been found */
      par ← p;
    p ← (c ≡ less ? p→llink : p→rlink);
  }
  else {      /* new name matches p */
    if (r ≠ Λ) {      /* and also r: illegal */
      printf("\n! Ambiguous prefix: matches <");
      print_prefix_name(p);
      printf(">\n and <");
      print_prefix_name(r);
      err_print(">");
      return name_dir;      /* the unsection */
    }
    r ← p;      /* remember match */
    p ← p→llink;      /* try another */
    q ← r→rlink;      /* we'll get back here if the new p doesn't match */
  }
  if (p ≡ Λ) p ← q, q ← Λ;      /* q held the other branch of r */
}
```

This code is used in section 49

**51.** ⟨ If no match found, add new name to tree 51 ⟩ ≡

```
if (r ≡ Λ)      /* no matches were found */
  return add_section_name(par, c, first, last + 1, ispref);
```

This code is used in section 49

**52.** Although error messages are given in anomalous cases, we do return the unique best match when a discrepancy is found, because users often change a title in one place while forgetting to change it elsewhere.

```
⟨ If one match found, check for compatibility and return match 52 ⟩ ≡
  switch (section_name_cmp(&first, name_len, r)) {      /* compare all of r with new name */
  case prefix:
    if (¬ispref) {
      printf("\n!␣New␣name␣is␣a␣prefix␣of␣<");
      print_section_name(r);
      err_print(">");
    }
    else if (name_len < prefix_length(r)) set_prefix_length(r, name_len);      /* fall through */
  case equal: return r;
  case extension:
    if (¬ispref ∨ first ≤ last) extend_section_name(r, first, last + 1, ispref);
    return r;
  case bad_extension: printf("\n!␣New␣name␣extends␣<");
    print_section_name(r);
    err_print(">");
    return r;
  default:      /* no match: illegal */
    printf("\n!␣Section␣name␣incompatible␣with␣<");
    print_prefix_name(r);
    printf(">,\n␣which␣abbreviates␣<");
    print_section_name(r);
    err_print(">");
    return r;
  }
```

This code is used in section 49

**53.** The return codes of *section_name_cmp*, which compares a string with the full name of a section, are those of *web_strcmp* plus *bad_extension*, used when the string is an extension of a supposedly already complete section name. This function has a side effect when the comparison string is an extension: it advances the address of the first character of the string by an amount equal to the length of the known part of the section name.

The name `@<foo...@>` should be an acceptable "abbreviation" for `@<foo@>`. If such an abbreviation comes after the complete name, there's no trouble recognizing it. If it comes before the complete name, we simply append a null chunk. This logic requires us to regard `@<foo...@>` as an "extension" of itself.

```
#define bad_extension 5
⟨ Predeclaration of procedures 33 ⟩ +≡
  int section_name_cmp( );
```

**54.**
```
int section_name_cmp(pfirst, len, r)
  char **pfirst;      /* pointer to beginning of comparison string */
  int len;      /* length of string */
  name_pointer r;      /* section name being compared */
{
  char *first ← *pfirst;      /* beginning of comparison string */
  name_pointer q ← r + 1;      /* access to subsequent chunks */
  char *ss, *s ← first_chunk(r);
  int c;      /* comparison */
  int ispref;      /* is chunk r a prefix? */
  while (1) {
    ss ← (r + 1)→byte_start − 1;
    if (*ss ≡ '␣' ∧ ss ≥ r→byte_start) ispref ← 1, q ← q→link;
    else ispref ← 0, ss++, q ← name_dir;
    switch (c ← web_strcmp(first, len, s, ss − s)) {
    case equal:
      if (q ≡ name_dir)
        if (ispref) {
          *pfirst ← first + (ss − s);
          return extension;      /* null extension */
        }
        else return equal;
      else return (q→byte_start ≡ (q + 1)→byte_start) ? equal : prefix;
    case extension:
      if (¬ispref) return bad_extension;
      first += ss − s;
      if (q ≠ name_dir) {
        len −= ss − s;
        s ← q→byte_start;
        r ← q;
        continue;
      }
      *pfirst ← first;
      return extension;
    default: return c;
    }
  }
}
```

**55.** The last component of **name_info** is different for `CTANGLE` and `CWEAVE`. In `CTANGLE`, if $p$ is a pointer to a section name, $p$→*equiv* is a pointer to its replacement text, an element of the array *text_info*. In `CWEAVE`, on the other hand, if $p$ points to an identifier, $p$→*xref* is a pointer to its list of cross-references, an element of the array *xmem*. The make-up of *text_info* and *xmem* is discussed in the `CTANGLE` and `CWEAVE` source files, respectively; here we just declare a common field *equiv_or_xref* as a pointer to a **char**.

⟨ More elements of **name_info** structure 31 ⟩ +≡
```
char *equiv_or_xref;      /* info corresponding to names */
```

**56. Reporting errors to the user.** A global variable called *history* will contain one of four values at the end of every run: *spotless* means that no unusual messages were printed; *harmless_message* means that a message of possible interest was printed but no serious errors were detected; *error_message* means that at least one error was found; *fatal_message* means that the program terminated abnormally. The value of *history* does not influence the behavior of the program; it is simply computed for the convenience of systems that might want to use such information.

```
#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
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  int history ← spotless; /* indicates how bad this run was */
```

**57.** The command '*err_print*(`"!␣Error␣message"`)' will report a syntax error to the user, by printing the error message at the beginning of a new line and then giving an indication of where the error was spotted in the source file. Note that no period follows the error message, since the error routine will automatically supply a period. A newline is automatically supplied if the string begins with `"|"`.

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  void err_print( );
```

**58.**

```
void err_print(s) /* prints '.' and location of error message */
    char *s;
{
  char *k, *l; /* pointers into buffer */
  printf(*s ≡ '!' ? "\n%s" : "%s", s);
  if (web_file_open) ⟨ Print error location based on input buffer 59 ⟩;
  update_terminal;
  mark_error;
}
```

**59.** The error locations can be indicated by using the global variables *loc*, *cur_line*, *cur_file_name* and *changing*, which tell respectively the first unlooked-at position in *buffer*, the current line number, the current file, and whether the current line is from *change_file* or *cur_file*. This routine should be modified on systems whose standard text editor has special line-numbering conventions.

```
⟨ Print error location based on input buffer 59 ⟩ ≡
  {
    if (changing ∧ include_depth ≡ change_depth)
      printf(". (l. %d of change file)\n", change_line);
    else if (include_depth ≡ 0) printf(". (l. %d)\n", cur_line);
    else printf(". (l. %d of include file %s)\n", cur_line, cur_file_name);
    l ← (loc ≥ limit ? limit : loc);
    if (l > buffer) {
      for (k ← buffer; k < l; k++)
        if (∗k ≡ '\t') putchar(' ');
        else putchar(∗k);      /∗ print the characters already read ∗/
      putchar('\n');
      for (k ← buffer; k < l; k++) putchar(' ');      /∗ space out the next line ∗/
    }
    for (k ← l; k < limit; k++) putchar(∗k);      /∗ print the part not yet read ∗/
    if (∗limit ≡ '|') putchar('|');      /∗ end of C text in section names ∗/
    putchar(' ');      /∗ to separate the message from future asterisks ∗/
  }
```

This code is used in section 58

**60.** When no recovery from some error has been provided, we have to wrap up and quit as graciously as possible. This is done by calling the function *wrap_up* at the end of the code.

CTANGLE and CWEAVE have their own notions about how to print the job statistics.

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  int wrap_up( );
  extern void print_stats( );
```

**61.** Some implementations may wish to pass the *history* value to the operating system so that it can be used to govern whether or not other programs are started. Here, for instance, we pass the operating system a status of 0 if and only if only harmless messages were printed.

```
  int wrap_up( )
  {
    putchar('\n');
    if (show_stats) print_stats( );      /∗ print statistics about memory usage ∗/
    ⟨ Print the job history 62 ⟩;
    if (history > harmless_message) return (1);
    else return (0);
  }
```

**62.**

```
⟨ Print the job history 62 ⟩ ≡
  switch (history) {
  case spotless:
    if (show_happiness) printf("(No errors were found.)\n");
    break;
  case harmless_message: printf("(Did you see the warning message above?)\n");
    break;
  case error_message: printf("(Pardon me, but I think I spotted something wrong.)\n");
    break;
  case fatal_message: printf("(That was a fatal error, my friend.)\n");
  }      /∗ there are no other cases ∗/
```

This code is used in section 61

**63.** When there is no way to recover from an error, the *fatal* subroutine is invoked. This happens most often when *overflow* occurs.

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  void fatal( ), overflow( );
```

**64.** The two parameters to *fatal* are strings that are essentially concatenated to print the final error message.

```
void fatal(s, t)
    char *s, *t;
{
  if (*s) printf(s);
  err_print(t);
  history ← fatal_message;
  exit(wrap_up( ));
}
```

**65.** An overflow stop occurs if CWEB's tables aren't large enough.

```
void overflow(t)
    char *t;
{
  printf("\n! Sorry, %s capacity exceeded", t);
  fatal("", "");
}
```

**66.** Sometimes the program's behavior is far different from what it should be, and CWEB prints an error message that is really for the CWEB maintenance person, not the user. In such cases the program says *confusion*("indication␣of␣where␣we␣are").

```
#define confusion(s) fatal("! This can't happen: ", s)
```

**67. Command line arguments.** The user calls CWEAVE and CTANGLE with arguments on the command line. These are either file names or flags to be turned off (beginning with "-") or flags to be turned on (beginning with "+". The following globals are for communicating the user's desires to the rest of the program. The various file name variables contain strings with the names of those files. Most of the 128 flags are undefined but available for future extensions.

```
#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? */
```

⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡

```
int argc;   /* copy of ac parameter to main */
char **argv;   /* copy of av parameter to main */
char C_file_name[max_file_name_length];   /* name of C_file */
char tex_file_name[max_file_name_length];   /* name of tex_file */
char idx_file_name[max_file_name_length];   /* name of idx_file */
char scn_file_name[max_file_name_length];   /* name of scn_file */
boolean flags[128];   /* an option for each 7-bit code */
```

**68.** The *flags* will be initially zero. Some of them are set to 1 before scanning the arguments; if additional flags are 1 by default they should be set before calling *common_init*.

⟨ Set the default options common to CTANGLE and CWEAVE 68 ⟩ ≡

```
show_banner ← show_happiness ← show_progress ← 1;
```

This code is used in section 4

**69.** We now must look at the command line arguments and set the file names accordingly. At least one file name must be present: the CWEB file. It may have an extension, or it may omit the extension to get ".w" or ".web" added. The TEX output file name is formed by replacing the CWEB file name extension by ".tex", and the C file name by replacing the extension by ".c", after removing the directory name (if any).

If there is a second file name present among the arguments, it is the change file, again either with an extension or without one to get ".ch". An omitted change file argument means that "/dev/null" should be used, when no changes are desired.

If there's a third file name, it will be the output file.

⟨ Predeclaration of procedures 33 ⟩ +≡

```
void scan_args();
```

**70.** **void** *scan_args*( )
```
{
  char *dot_pos;      /* position of '.' in the argument */
  char *name_pos;     /* file name beginning, sans directory */
  register char *s;   /* register for scanning strings */
  boolean found_web ← 0, found_change ← 0, found_out ← 0;
    /* have these names have been seen? */
  boolean flag_change;
  while (--argc > 0) {
    if ((**(++argv) ≡ '-' ∨ **argv ≡ '+') ∧ *(*argv + 1)) ⟨ Handle flag argument 74 ⟩
    else {
      s ← name_pos ← *argv; dot_pos ← Λ;
      while (*s) {
        if (*s ≡ '.') dot_pos ← s++;
        else if (*s ≡ '/') dot_pos ← Λ, name_pos ← ++s;
        else s++;
      }
      if (¬found_web) ⟨ Make web_file_name, tex_file_name and C_file_name 71 ⟩
      else if (¬found_change) ⟨ Make change_file_name from fname 72 ⟩
      else if (¬found_out) ⟨ Override tex_file_name and C_file_name 73 ⟩
      else ⟨ Print usage error message and quit 75 ⟩;
    }
  }
  if (¬found_web) ⟨ Print usage error message and quit 75 ⟩;
  if (found_change ≤ 0) strcpy(change_file_name, "/dev/null");
}
```

**71.** We use all of **argv* for the *web_file_name* if there is a `'.'` in it, otherwise we add `".w"`. If this file can't be opened, we prepare an *alt_web_file_name* by adding `"web"` after the dot. The other file names come from adding other things after the dot. We must check that there is enough room in *web_file_name* and the other arrays for the argument.

⟨ Make *web_file_name*, *tex_file_name* and *C_file_name* 71 ⟩ ≡
```
{
  if (s − *argv > max_file_name_length − 5) ⟨ Complain about argument length 76 ⟩;
  if (dot_pos ≡ Λ) sprintf(web_file_name, "%s.w", *argv);
  else {
    strcpy(web_file_name, *argv);
    *dot_pos ← 0;     /* string now ends where the dot was */
  }
  sprintf(alt_web_file_name, "%s.web", *argv);
  sprintf(tex_file_name, "%s.tex", name_pos);     /* strip off directory name */
  sprintf(idx_file_name, "%s.idx", name_pos);
  sprintf(scn_file_name, "%s.scn", name_pos);
  sprintf(C_file_name, "%s.c", name_pos);
  found_web ← 1;
}
```
This code is used in section 70

**72.** ⟨ Make *change_file_name* from *fname* 72 ⟩ ≡

```
{
  if (strcmp(*argv, "-") ≡ 0) found_change ← −1;
  else {
    if (s − *argv > max_file_name_length − 4) ⟨ Complain about argument length 76 ⟩;
    if (dot_pos ≡ Λ) sprintf(change_file_name, "%s.ch", *argv);
    else strcpy(change_file_name, *argv);
    found_change ← 1;
  }
}
```

This code is used in section 70

**73.** ⟨ Override *tex_file_name* and *C_file_name* 73 ⟩ ≡

```
{
  if (s − *argv > max_file_name_length − 5) ⟨ Complain about argument length 76 ⟩;
  if (dot_pos ≡ Λ) {
    sprintf(tex_file_name, "%s.tex", *argv);
    sprintf(idx_file_name, "%s.idx", *argv);
    sprintf(scn_file_name, "%s.scn", *argv);
    sprintf(C_file_name, "%s.c", *argv);
  }
  else {
    strcpy(tex_file_name, *argv);
    if (flags['x']) {      /* indexes will be generated */
      if (program ≡ cweave ∧ strcmp(*argv + strlen(*argv) − 4, ".tex") ≠ 0)
        fatal("! Output file name should end with .tex\n", *argv);
      strcpy(idx_file_name, *argv);
      strcpy(idx_file_name + strlen(*argv) − 4, ".idx");
      strcpy(scn_file_name, *argv);
      strcpy(scn_file_name + strlen(*argv) − 4, ".scn");
    }
    strcpy(C_file_name, *argv);
  }
  found_out ← 1;
}
```

This code is used in section 70

**74.** ⟨ Handle flag argument 74 ⟩ ≡

```
{
  if (**argv ≡ '-') flag_change ← 0;
  else flag_change ← 1;
  for (dot_pos ← *argv + 1; *dot_pos > '\0'; dot_pos++) flags[*dot_pos] ← flag_change;
}
```

This code is used in section 70

**75.** ⟨ Print usage error message and quit 75 ⟩ ≡

```
{
  if (program ≡ ctangle)
    fatal("! Usage: ctangle [options] webfile[.w] [{changefile[.ch]|-}\
        [outfile[.c]]]\n", "");
  else fatal("! Usage: cweave [options] webfile[.w] [{changefile[.ch]|-}\
        [outfile[.tex]]]\n", "");
}
```

This code is used in section 70

**76.** ⟨ Complain about argument length 76 ⟩ ≡
*fatal*("! Filename too long\n", ∗*argv*);
This code is used in sections 71, 72, and 73

**77. Output.** Here is the code that opens the output file:

```
⟨ Definitions that should agree with CTANGLE and CWEAVE 2 ⟩ +≡
  FILE *C_file;      /* where output of CTANGLE goes */
  FILE *tex_file;    /* where output of CWEAVE goes */
  FILE *idx_file;    /* where index from CWEAVE goes */
  FILE *scn_file;    /* where list of sections from CWEAVE goes */
  FILE *active_file;    /* currently active file for CWEAVE output */
```

**78.**

```
⟨ Scan arguments and open output files 78 ⟩ ≡
  scan_args();
  if (program ≡ ctangle) {
    if ((C_file ← fopen(C_file_name, "w")) ≡ Λ) fatal("! Cannot open output file ", C_file_name);
  }
  else {
    if ((tex_file ← fopen(tex_file_name, "w")) ≡ Λ)
      fatal("! Cannot open output file ", tex_file_name);
  }
```

This code is used in section 4

**79.** The *update_terminal* procedure is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent.

```
#define update_terminal fflush(stdout)    /* empty the terminal output buffer */
```

**80.** Terminal output uses *putchar* and *putc* when we have to translate from CWEB's code into the external character code, and *printf* when we just want to print strings. Several macros make other kinds of output convenient.

```
#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? */
```

**81.** We predeclare several standard system functions here instead of including their system header files, because the names of the header files are not as standard as the names of the functions. (For example, some C environments have <string.h> where others have <strings.h>.)

```
⟨ Predeclaration of procedures 33 ⟩ +≡
  extern int strlen();    /* length of string */
  extern int strcmp();    /* compare strings lexicographically */
  extern char *strcpy();    /* copy one string to another */
  extern int strncmp();    /* compare up to n string characters */
  extern char *strncpy();    /* copy up to n string characters */
```

**82. Index.**

## Index of common

## Contents of ctangle

## Sections of ctangle

⟨ Case of a section number 54 ⟩ Used in section 49
⟨ Case of an identifier 53 ⟩ Used in section 49
⟨ Cases like != 50 ⟩ Used in section 49
⟨ Common code for CWEAVE and CTANGLE 5, 7, 8, 9, 10, 11, 12, 13, 14, 15 ⟩ Used in section 1
⟨ Compress two-symbol operator 64 ⟩ Used in section 63
⟨ Copy a string or verbatim construction or numerical constant 80 ⟩ Used in section 78
⟨ Copy an ASCII constant 81 ⟩ Used in section 78
⟨ Expand section $a - °24000$, **goto** *restart* 34 ⟩ Used in section 33
⟨ Get a constant 66 ⟩ Used in section 63
⟨ Get a string 67 ⟩ Used in section 63
⟨ Get an identifier 65 ⟩ Used in section 63
⟨ Get control code and possible section name 68 ⟩ Cited in section 84 Used in section 63
⟨ Global variables 17, 23, 28, 32, 36, 38, 45, 51, 56, 59, 61, 75, 82 ⟩ Used in section 1
⟨ If end of name or erroneous nesting, **break** 73 ⟩ Used in section 72
⟨ If it's not there, add *cur_section_name* to the output file stack, or complain we're out of room 40 ⟩ Used in section 70
⟨ If section is not being defined, **continue** 86 ⟩ Used in section 83
⟨ In cases that $a$ is a non-**char** token (*identifier*, *section_name*, etc.), either process it and change $a$ to a byte that should be stored, or **continue** if $a$ should be ignored, or **goto** *done* if $a$ signals the end of this replacement text 78 ⟩ Used in section 76
⟨ Include files 6, 62 ⟩ Used in section 1
⟨ Initialize the output stacks 29 ⟩ Used in section 42
⟨ Insert the line number into *tok_mem* 77 ⟩ Used in sections 63, 76, and 78
⟨ Insert the section number into *tok_mem* 88 ⟩ Used in section 87
⟨ Output macro definitions if appropriate 44 ⟩ Used in section 42
⟨ Predeclaration of procedures 2, 41, 46, 48, 90, 92 ⟩ Used in section 1
⟨ Put section name into *section_text* 72 ⟩ Used in section 70
⟨ Read in transliteration of a character 94 ⟩ Used in section 93
⟨ Scan a definition 85 ⟩ Used in section 83
⟨ Scan a verbatim string 74 ⟩ Used in section 68
⟨ Scan an ASCII constant 69 ⟩ Used in section 68
⟨ Scan the C part of the current section 87 ⟩ Used in section 83
⟨ Scan the section name and make *cur_section_name* point to it 70 ⟩ Used in section 68
⟨ Set initial values 18, 20, 24, 39, 52, 57, 71 ⟩ Used in section 3
⟨ Skip ahead until *next_control* corresponds to @d, @<, @␣ or the like 84 ⟩ Used in section 83
⟨ Typedef declarations 16, 27 ⟩ Used in section 1
⟨ Update the data structure so that the replacement text is accessible 89 ⟩ Used in section 87
⟨ Was an '@' missed here? 79 ⟩ Used in section 78
⟨ Write all the named output files 43 ⟩ Used in section 42

**1. Introduction.** This is the CTANGLE program by Silvio Levy and Donald E. Knuth, based on TANGLE by Knuth. We are thankful to Nelson Beebe, Hans-Hermann Bode (to whom the C++ adaptation is due), Klaus Guntermann, Norman Ramsey, Tomas Rokicki, Joachim Schnitter, Joachim Schrod, Lee Wittenberg, and others who have contributed improvements.

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

```
#define banner "This␣is␣CTANGLE␣(Version␣3.1)\n"
```

⟨ Include files 6 ⟩
⟨ Preprocessor definitions ⟩
⟨ Common code for CWEAVE and CTANGLE 5 ⟩
⟨ Typedef declarations 16 ⟩
⟨ Global variables 17 ⟩
⟨ Predeclaration of procedures 2 ⟩

**2.** We predeclare several standard system functions here instead of including their system header files, because the names of the header files are not as standard as the names of the functions. (For example, some C environments have <string.h> where others have <strings.h>.)

⟨ Predeclaration of procedures 2 ⟩ ≡

```
extern int strlen( );      /* length of string */
extern int strcmp( );      /* compare strings lexicographically */
extern char *strcpy( );    /* copy one string to another */
extern int strncmp( );     /* compare up to n string characters */
extern char *strncpy( );   /* copy up to n string characters */
```

See also sections 41, 46, 48, 90, and 92

This code is used in section 1

**3.** CTANGLE has a fairly straightforward outline. It operates in two phases: first it reads the source file, saving the C code in compressed form; then outputs the code, after shuffling it around.

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

```
int main(ac, av)
    int ac;
    char **av;
{
  argc ← ac;
  argv ← av;
  program ← ctangle;
  ⟨ Set initial values 18 ⟩;
  common_init( );
  if (show_banner) printf(banner);      /* print a "banner line" */
  phase_one( );      /* read all the user's text and compress it into tok_mem */
  phase_two( );      /* output the contents of the compressed tables */
  return wrap_up( );      /* and exit gracefully */
}
```

**4.** The following parameters were sufficient in the original TANGLE to handle TEX, so they should be sufficient for most applications of CTANGLE. If you change *max_bytes*, *max_names* or *hash_size* you should also change them in the file "common.w".

```
#define max_bytes 90000
        /* the number of bytes in identifiers, index entries, and section names; used in "common.w" */
#define max_toks 270000    /* number of bytes in compressed C code */
#define max_names 4000    /* number of identifiers, strings, section names; must be less than 10240;
          used in "common.w" */
#define max_texts 2500    /* number of replacement texts, must be less than 10240 */
#define hash_size 353    /* should be prime; used in "common.w" */
#define longest_name 1000    /* section names shouldn't be longer than this */
#define stack_size 50    /* number of simultaneous levels of macro expansion */
#define buf_size 100    /* for CWEAVE and CTANGLE */
```

**5.** 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 needs to be updated when "common.w" changes.

First comes general stuff:

```
#define ctangle 0
#define cweave 1
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ ≡
  typedef short boolean;
  typedef char unsigned eight_bits;
  extern boolean program;    /* CWEAVE or CTANGLE? */
  extern int phase;    /* which phase are we in? */
```

See also sections 7, 8, 9, 10, 11, 12, 13, 14, and 15

This code is used in section 1

**6.** ⟨ Include files 6 ⟩ ≡

```
#include <stdio.h>
```

See also section 62

This code is used in section 1

**7.** 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 not_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 ϖ */
#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 ↰ */
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  char section_text[longest_name + 1];    /* name being sought for */
  char *section_text_end ← section_text + longest_name;    /* end of section_text */
  char *id_first;    /* where the current identifier begins in the buffer */
  char *id_loc;    /* just after the current identifier in the buffer */
```

**8.** Code related to input routines:

```
#define xisalpha(c) (isalpha(c) ∧ ((eight_bits) c < °200))
#define xisdigit(c) (isdigit(c) ∧ ((eight_bits) c < °200))
#define xisspace(c) (isspace(c) ∧ ((eight_bits) c < °200))
#define xislower(c) (islower(c) ∧ ((eight_bits) c < °200))
#define xisupper(c) (isupper(c) ∧ ((eight_bits) c < °200))
#define xisxdigit(c) (isxdigit(c) ∧ ((eight_bits) c < °200))
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  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 */
```

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

```
#define length(c) (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 */
#define chunk_marker 0
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  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;
    char *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 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 byte_start */
  extern char *byte_ptr;      /* first unused position in byte_mem */
  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 */
  extern name_pointer id_lookup( );      /* looks up a string in the identifier table */
  extern name_pointer section_lookup( );      /* finds section name */
  extern void print_section_name( ), sprint_section_name( );
```

**10.** 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 5 ⟩ +≡
  extern history; /* indicates how bad this run was */
  extern err_print( ); /* print error message and context */
  extern wrap_up( ); /* indicate history and exit */
  extern void fatal( ); /* issue error message and die */
  extern void overflow( ); /* succumb because a table has overflowed */
```

**11.** Code related to file handling:

```
  format line x /* make line an unreserved word */
#define max_file_name_length 60
#define cur_file file[include_depth] /* current file */
#define cur_file_name file_name[include_depth] /* current file name */
#define web_file_name file_name[0] /* main source file name */
#define cur_line line[include_depth] /* number of current line in current file */
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  extern include_depth; /* current level of nesting */
  extern FILE *file[]; /* stack of non-change files */
  extern FILE *change_file; /* change file */
  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 char file_name[][max_file_name_length]; /* stack of non-change file names */
  extern char change_file_name[]; /* name of change file */
  extern line[]; /* number of current line in the stacked files */
  extern change_line; /* number of current line in change file */
  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 */
  extern reset_input( ); /* initialize to read the web file and change file */
  extern get_line( ); /* inputs the next line */
  extern check_complete( ); /* checks that all changes were picked up */
```

**12.** Code related to section numbers:

```
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  typedef unsigned short sixteen_bits;
  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 */
```

**13.** 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? */
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  extern int argc;  /* copy of ac parameter to main */
  extern char **argv;  /* copy of av parameter to main */
  extern boolean flags[];  /* an option for each 7-bit code */
```

**14.** Code relating 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)
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  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 */
```

**15.** The procedure that gets everything rolling:

```
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  extern void common_init();
```

**16. Data structures exclusive to CTANGLE.** We've already seen that the *byte_mem* array holds the names of identifiers, strings, and sections; the *tok_mem* array holds the replacement texts for sections. Allocation is sequential, since things are deleted only during Phase II, and only in a last-in-first-out manner.

A **text** variable is a structure containing a pointer into *tok_mem*, which tells where the corresponding text starts, and an integer *text_link*, which, as we shall see later, is used to connect pieces of text that have the same name. All the **text**s are stored in the array *text_info*, and we use a *text_pointer* variable to refer to them.

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

If your machine does not support **unsigned char** you should change the definition of **eight_bits** to **unsigned short**.

```
⟨ Typedef declarations 16 ⟩ ≡
  typedef struct {
    eight_bits *tok_start;      /* pointer into tok_mem */
    sixteen_bits text_link;     /* relates replacement texts */
  } text;
  typedef text *text_pointer;
```

See also section 27

This code is used in section 1

**17.**

```
⟨ Global variables 17 ⟩ ≡
  text text_info[max_texts];
  text_pointer text_info_end ← text_info + max_texts − 1;
  text_pointer text_ptr;      /* first unused position in text_info */
  eight_bits tok_mem[max_toks];
  eight_bits *tok_mem_end ← tok_mem + max_toks − 1;
  eight_bits *tok_ptr;        /* first unused position in tok_mem */
```

See also sections 23, 28, 32, 36, 38, 45, 51, 56, 59, 61, 75, and 82

This code is used in section 1

**18.**

```
⟨ Set initial values 18 ⟩ ≡
  text_info→tok_start ← tok_ptr ← tok_mem;
  text_ptr ← text_info + 1;
  text_ptr→tok_start ← tok_mem;      /* this makes replacement text 0 of length zero */
```

See also sections 20, 24, 39, 52, 57, and 71

This code is used in section 3

**19.** If $p$ is a pointer to a section name, $p$→*equiv* is a pointer to its replacement text, an element of the array *text_info*.

```
#define equiv equiv_or_xref      /* info corresponding to names */
```

**20.**

```
⟨ Set initial values 18 ⟩ +≡
  name_dir→equiv ← (char *) text_info;      /* the undefined section has no replacement text */
```

**21.** Here's the procedure that decides whether a name of length $l$ starting at position *first* equals the identifier pointed to by $p$:

```
int names_match(p, first, l)
    name_pointer p;    /* points to the proposed match */
    char *first;    /* position of first character of string */
    int l;    /* length of identifier */
{
  if (length(p) ≠ l) return 0;
  return ¬strncmp(first, p→byte_start, l);
}
```

**22.** The common lookup routine refers to separate routines *init_node* and *init_p* when the data structure grows. Actually *init_p* is called only by CWEAVE, but we need to declare a dummy version so that the loader won't complain of its absence.

```
void init_node(node)
    name_pointer node;
{
  node→equiv ← (char *) text_info;
}
void init_p( )
{ }
```

**23. Tokens.** Replacement texts, which represent C code in a compressed format, appear in *tok_mem* as mentioned above. The codes in these texts are called 'tokens'; some tokens occupy two consecutive eight-bit byte positions, and the others take just one byte.

If $p$ points to a replacement text, $p\rightarrow tok\_start$ is the *tok_mem* position of the first eight-bit code of that text. If $p\rightarrow text\_link \equiv 0$, this is the replacement text for a macro, otherwise it is the replacement text for a section. In the latter case $p\rightarrow text\_link$ is either equal to *section_flag*, which means that there is no further text for this section, or $p\rightarrow text\_link$ points to a continuation of this replacement text; such links are created when several sections have C texts with the same name, and they also tie together all the C texts of unnamed sections. The replacement text pointer for the first unnamed section appears in $text\_info\rightarrow text\_link$, and the most recent such pointer is *last_unnamed*.

```
#define section_flag max_texts /* final text_link in section replacement texts */
⟨ Global variables 17 ⟩ +≡
  text_pointer last_unnamed; /* most recent replacement text of unnamed section */
```

**24.**
```
⟨ Set initial values 18 ⟩ +≡
  last_unnamed ← text_info;
  text_info→text_link ← 0;
```

**25.** If the first byte of a token is less than $°200$, the token occupies a single byte. Otherwise we make a sixteen-bit token by combining two consecutive bytes $a$ and $b$. If $°200 \le a < °250$, then $(a - °200) \times 2^8 + b$ points to an identifier; if $°250 \le a < °320$, then $(a - °250) \times 2^8 + b$ points to a section name (or, if it has the special value *output_defs_flag*, to the area where the preprocessor definitions are stored); and if $°320 \le a < °400$, then $(a - °320) \times 2^8 + b$ is the number of the section in which the current replacement text appears.

Codes less than $°200$ are 7-bit **char** codes that represent themselves. Some of the 7-bit codes will not be present, however, so we can use them for special purposes. The following symbolic names are used:

*join* denotes the concatenation of adjacent items with no space or line breaks allowed between them (the `@&` operation of `CWEB`).

*string* denotes the beginning or end of a string, verbatim construction or numerical constant.

```
#define string °2 /* takes the place of extended ASCII α */
#define join °177 /* takes the place of ASCII delete */
#define output_defs_flag (2 * °24000 - 1)
```

**26.** The following procedure is used to enter a two-byte value into *tok_mem* when a replacement text is being generated.

```
void store_two_bytes(x)
    sixteen_bits x;
{
  if (tok_ptr + 2 > tok_mem_end) overflow("token");
  *tok_ptr++ ← x ≫ 8; /* store high byte */
  *tok_ptr++ ← x & °377; /* store low byte */
}
```

**27. Stacks for output.** The output process uses a stack to keep track of what is going on at different "levels" as the sections are being written out. Entries on this stack have five parts:

*end_field* is the *tok_mem* location where the replacement text of a particular level will end;
*byte_field* is the *tok_mem* location from which the next token on a particular level will be read;
*name_field* points to the name corresponding to a particular level;
*repl_field* points to the replacement text currently being read at a particular level;
*section_field* is the section number, or zero if this is a macro.

The current values of these five quantities are referred to quite frequently, so they are stored in a separate place instead of in the *stack* array. We call the current values *cur_end*, *cur_byte*, *cur_name*, *cur_repl*, and *cur_section*.

The global variable *stack_ptr* tells how many levels of output are currently in progress. The end of all output occurs when the stack is empty, i.e., when *stack_ptr* ≡ *stack*.

```
⟨ Typedef declarations 16 ⟩ +≡
  typedef struct {
    eight_bits *end_field;      /* ending location of replacement text */
    eight_bits *byte_field;     /* present location within replacement text */
    name_pointer name_field;    /* byte_start index for text being output */
    text_pointer repl_field;    /* tok_start index for text being output */
    sixteen_bits section_field; /* section number or zero if not a section */
  } output_state;
  typedef output_state *stack_pointer;
```

**28.**
```
#define cur_end cur_state.end_field      /* current ending location in tok_mem */
#define cur_byte cur_state.byte_field    /* location of next output byte in tok_mem */
#define cur_name cur_state.name_field    /* pointer to current name being expanded */
#define cur_repl cur_state.repl_field    /* pointer to current replacement text */
#define cur_section cur_state.section_field  /* current section number being expanded */
⟨ Global variables 17 ⟩ +≡
  output_state cur_state;      /* cur_end, cur_byte, cur_name, cur_repl and cur_section */
  output_state stack[stack_size + 1];  /* info for non-current levels */
  stack_pointer stack_ptr;     /* first unused location in the output state stack */
  stack_pointer stack_end ← stack + stack_size;  /* end of stack */
```

**29.** To get the output process started, we will perform the following initialization steps. We may assume that *text_info→text_link* is nonzero, since it points to the C text in the first unnamed section that generates code; if there are no such sections, there is nothing to output, and an error message will have been generated before we do any of the initialization.

```
⟨ Initialize the output stacks 29 ⟩ ≡
  stack_ptr ← stack + 1;
  cur_name ← name_dir;
  cur_repl ← text_info→text_link + text_info;
  cur_byte ← cur_repl→tok_start;
  cur_end ← (cur_repl + 1)→tok_start;
  cur_section ← 0;
```

This code is used in section 42

**30.** When the replacement text for name $p$ is to be inserted into the output, the following subroutine is called to save the old level of output and get the new one going.

We assume that the C compiler can copy structures.

```
void push_level(p)      /* suspends the current level */
    name_pointer p;
{
  if (stack_ptr ≡ stack_end) overflow("stack");
  *stack_ptr ← cur_state;
  stack_ptr++;
  if (p ≠ Λ) {      /* p ≡ Λ means we are in output_defs */
    cur_name ← p;
    cur_repl ← (text_pointer) p→equiv;
    cur_byte ← cur_repl→tok_start;
    cur_end ← (cur_repl + 1)→tok_start;
    cur_section ← 0;
  }
}
```

**31.** When we come to the end of a replacement text, the *pop_level* subroutine does the right thing: It either moves to the continuation of this replacement text or returns the state to the most recently stacked level.

```
void pop_level(flag)      /* do this when cur_byte reaches cur_end */
    int flag;      /* flag ≡ 0 means we are in output_defs */
{
  if (flag ∧ cur_repl→text_link < section_flag) {      /* link to a continuation */
    cur_repl ← cur_repl→text_link + text_info;      /* stay on the same level */
    cur_byte ← cur_repl→tok_start;
    cur_end ← (cur_repl + 1)→tok_start;
    return;
  }
  stack_ptr--;      /* go down to the previous level */
  if (stack_ptr > stack) cur_state ← *stack_ptr;
}
```

**32.** The heart of the output procedure is the function *get_output*, which produces the next token of output and sends it on to the lower-level function *out_char*. The main purpose of *get_output* is to handle the necessary stacking and unstacking. It sends the value *section_number* if the next output begins or ends the replacement text of some section, in which case *cur_val* is that section's number (if beginning) or the negative of that value (if ending). (A section number of 0 indicates not the beginning or ending of a section, but a **#line** command.) And it sends the value *identifier* if the next output is an identifier, in which case *cur_val* points to that identifier name.

```
#define section_number °201      /* code returned by get_output for section numbers */
#define identifier °202      /* code returned by get_output for identifiers */
⟨ Global variables 17 ⟩ +≡
  int cur_val;      /* additional information corresponding to output token */
```

**33.** If *get_output* finds that no more output remains, it returns with *stack_ptr* ≡ *stack*.

```
void get_output( )      /* sends next token to out_char */
{
  sixteen_bits a;      /* value of current byte */
restart:
  if (stack_ptr ≡ stack) return;
  if (cur_byte ≡ cur_end) {
    cur_val ← −((int) cur_section);      /* cast needed because of sign extension */
    pop_level(1);
    if (cur_val ≡ 0) goto restart;
    out_char(section_number);
    return;
  }
  a ← ∗cur_byte++;
  if (out_state ≡ verbatim ∧ a ≠ string ∧ a ≠ constant ∧ a ≠ '\n') C_putc(a);
      /* a high-bit character can occur in a string */
  else if (a < °200) out_char(a);      /* one-byte token */
  else {
    a ← (a − °200) ∗ °400 + ∗cur_byte++;
    switch (a/°24000) {      /* °24000 ≡ (°250 − °200) ∗ °400 */
    case 0: cur_val ← a;
      out_char(identifier);
      break;
    case 1:
      if (a ≡ output_defs_flag) output_defs( );
      else ⟨ Expand section a − °24000, goto restart 34 ⟩;
      break;
    default: cur_val ← a − °50000;
      if (cur_val > 0) cur_section ← cur_val;
      out_char(section_number);
    }
  }
}
```

**34.** The user may have forgotten to give any C text for a section name, or the C text may have been associated with a different name by mistake.

```
⟨ Expand section a − °24000, goto restart 34 ⟩ ≡
{
  a −= °24000;
  if ((a + name_dir)→equiv ≠ (char ∗) text_info) push_level(a + name_dir);
  else if (a ≠ 0) {
    printf("\n! Not present: <");
    print_section_name(a + name_dir);
    err_print(">");
  }
  goto restart;
}
```

This code is used in section 33

**35. Producing the output.** The *get_output* routine above handles most of the complexity of output generation, but there are two further considerations that have a nontrivial effect on CTANGLE's algorithms.

**36.** First, we want to make sure that the output has spaces and line breaks in the right places (e.g., not in the middle of a string or a constant or an identifier, not at a '@&' position where quantities are being joined together, and certainly after an = because the C compiler thinks =- is ambiguous).

The output process can be in one of following states:

*num_or_id* means that the last item in the buffer is a number or identifier, hence a blank space or line break must be inserted if the next item is also a number or identifier.

*unbreakable* means that the last item in the buffer was followed by the @& operation that inhibits spaces between it and the next item.

*verbatim* means we're copying only character tokens, and that they are to be output exactly as stored. This is the case during strings, verbatim constructions and numerical constants.

*normal* means none of the above.

Furthermore, if the variable *protect* is positive, newlines are preceded by a '\'.

```
#define normal 0 /* non-unusual state */
#define num_or_id 1 /* state associated with numbers and identifiers */
#define unbreakable 3 /* state associated with @& */
#define verbatim 4 /* state in the middle of a string */
⟨ Global variables 17 ⟩ +≡
  eight_bits out_state; /* current status of partial output */
  boolean protect; /* should newline characters be quoted? */
```

**37.** Here is a routine that is invoked when we want to output the current line. During the output process, *cur_line* equals the number of the next line to be output.

```
void flush_buffer( ) /* writes one line to output file */
{
  C_putc('\n');
  if (cur_line % 100 ≡ 0 ∧ show_progress) {
    printf(".");
    if (cur_line % 500 ≡ 0) printf("%d", cur_line);
    update_terminal; /* progress report */
  }
  cur_line++;
}
```

**38.** Second, we have modified the original TANGLE so that it will write output on multiple files. If a section name is introduced in at least one place by @( instead of @<, we treat it as the name of a file. All these special sections are saved on a stack, *output_files*. We write them out after we've done the unnamed section.

```
#define max_files 256
⟨ Global variables 17 ⟩ +≡
  name_pointer output_files[max_files];
  name_pointer *cur_out_file, *end_output_files, *an_output_file;
  char cur_section_name_char; /* is it '<' or '(' */
  char output_file_name[longest_name]; /* name of the file */
```

**39.** We make *end_output_files* point just beyond the end of *output_files*. The stack pointer *cur_out_file* starts out there. Every time we see a new file, we decrement *cur_out_file* and then write it in.

```
⟨ Set initial values 18 ⟩ +≡
  cur_out_file ← end_output_files ← output_files + max_files;
```

**40.** ⟨ If it's not there, add *cur_section_name* to the output file stack, or complain we're out of room 40 ⟩ ≡

```
{
  if (cur_out_file > output_files) {
    for (an_output_file ← cur_out_file; an_output_file < end_output_files; an_output_file++)
      if (*an_output_file ≡ cur_section_name) break;
    if (an_output_file ≡ end_output_files) *--cur_out_file ← cur_section_name;
  }
  else {
    overflow("output␣files");
  }
}
```

This code is used in section 70

**41. The big output switch.** Here then is the routine that does the output.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void phase_two( );
```

**42.**

```
void phase_two( )
{
  web_file_open ← 0;
  cur_line ← 1;
  ⟨ Initialize the output stacks 29 ⟩;
  ⟨ Output macro definitions if appropriate 44 ⟩;
  if (text_info→text_link ≡ 0 ∧ cur_out_file ≡ end_output_files) {
    printf("\n! No program text was specified.");
    mark_harmless;
  }
  else {
    if (cur_out_file ≡ end_output_files) {
      if (show_progress) printf("\nWriting the output file (%s):", C_file_name);
    }
    else {
      if (show_progress) {
        printf("\nWriting the output files:");
        printf(" (%s)", C_file_name);
        update_terminal;
      }
      if (text_info→text_link ≡ 0) goto writeloop;
    }
    while (stack_ptr > stack) get_output( );
    flush_buffer( );
  writeloop: ⟨ Write all the named output files 43 ⟩;
    if (show_happiness) printf("\nDone.");
  }
}
```

**43.** To write the named output files, we proceed as for the unnamed section. The only subtlety is that we have to open each one.

```
⟨ Write all the named output files 43 ⟩ ≡
  for (an_output_file ← end_output_files; an_output_file > cur_out_file; ) {
    an_output_file −−;
    sprint_section_name(output_file_name, ∗an_output_file);
    fclose(C_file);
    C_file ← fopen(output_file_name, "w");
    if (C_file ≡ 0) fatal("! Cannot open output file:", output_file_name);
    printf("\n(%s)", output_file_name);
    update_terminal;
    cur_line ← 1;
    stack_ptr ← stack + 1;
    cur_name ← (∗an_output_file);
    cur_repl ← (text_pointer) cur_name→equiv;
    cur_byte ← cur_repl→tok_start;
    cur_end ← (cur_repl + 1)→tok_start;
    while (stack_ptr > stack) get_output( );
    flush_buffer( );
  }
```

This code is used in section 42

**44.** If a `@h` was not encountered in the input, we go through the list of replacement texts and copy the ones that refer to macros, preceded by the `#define` preprocessor command.

```
⟨ Output macro definitions if appropriate 44 ⟩ ≡
  if (¬output_defs_seen) output_defs( );
```

This code is used in section 42

**45.**
```
⟨ Global variables 17 ⟩ +≡
  boolean output_defs_seen ← 0;
```

**46.**
```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void output_defs( );
```

**47.**
```
void output_defs( )
{
  sixteen_bits a;
  push_level(Λ);
  for (cur_text ← text_info + 1; cur_text < text_ptr; cur_text++)
    if (cur_text→text_link ≡ 0) {      /* cur_text is the text for a macro */
      cur_byte ← cur_text→tok_start;
      cur_end ← (cur_text + 1)→tok_start;
      C_printf("%s","#define␣");
      out_state ← normal;
      protect ← 1;      /* newlines should be preceded by '\\' */
      while (cur_byte < cur_end) {
        a ← *cur_byte++;
        if (cur_byte ≡ cur_end ∧ a ≡ '\n') break;      /* disregard a final newline */
        if (out_state ≡ verbatim ∧ a ≠ string ∧ a ≠ constant ∧ a ≠ '\n') C_putc(a);
            /* a high-bit character can occur in a string */
        else if (a < °200) out_char(a);      /* one-byte token */
        else {
          a ← (a − °200) * °400 + *cur_byte++;
          if (a < °24000) {      /* °24000 ≡ (°250 − °200) * °400 */
            cur_val ← a;
            out_char(identifier);
          }
          else if (a < °50000) {
            confusion("macro␣defs␣have␣strange␣char");
          }
          else {
            cur_val ← a − °50000;
            cur_section ← cur_val;
            out_char(section_number);
          }      /* no other cases */
        }
      }
      protect ← 0;
      flush_buffer( );
    }
  pop_level(0);
}
```

**48.** A many-way switch is used to send the output. Note that this function is not called if *out_state* ≡ *verbatim*, except perhaps with arguments `'\n'` (protect the newline), *string* (end the string), or *constant* (end the constant).

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void out_char( );
```

**49.**

```
  void out_char(cur_char)
      eight_bits cur_char;
  {
    char *j, *k;      /* pointer into byte_mem */
  restart:
    switch (cur_char) {
    case '\n':
      if (protect) C_putc('␣');
      if (protect ∨ out_state ≡ verbatim) C_putc('\\');
      flush_buffer( );
      if (out_state ≠ verbatim) out_state ← normal;
      break;
    ⟨ Case of an identifier 53 ⟩;
    ⟨ Case of a section number 54 ⟩;
    ⟨ Cases like != 50 ⟩;
    case '=': C_putc('=');
      C_putc('␣');
      out_state ← normal;
      break;
    case join: out_state ← unbreakable;
      break;
    case constant:
      if (out_state ≡ verbatim) {
        out_state ← num_or_id;
        break;
      }
      if (out_state ≡ num_or_id) C_putc('␣');
      out_state ← verbatim;
      break;
    case string:
      if (out_state ≡ verbatim) out_state ← normal;
      else out_state ← verbatim;
      break;
    default: C_putc(cur_char);
      out_state ← normal;
      break;
    }
  }
```

**50.** ⟨ Cases like `!= 50` ⟩ ≡

```
case plus_plus: C_putc('+');
  C_putc('+');
  out_state ← normal;
  break;
case minus_minus: C_putc('-');
  C_putc('-');
  out_state ← normal;
  break;
case minus_gt: C_putc('-');
  C_putc('>');
  out_state ← normal;
  break;
case gt_gt: C_putc('>');
  C_putc('>');
  out_state ← normal;
  break;
case eq_eq: C_putc('=');
  C_putc('=');
  out_state ← normal;
  break;
case lt_lt: C_putc('<');
  C_putc('<');
  out_state ← normal;
  break;
case gt_eq: C_putc('>');
  C_putc('=');
  out_state ← normal;
  break;
case lt_eq: C_putc('<');
  C_putc('=');
  out_state ← normal;
  break;
case not_eq: C_putc('!');
  C_putc('=');
  out_state ← normal;
  break;
case and_and: C_putc('&');
  C_putc('&');
  out_state ← normal;
  break;
case or_or: C_putc('|');
  C_putc('|');
  out_state ← normal;
  break;
case dot_dot_dot: C_putc('.');
  C_putc('.');
  C_putc('.');
  out_state ← normal;
  break;
case colon_colon: C_putc(':');
  C_putc(':');
  out_state ← normal;
  break;
case period_ast: C_putc('.');
```

```
    C_putc('*');
    out_state ← normal;
    break;
case minus_gt_ast: C_putc('-');
    C_putc('>');
    C_putc('*');
    out_state ← normal;
    break;
```

This code is used in section 49

**51.** When an identifier is output to the C file, characters in the range 128–255 must be changed into something else, so the C compiler won't complain. By default, `CTANGLE` converts the character with code $16x+y$ to the three characters '`X`$xy$', but a different transliteration table can be specified. Thus a German might want *grün* to appear as a still readable `gruen`. This makes debugging a lot less confusing.

```
#define translit_length 10
⟨ Global variables 17 ⟩ +≡
  char translit[128][translit_length];
```

**52.** ⟨ Set initial values 18 ⟩ +≡

```
  {
    int i;
    for (i ← 0; i < 128; i++) sprintf(translit[i], "X%02X", (unsigned) (128 + i));
  }
```

**53.** ⟨ Case of an identifier 53 ⟩ ≡

```
case identifier:
  if (out_state ≡ num_or_id) C_putc('␣');
  j ← (cur_val + name_dir)→byte_start;
  k ← (cur_val + name_dir + 1)→byte_start;
  while (j < k) {
    if ((unsigned char) (*j) < °200) C_putc(*j);
    else C_printf("%s", translit[(unsigned char) (*j) − °200]);
    j++;
  }
  out_state ← num_or_id;
  break;
```

This code is used in section 49

**54.** ⟨ Case of a section number 54 ⟩ ≡

```
case section_number:
  if (cur_val > 0) C_printf("/*%d:*/", cur_val);
  else if (cur_val < 0) C_printf("/*:%d*/", -cur_val);
  else if (protect) {
    cur_byte += 4;      /* skip line number and file name */
    cur_char ← '\n';
    goto restart;
  }
  else {
    sixteen_bits a;
    a ← °400 * *cur_byte++;
    a += *cur_byte++;      /* gets the line number */
    C_printf("\n#line␣%d␣\"", a);
    cur_val ← *cur_byte++;
    cur_val ← °400 * (cur_val - °200) + *cur_byte++;      /* points to the file name */
    for (j ← (cur_val + name_dir)→byte_start, k ← (cur_val + name_dir + 1)→byte_start; j < k; j++)
      C_putc(*j);
    C_printf("%s", "\"\n");
  }
  break;
```

This code is used in section 49

**55. Introduction to the input phase.** We have now seen that CTANGLE will be able to output the full C program, if we can only get that program into the byte memory in the proper format. The input process is something like the output process in reverse, since we compress the text as we read it in and we expand it as we write it out.

There are three main input routines. The most interesting is the one that gets the next token of a C text; the other two are used to scan rapidly past TEX text in the CWEB source code. One of the latter routines will jump to the next token that starts with '@', and the other skips to the end of a C comment.

**56.** Control codes in CWEB begin with '@', and the next character identifies the code. Some of these are of interest only to CWEAVE, so CTANGLE ignores them; the others are converted by CTANGLE into internal code numbers by the *ccode* table below. The ordering of these internal code numbers has been chosen to simplify the program logic; larger numbers are given to the control codes that denote more significant milestones.

```
#define ignore 0 /* control code of no interest to CTANGLE */
#define ord °302 /* control code for '@'' */
#define control_text °303 /* control code for '@t', '@^', etc. */
#define translit_code °304 /* control code for '@l' */
#define output_defs_code °305 /* control code for '@h' */
#define format_code °306 /* control code for '@f' */
#define definition °307 /* control code for '@d' */
#define begin_C °310 /* control code for '@c' */
#define section_name °311 /* control code for '@<' */
#define new_section °312 /* control code for '@␣' and '@*' */
⟨ Global variables 17 ⟩ +≡
  eight_bits ccode[256]; /* meaning of a char following @ */
```

**57.** ⟨ Set initial values 18 ⟩ +≡

```
{
  int c; /* must be int so the for loop will end */
  for (c ← 0; c < 256; c++) ccode[c] ← ignore;
  ccode['␣'] ← ccode['\t'] ← ccode['\n'] ← ccode['\v'] ← ccode['\r'] ← ccode['\f'] ←
      ccode['*'] ← new_section;
  ccode['@'] ← '@';
  ccode['='] ← string;
  ccode['d'] ← ccode['D'] ← definition;
  ccode['f'] ← ccode['F'] ← ccode['s'] ← ccode['S'] ← format_code;
  ccode['c'] ← ccode['C'] ← ccode['p'] ← ccode['P'] ← begin_C;
  ccode['^'] ← ccode[':'] ← ccode['.'] ← ccode['t'] ← ccode['T'] ← ccode['q'] ← ccode['Q'] ←
      control_text;
  ccode['h'] ← ccode['H'] ← output_defs_code;
  ccode['l'] ← ccode['L'] ← translit_code;
  ccode['&'] ← join;
  ccode['<'] ← ccode['('] ← section_name;
  ccode['\''] ← ord;
}
```

**58.** The *skip_ahead* procedure reads through the input at fairly high speed until finding the next non-ignorable control code, which it returns.

```
eight_bits skip_ahead( )      /* skip to next control code */
{
  eight_bits c;      /* control code found */
  while (1) {
    if (loc > limit ∧ (get_line( ) ≡ 0)) return (new_section);
    *(limit + 1) ← '@';
    while (*loc ≠ '@') loc++;
    if (loc ≤ limit) {
      loc++;
      c ← ccode[(eight_bits) *loc];
      loc++;
      if (c ≠ ignore ∨ *(loc − 1) ≡ '>') return (c);
    }
  }
}
```

**59.** The *skip_comment* procedure reads through the input at somewhat high speed in order to pass over comments, which CTANGLE does not transmit to the output. If the comment is introduced by `/*`, *skip_comment* proceeds until finding the end-comment token `*/` or a newline; in the latter case *skip_comment* will be called again by *get_next*, since the comment is not finished. This is done so that the each newline in the C part of a section is copied to the output; otherwise the **#line** commands inserted into the C file by the output routines become useless. On the other hand, if the comment is introduced by `//` (i.e., if it is a C++ "short comment"), it always is simply delimited by the next newline. The boolean argument *is_long_comment* distinguishes between the two types of comments.

If *skip_comment* comes to the end of the section, it prints an error message. No comment, long or short, is allowed to contain '`@ `' or '`@*`'.

⟨ Global variables 17 ⟩ +≡

```
boolean comment_continues ← 0;      /* are we scanning a comment? */
```

```
60. int skip_comment(is_long_comment)      /* skips over comments */
    boolean is_long_comment;
{
  char c;     /* current character */
  while (1) {
    if (loc > limit) {
      if (is_long_comment) {
        if (get_line()) return (comment_continues ← 1);
        else {
          err_print("! Input ended in mid-comment");
          return (comment_continues ← 0);
        }
      }
      else return (comment_continues ← 0);
    }
    c ← *(loc++);
    if (is_long_comment ∧ c ≡ '*' ∧ *loc ≡ '/') {
      loc++;
      return (comment_continues ← 0);
    }
    if (c ≡ '@') {
      if (ccode[(eight_bits) *loc] ≡ new_section) {
        err_print("! Section name ended in mid-comment");
        loc−−;
        return (comment_continues ← 0);
      }
      else loc++;
    }
  }
}
```

**61. Inputting the next token.**

```
#define constant °3
⟨ Global variables 17 ⟩ +≡
  name_pointer cur_section_name;      /* name of section just scanned */
```

**62.** ⟨ Include files 6 ⟩ +≡

```
#include <ctype.h>      /* definition of isalpha, isdigit and so on */
#include <stdlib.h>     /* definition of exit */
```

**63.** As one might expect, *get_next* consists mostly of a big switch that branches to the various special cases that can arise.

```
#define isxalpha(c) ((c) ≡ '_')      /* non-alpha character allowed in identifier */
#define ishigh(c) ((unsigned char) (c) > °177)
  eight_bits get_next( )      /* produces the next input token */
  {
    static int preprocessing ← 0;
    eight_bits c;      /* the current character */
    while (1) {
      if (loc > limit) {
        if (preprocessing ∧ *(limit − 1) ≠ '\\') preprocessing ← 0;
        if (get_line( ) ≡ 0) return (new_section);
        else if (print_where) {
          print_where ← 0;
          ⟨ Insert the line number into tok_mem 77 ⟩;
        }
        else return ('\n');
      }
      c ← *loc;
      if (comment_continues ∨ (c ≡ '/' ∧ (*(loc + 1) ≡ '*' ∨ *(loc + 1) ≡ '/'))) {
        skip_comment(comment_continues ∨ *(loc + 1) ≡ '*');
          /* scan to end of comment or newline */
        if (comment_continues) return ('\n');
        else continue;
      }
      loc++;
      if (xisdigit(c) ∨ c ≡ '\\' ∨ c ≡ '.') ⟨ Get a constant 66 ⟩
      else if (c ≡ '\'' ∨ c ≡ '"' ∨ (c ≡ 'L' ∧ (*loc ≡ '\'' ∨ *loc ≡ '"'))) ⟨ Get a string 67 ⟩
      else if (isalpha(c) ∨ isxalpha(c) ∨ ishigh(c)) ⟨ Get an identifier 65 ⟩
      else if (c ≡ '@') ⟨ Get control code and possible section name 68 ⟩
      else if (xisspace(c)) {
        if (¬preprocessing ∨ loc > limit) continue;
            /* we don't want a blank after a final backslash */
        else return ('␣');      /* ignore spaces and tabs, unless preprocessing */
      }
      else if (c ≡ '#' ∧ loc ≡ buffer + 1) preprocessing ← 1;
    mistake: ⟨ Compress two-symbol operator 64 ⟩
      return (c);
    }
  }
```

**64.** The following code assigns values to the combinations `++`, `--`, `->`, `>=`, `<=`, `==`, `<<`, `>>`, `!=`, and `&&`, and to the C++ combinations `...`, `::`, `.*` and `->*`. The compound assignment operators (e.g., `+=`) are treated as separate tokens.

```
#define compress(c) if (loc++ ≤ limit) return (c)
⟨ Compress two-symbol operator 64 ⟩ ≡
  switch (c) {
  case '+':
    if (*loc ≡ '+') compress(plus_plus);
    break;
  case '-':
    if (*loc ≡ '-') {
      compress(minus_minus);
    }
    else if (*loc ≡ '>')
      if (*(loc + 1) ≡ '*') {
        loc++;
        compress(minus_gt_ast);
      }
      else compress(minus_gt);
    break;
  case '.':
    if (*loc ≡ '*') {
      compress(period_ast);
    }
    else if (*loc ≡ '.' ∧ *(loc + 1) ≡ '.') {
      loc++;
      compress(dot_dot_dot);
    }
    break;
  case ':':
    if (*loc ≡ ':') compress(colon_colon);
    break;
  case '=':
    if (*loc ≡ '=') compress(eq_eq);
    break;
  case '>':
    if (*loc ≡ '=') {
      compress(gt_eq);
    }
    else if (*loc ≡ '>') compress(gt_gt);
    break;
  case '<':
    if (*loc ≡ '=') {
      compress(lt_eq);
    }
    else if (*loc ≡ '<') compress(lt_lt);
    break;
  case '&':
    if (*loc ≡ '&') compress(and_and);
    break;
  case '|':
    if (*loc ≡ '|') compress(or_or);
    break;
  case '!':
    if (*loc ≡ '=') compress(not_eq);
```

```
    break;
  }
```

This code is used in section 63

**65.** ⟨ Get an identifier 65 ⟩ ≡

```
  {
    id_first ← --loc;
    while (isalpha(*++loc) ∨ isdigit(*loc) ∨ isxalpha(*loc) ∨ ishigh(*loc)) ;
    id_loc ← loc;
    return (identifier);
  }
```

This code is used in section 63

**66.** ⟨ Get a constant 66 ⟩ ≡

```
  {
    id_first ← loc − 1;
    if (*id_first ≡ '.' ∧ ¬xisdigit(*loc)) goto mistake;      /* not a constant */
    if (*id_first ≡ '\\')
      while (xisdigit(*loc)) loc++;      /* octal constant */
    else {
      if (*id_first ≡ '0') {
        if (*loc ≡ 'x' ∨ *loc ≡ 'X') {      /* hex constant */
          loc++;
          while (xisxdigit(*loc)) loc++;
          goto found;
        }
      }
      while (xisdigit(*loc)) loc++;
      if (*loc ≡ '.') {
        loc++;
        while (xisdigit(*loc)) loc++;
      }
      if (*loc ≡ 'e' ∨ *loc ≡ 'E') {      /* float constant */
        if (*++loc ≡ '+' ∨ *loc ≡ '-') loc++;
        while (xisdigit(*loc)) loc++;
      }
    }
  found:
    while (*loc ≡ 'u' ∨ *loc ≡ 'U' ∨ *loc ≡ 'l' ∨ *loc ≡ 'L' ∨ *loc ≡ 'f' ∨ *loc ≡ 'F') loc++;
    id_loc ← loc;
    return (constant);
  }
```

This code is used in section 63

**67.** 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 67 ⟩ ≡
  {
    char delim ← c;      /* what started the string */
    id_first ← section_text + 1;
    id_loc ← section_text;
    *++id_loc ← delim;
    if (delim ≡ 'L') {      /* wide character constant */
      delim ← *loc++;
      *++id_loc ← delim;
    }
    while (1) {
      if (loc ≥ limit) {
        if (*(limit − 1) ≠ '\\') {
          err_print("! String didn't end");
          loc ← limit;
          break;
        }
        if (get_line() ≡ 0) {
          err_print("! Input ended in middle of string");
          loc ← buffer;
          break;
        }
        else if (++id_loc ≤ section_text_end) *id_loc ← '\n';      /* will print as "\\\n" */
      }
      if ((c ← *loc++) ≡ delim) {
        if (++id_loc ≤ section_text_end) *id_loc ← c;
        break;
      }
      if (c ≡ '\\') {
        if (loc ≥ limit) continue;
        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) {
      printf("\n! String too long: ");
      term_write(section_text + 1, 25);
      err_print("...");
    }
    id_loc++;
    return (string);
  }
```

This code is used in section 63

**68.** 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 68 ⟩ ≡
  {
    c ← ccode[(eight_bits) *loc++];
    switch (c) {
    case ignore: continue;
    case output_defs_code: output_defs_seen ← 1;
      return (c);
    case translit_code: err_print("! Use @l in limbo only");
      continue;
    case control_text:
      while ((c ← skip_ahead()) ≡ '@') ;      /* only @@ and @> are expected */
      if (*(loc − 1) ≠ '>') err_print("! Double @ should be used in control text");
      continue;
    case section_name: cur_section_name_char ← *(loc − 1);
      ⟨ Scan the section name and make cur_section_name point to it 70 ⟩;
    case string: ⟨ Scan a verbatim string 74 ⟩;
    case ord: ⟨ Scan an ASCII constant 69 ⟩;
    default: return (c);
    }
  }
```

This code is cited in section 84

This code is used in section 63

**69.** After scanning a valid ASCII constant that follows @', this code plows ahead until it finds the next single quote. (Special care is taken if the quote is part of the constant.) Anything after a valid ASCII constant is ignored; thus, @'\nopq' gives the same result as @'\n'.

```
⟨ Scan an ASCII constant 69 ⟩ ≡
  id_first ← loc;
  if (*loc ≡ '\\') {
    if (*++loc ≡ '\'') loc++;
  }
  while (*loc ≠ '\'') {
    if (*loc ≡ '@') {
      if (*(loc + 1) ≠ '@') err_print("! Double @ should be used in ASCII constant");
      else loc++;
    }
    loc++;
    if (loc > limit) {
      err_print("! String didn't end");
      loc ← limit − 1;
      break;
    }
  }
  loc++;
  return (ord);
```

This code is used in section 68

**70.** ⟨ Scan the section name and make *cur_section_name* point to it 70 ⟩ ≡

```
{
  char *k;      /* pointer into section_text */
  ⟨ Put section name into section_text 72 ⟩;
  if (k - section_text > 3 ∧ strncmp(k - 2, "...", 3) ≡ 0)
    cur_section_name ← section_lookup(section_text + 1, k - 3, 1);      /* 1 means is a prefix */
  else cur_section_name ← section_lookup(section_text + 1, k, 0);
  if (cur_section_name_char ≡ '(') ⟨ If it's not there, add cur_section_name to the output file stack,
        or complain we're out of room 40 ⟩;
  return (section_name);
}
```

This code is used in section 68

**71.** Section names are placed into the *section_text* array with consecutive spaces, tabs, and carriage-returns replaced by single spaces. There will be no spaces at the beginning or the end. (We set *section_text*[0] ← '␣' to facilitate this, since the *section_lookup* routine uses *section_text*[1] as the first character of the name.)

⟨ Set initial values 18 ⟩ +≡

```
section_text[0] ← '␣';
```

**72.** ⟨ Put section name into *section_text* 72 ⟩ ≡

```
k ← section_text;
while (1) {
  if (loc > limit ∧ get_line() ≡ 0) {
    err_print("!␣Input␣ended␣in␣section␣name");
    loc ← buffer + 1;
    break;
  }
  c ← *loc;
  ⟨ If end of name or erroneous nesting, break 73 ⟩;
  loc++;
  if (k < section_text_end) k++;
  if (xisspace(c)) {
    c ← '␣';
    if (*(k - 1) ≡ '␣') k--;
  }
  *k ← c;
}
if (k ≥ section_text_end) {
  printf("\n!␣Section␣name␣too␣long:␣");
  term_write(section_text + 1, 25);
  printf("...");
  mark_harmless;
}
if (*k ≡ '␣' ∧ k > section_text) k--;
```

This code is used in section 70

**73.** ⟨ If end of name or erroneous nesting, **break** 73 ⟩ ≡

```
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 (ccode[(eight_bits) c] ≡ section_name) {
    err_print("! Nesting of section names not allowed");
    break;
  }
  *(++k) ← '@';
  loc++;      /* now c ≡ *loc again */
}
```

This code is used in section 72

**74.** At the present point in the program we have $*(loc - 1) \equiv string$; 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 74 ⟩ ≡

```
{
  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 (string);
}
```

This code is used in section 68

**75. Scanning a macro definition.** The rules for generating the replacement texts corresponding to macros and C texts of a section are almost identical; the only differences are that

a) Section names are not allowed in macros; in fact, the appearance of a section name terminates such macros and denotes the name of the current section.
b) The symbols `@d` and `@f` and `@c` are not allowed after section names, while they terminate macro definitions.

Therefore there is a single procedure *scan_repl* whose parameter $t$ specifies either *macro* or *section_name*. After *scan_repl* has acted, *cur_text* will point to the replacement text just generated, and *next_control* will contain the control code that terminated the activity.

```
#define macro 0
#define app_repl(c)
        {
          if (tok_ptr ≡ tok_mem_end) overflow("token");
          *tok_ptr++ ← c;
        }
⟨ Global variables 17 ⟩ +≡
  text_pointer cur_text;      /* replacement text formed by scan_repl */
  eight_bits next_control;

76.  void scan_repl(t)      /* creates a replacement text */
       eight_bits t;
  {
    sixteen_bits a;      /* the current token */
    if (t ≡ section_name) {
      ⟨ Insert the line number into tok_mem 77 ⟩;
    }
    while (1)
      switch (a ← get_next( )) {
        ⟨ In cases that a is a non-char token (identifier, section_name, etc.), either process it and
            change a to a byte that should be stored, or continue if a should be ignored, or goto
            done if a signals the end of this replacement text 78 ⟩
      default: app_repl(a);      /* store a in tok_mem */
      }
  done: next_control ← (eight_bits) a;
    if (text_ptr > text_info_end) overflow("text");
    cur_text ← text_ptr;
    (++text_ptr)→tok_start ← tok_ptr;
  }
```

**77.** Here is the code for the line number: first a **sixteen_bits** equal to $^\circ 150000$; then the numeric line number; then a pointer to the file name.

```
⟨ Insert the line number into tok_mem 77 ⟩ ≡
  store_two_bytes(°150000);
  if (changing) id_first ← change_file_name;
  else id_first ← cur_file_name;
  id_loc ← id_first + strlen(id_first);
  if (changing) store_two_bytes((sixteen_bits) change_line);
  else store_two_bytes((sixteen_bits) cur_line);
  {
    int a ← id_lookup(id_first, id_loc) − name_dir;
    app_repl((a/°400) + °200);
    app_repl(a % °400);
  }
```

This code is used in sections 63, 76, and 78

**78.** ⟨ In cases that $a$ is a non-**char** token (*identifier*, *section_name*, etc.), either process it and change $a$ to a byte that should be stored, or **continue** if $a$ should be ignored, or **goto** *done* if $a$ signals the end of this replacement text 78 ⟩ ≡

```
case identifier: a ← id_lookup(id_first, id_loc) − name_dir;
  app_repl((a/°400) + °200);
  app_repl(a % °400);
  break;
case section_name:
  if (t ≠ section_name) goto done;
  else {
    ⟨ Was an '@' missed here? 79 ⟩;
    a ← cur_section_name − name_dir;
    app_repl((a/°400) + °250);
    app_repl(a % °400);
    ⟨ Insert the line number into tok_mem 77 ⟩;
    break;
  }
case output_defs_code: a ← output_defs_flag;
  app_repl((a/°400) + °200);
  app_repl(a % °400);
  ⟨ Insert the line number into tok_mem 77 ⟩;
  break;
case constant: case string: ⟨ Copy a string or verbatim construction or numerical constant 80 ⟩;
case ord: ⟨ Copy an ASCII constant 81 ⟩;
case definition: case format_code: case begin_C:
  if (t ≠ section_name) goto done;
  else {
    err_print("! @d, @f and @c are ignored in C text");
    continue;
  }
case new_section: goto done;
```

This code is used in section 76

**79.** ⟨ Was an '@' missed here? 79 ⟩ ≡

```
{
  char *try_loc ← loc;
  while (*try_loc ≡ ' ' ∧ try_loc < limit) try_loc++;
  if (*try_loc ≡ '+' ∧ try_loc < limit) try_loc++;
  while (*try_loc ≡ ' ' ∧ try_loc < limit) try_loc++;
  if (*try_loc ≡ '=') err_print("! Missing ' @ ' before a named section");   /* user who isn't
      defining a section should put newline after the name, as explained in the manual */
}
```

This code is used in section 78

**80.** ⟨ Copy a string or verbatim construction or numerical constant 80 ⟩ ≡

```
app_repl(a);      /* string or constant */
while (id_first < id_loc) {      /* simplify @@ pairs */
  if (*id_first ≡ '@') {
    if (*(id_first + 1) ≡ '@') id_first++;
    else err_print("! Double @ should be used in string");
  }
  app_repl(*id_first++);
}
app_repl(a);
break;
```

This code is used in section 78

**81.** This section should be rewritten on machines that don't use ASCII code internally.

```
⟨ Copy an ASCII constant 81 ⟩ ≡
  {
    int c ← (eight_bits) *id_first;
    if (c ≡ '\\') {
      c ← *++id_first;
      if (c ≥ '0' ∧ c ≤ '7') {
        c −= '0';
        if (*(id_first + 1) ≥ '0' ∧ *(id_first + 1) ≤ '7') {
          c ← 8 * c + *(++id_first) − '0';
          if (*(id_first + 1) ≥ '0' ∧ *(id_first + 1) ≤ '7' ∧ c < 32) c ← 8 * c + *(++id_first) − '0';
        }
      }
      else
        switch (c) {
        case 't': c ← '\t'; break;
        case 'n': c ← '\n'; break;
        case 'b': c ← '\b'; break;
        case 'f': c ← '\f'; break;
        case 'v': c ← '\v'; break;
        case 'r': c ← '\r'; break;
        case 'a': c ← '\7'; break;
        case '?': c ← '?'; break;
        case 'x':
          if (xisdigit(*(id_first + 1))) c ← *(++id_first) − '0';
          else if (xisxdigit(*(id_first + 1))) {
            ++id_first;
            c ← toupper(*id_first) − 'A' + 10;
          }
          if (xisdigit(*(id_first + 1))) c ← 16 * c + *(++id_first) − '0';
          else if (xisxdigit(*(id_first + 1))) {
            ++id_first;
            c ← 16 * c + toupper(*id_first) − 'A' + 10;
          }
          break;
        case '\\': c ← '\\'; break;
        case '\'': c ← '\''; break;
        case '\"': c ← '\"'; break;
        default: err_print("! Unrecognized escape sequence");
        }
    }     /* at this point c should have been converted to its ASCII code number */
    app_repl(constant);
    if (c ≥ 100) app_repl('0' + c/100);
    if (c ≥ 10) app_repl('0' + (c/10) % 10);
    app_repl('0' + c % 10);
    app_repl(constant);
  }
  break;
```

This code is used in section 78

**82. Scanning a section.** The *scan_section* procedure starts when '@␣' or '@*' has been sensed in the input, and it proceeds until the end of that section. It uses *section_count* to keep track of the current section number; with luck, CWEAVE and CTANGLE will both assign the same numbers to sections.

```
⟨ Global variables 17 ⟩ +≡
  extern sixteen_bits section_count;      /* the current section number */
```

**83.** The body of *scan_section* is a loop where we look for control codes that are significant to CTANGLE: those that delimit a definition, the C part of a module, or a new module.

```
void scan_section( )
{
  name_pointer p;      /* section name for the current section */
  text_pointer q;      /* text for the current section */
  sixteen_bits a;      /* token for left-hand side of definition */
  section_count++;
  if (*(loc − 1) ≡ '*' ∧ show_progress) {      /* starred section */
    printf("*%d", section_count);
    update_terminal;
  }
  next_control ← 0;
  while (1) {
    ⟨ Skip ahead until next_control corresponds to @d, @<, @␣ or the like 84 ⟩;
    if (next_control ≡ definition) {      /* @d */
      ⟨ Scan a definition 85 ⟩
      continue;
    }
    if (next_control ≡ begin_C) {      /* @c or @p */
      p ← name_dir;
      break;
    }
    if (next_control ≡ section_name) {      /* @< or @( */
      p ← cur_section_name;
      ⟨ If section is not being defined, continue 86 ⟩;
      break;
    }
    return;      /* @␣ or @* */
  }
  ⟨ Scan the C part of the current section 87 ⟩;
}
```

**84.** At the top of this loop, if *next_control* ≡ *section_name*, the section name has already been scanned (see ⟨ Get control code and possible section name 68 ⟩). Thus, if we encounter *next_control* ≡ *section_name* in the skip-ahead process, we should likewise scan the section name, so later processing will be the same in both cases.

```
⟨ Skip ahead until next_control corresponds to @d, @<, @␣ or the like 84 ⟩ ≡
  while (next_control < definition)      /* definition is the lowest of the "significant" codes */
    if ((next_control ← skip_ahead( )) ≡ section_name) {
      loc −= 2;
      next_control ← get_next( );
    }
```

This code is used in section 83

**85.** ⟨ Scan a definition 85 ⟩ ≡

```
{
  while ((next_control ← get_next( )) ≡ '\n') ;      /* allow newline before definition */
  if (next_control ≠ identifier) {
    err_print("! Definition flushed, must start with identifier");
    continue;
  }
  app_repl(((a ← id_lookup(id_first, id_loc) − name_dir)/°400) + °200);      /* append the lhs */
  app_repl(a % °400);
  if (*loc ≠ '(') {      /* identifier must be separated from replacement text */
    app_repl(string);
    app_repl(' ');
    app_repl(string);
  }
  print_where ← 0;
  scan_repl(macro);
  cur_text→text_link ← 0;      /* text_link ≡ 0 characterizes a macro */
}
```

This code is used in section 83

**86.** If the section name is not followed by = or +=, no C code is forthcoming: the section is being cited, not being defined. This use is illegal after the definition part of the current section has started, except inside a comment, but CTANGLE does not enforce this rule: it simply ignores the offending section name and everything following it, up to the next significant control code.

⟨ If section is not being defined, **continue** 86 ⟩ ≡

```
while ((next_control ← get_next( )) ≡ '+') ;      /* allow optional += */
if (next_control ≠ '=' ∧ next_control ≠ eq_eq) continue;
```

This code is used in section 83

**87.** ⟨ Scan the C part of the current section 87 ⟩ ≡

```
⟨ Insert the section number into tok_mem 88 ⟩;
scan_repl(section_name);      /* now cur_text points to the replacement text */
⟨ Update the data structure so that the replacement text is accessible 89 ⟩;
```

This code is used in section 83

**88.** ⟨ Insert the section number into *tok_mem* 88 ⟩ ≡

```
store_two_bytes((sixteen_bits) (°150000 + section_count));      /* °150000 ≡ °320 * °400 */
```

This code is used in section 87

**89.** ⟨ Update the data structure so that the replacement text is accessible 89 ⟩ ≡

```
if (p ≡ name_dir ∨ p ≡ 0) {      /* unnamed section, or bad section name */
  (last_unnamed)→text_link ← cur_text − text_info;
  last_unnamed ← cur_text;
}
else if (p→equiv ≡ (char *) text_info) p→equiv ← (char *) cur_text;      /* first section of this name */
else {
  q ← (text_pointer) p→equiv;
  while (q→text_link < section_flag) q ← q→text_link + text_info;      /* find end of list */
  q→text_link ← cur_text − text_info;
}
cur_text→text_link ← section_flag;      /* mark this replacement text as a nonmacro */
```

This code is used in section 87

**90.** ⟨ Predeclaration of procedures 2 ⟩ +≡

```
void phase_one( );
```

**91.** **void** *phase_one*( )
```
{
  phase ← 1;
  section_count ← 0;
  reset_input( );
  skip_limbo( );
  while (¬input_has_ended) scan_section( );
  check_complete( );
  phase ← 2;
}
```

**92.** Only a small subset of the control codes is legal in limbo, so limbo processing is straightforward.

⟨ Predeclaration of procedures 2 ⟩ +≡
```
void skip_limbo( );
```

**93.** **void** *skip_limbo*( )
```
{
  char c;
  while (1) {
    if (loc > limit ∧ get_line( ) ≡ 0) return;
    *(limit + 1) ← '@';
    while (*loc ≠ '@') loc++;
    if (loc++ ≤ limit) {
      c ← *loc++;
      if (ccode[(eight_bits) c] ≡ new_section) break;
      switch (ccode[(eight_bits) c]) {
      case translit_code: ⟨ Read in transliteration of a character 94 ⟩;
        break;
      case format_code: case '@': break;
      case control_text:
        if (c ≡ 'q' ∨ c ≡ 'Q') {
          while ((c ← skip_ahead( )) ≡ '@') ;
          if (*(loc − 1) ≠ '>') err_print("! Double @ should be used in control text");
          break;
        }     /* otherwise fall through */
      default: err_print("! Double @ should be used in limbo");
      }
    }
  }
}
```

**94.** ⟨ Read in transliteration of a character 94 ⟩ ≡

```
while (xisspace(*loc) ∧ loc < limit) loc++;
loc += 3;
if (loc > limit ∨ ¬xisxdigit(*(loc − 3)) ∨ ¬xisxdigit(*(loc − 2))
        ∨ (*(loc − 3) ≥ '0' ∧ *(loc − 3) ≤ '7') ∨ ¬xisspace(*(loc − 1)))
  err_print("! Improper hex number following @l");
else {
  unsigned i;
  char *beg;
  sscanf(loc − 3, "%x", &i);
  while (xisspace(*loc) ∧ loc < limit) loc++;
  beg ← loc;
  while (loc < limit ∧ (xisalpha(*loc) ∨ xisdigit(*loc) ∨ *loc ≡ '_')) loc++;
  if (loc − beg ≥ translit_length) err_print("! Replacement string in @l too long");
  else {
    strncpy(translit[i − °200], beg, loc − beg);
    translit[i − °200][loc − beg] ← '\0';
  }
}
```

This code is used in section 93

**95.** Because on some systems the difference between two pointers is a **long** but not an **int**, we use `%ld` to print these quantities.

```
void print_stats( )
{
  printf("\nMemory usage statistics:\n");
  printf("%ld names (out of %ld)\n", (long) (name_ptr − name_dir), (long) max_names);
  printf("%ld replacement texts (out of %ld)\n", (long) (text_ptr − text_info), (long)
      max_texts);
  printf("%ld bytes (out of %ld)\n", (long) (byte_ptr − byte_mem), (long) max_bytes);
  printf("%ld tokens (out of %ld)\n", (long) (tok_ptr − tok_mem), (long) max_toks);
}
```

**96. Index.** Here is a cross-reference table for CTANGLE. 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 and a few other things like "ASCII code dependencies" are indexed here too.

## Index of ctangle

**O**



**P**



**Q**



**R**



**S**

## Contents of cweave

## Sections of cweave

⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section definition 217 ⟩ Used in section 216
⟨ Clear *bal* and **return** 95 ⟩ Used in section 92
⟨ Combine the irreducible scraps that remain 170 ⟩ Used in section 169
⟨ Common code for CWEAVE and CTANGLE 5, 7, 8, 9, 10, 11, 12, 13, 14, 15 ⟩ Used in section 1
⟨ Compress two-symbol operator 46 ⟩ Used in section 40
⟨ Copy a control code into the buffer 203 ⟩ Used in section 202
⟨ Copy special things when $c \equiv$ '@', '\\' 94 ⟩ Used in section 92
⟨ Copy the C text into the *buffer* array 202 ⟩ Used in section 200
⟨ Do the first pass of sorting 229 ⟩ Used in section 225
⟨ Emit the scrap for a section name if present 218 ⟩ Used in section 216
⟨ Get a constant 48 ⟩ Used in section 40
⟨ Get a string 49 ⟩ Used in sections 40 and 50
⟨ Get an identifier 47 ⟩ Used in section 40
⟨ Get control code and possible section name 50 ⟩ Used in section 40
⟨ Global variables 17, 19, 25, 31, 37, 41, 43, 58, 68, 73, 77, 97, 104, 108, 167, 186, 190, 206, 215, 226, 228, 232, 234, 243 ⟩ Used in section 1
⟨ If end of name or erroneous control code, **break** 54 ⟩ Used in section 53
⟨ If semi-tracing, show the irreducible scraps 171 ⟩ Used in section 170
⟨ If tracing, print an indication of where we are 172 ⟩ Used in section 169
⟨ Include files 6, 38 ⟩ Used in section 1
⟨ Insert new cross-reference at $q$, not at beginning of list 116 ⟩ Used in section 115
⟨ Invert the cross-reference list at *cur_name*, making *cur_xref* the head 244 ⟩ Used in section 242
⟨ Look ahead for strongest line break, **goto** *reswitch* 197 ⟩ Used in section 196
⟨ Make sure that there is room for the new scraps, tokens, and texts 176 ⟩ Used in sections 175 and 183
⟨ Make sure the entries *pp* through $pp + 3$ of *cat* are defined 166 ⟩ Used in section 165
⟨ Match a production at *pp*, or increase *pp* if there is no match 110 ⟩ Used in section 165
⟨ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 196 ⟩ Used in section 194
⟨ Output a section name 199 ⟩ Used in section 194
⟨ Output all the section names 247 ⟩ Used in section 225
⟨ Output all the section numbers on the reference list *cur_xref* 222 ⟩ Used in section 221
⟨ Output an identifier 195 ⟩ Used in section 194
⟨ Output index entries for the list at *sort_ptr* 240 ⟩ Used in section 238
⟨ Output saved *indent* or *outdent* tokens 198 ⟩ Used in sections 194 and 197
⟨ Output the code for the beginning of a new section 208 ⟩ Used in section 207
⟨ Output the code for the end of a section 223 ⟩ Used in section 207
⟨ Output the cross-references at *cur_name* 242 ⟩ Used in section 240
⟨ Output the name at *cur_name* 241 ⟩ Used in section 240
⟨ Output the text of the section name 200 ⟩ Used in section 199
⟨ Predeclaration of procedures 2, 34, 39, 55, 59, 62, 64, 74, 83, 91, 114, 180, 193, 204, 211, 220, 224, 236, 245 ⟩ Used in section 1
⟨ Print a snapshot of the scrap list if debugging 168 ⟩ Used in sections 162 and 164
⟨ Print error messages about unused or undefined section names 76 ⟩ Used in section 60
⟨ Print token $r$ in symbolic form 107 ⟩ Used in section 106
⟨ Print warning message, break the line, **return** 85 ⟩ Used in section 84
⟨ Process a format definition 70 ⟩ Used in section 69
⟨ Process simple format in limbo 71 ⟩ Used in section 35
⟨ Put section name into *section_text* 53 ⟩ Used in section 51
⟨ Raise preprocessor flag 42 ⟩ Used in section 40
⟨ Reduce the scraps using the productions until no more rules apply 165 ⟩ Used in section 169
⟨ Replace "@@" by "@" 67 ⟩ Used in sections 63 and 66
⟨ Rest of *trans_plus* union 231 ⟩ Used in section 103
⟨ Scan a verbatim string 57 ⟩ Used in section 50
⟨ Scan the section name and make *cur_section* point to it 51 ⟩ Used in section 50
⟨ Set initial values 20, 26, 32, 52, 80, 82, 98, 105, 187, 233, 235 ⟩ Used in section 3

⟨ Show cross-references to this section 219 ⟩ Used in section 207
⟨ Skip next character, give error if not '@' 201 ⟩ Used in section 200
⟨ Sort and output the index 238 ⟩ Used in section 225
⟨ Special control codes for debugging 33 ⟩ Used in section 32
⟨ Split the list at *sort_ptr* into further lists 239 ⟩ Used in section 238
⟨ Start a format definition 214 ⟩ Used in section 210
⟨ Start a macro definition 213 ⟩ Used in section 210
⟨ Store all the reserved words 28 ⟩ Used in section 3
⟨ Store cross-reference data for the current section 61 ⟩ Used in section 60
⟨ Store cross-references in the C part of a section 72 ⟩ Used in section 61
⟨ Store cross-references in the TEX part of a section 66 ⟩ Used in section 61
⟨ Store cross-references in the definition part of a section 69 ⟩ Used in section 61
⟨ Tell about changed sections 227 ⟩ Used in section 225
⟨ Translate the C part of the current section 216 ⟩ Used in section 207
⟨ Translate the TEX part of the current section 209 ⟩ Used in section 207
⟨ Translate the current section 207 ⟩ Used in section 205
⟨ Translate the definition part of the current section 210 ⟩ Used in section 207
⟨ Typedef declarations 18, 24, 103, 185 ⟩ Used in section 1

**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 C++ adaptation is due), Klaus Guntermann, Norman Ramsey, Tomas Rokicki, Joachim Schnitter, Joachim Schrod, Lee Wittenberg, and others who have contributed improvements.

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

```
#define banner "This␣is␣CWEAVE␣(Version␣3.1)\n"
```

⟨ Include files 6 ⟩
⟨ Preprocessor definitions ⟩
⟨ Common code for CWEAVE and CTANGLE 5 ⟩
⟨ Typedef declarations 18 ⟩
⟨ Global variables 17 ⟩
⟨ Predeclaration of procedures 2 ⟩

**2.** We predeclare several standard system functions here instead of including their system header files, because the names of the header files are not as standard as the names of the functions. (For example, some C environments have <string.h> where others have <strings.h>.)

⟨ Predeclaration of procedures 2 ⟩ ≡

```
extern int strlen( );      /* length of string */
extern int strcmp( );      /* compare strings lexicographically */
extern char *strcpy( );    /* copy one string to another */
extern int strncmp( );     /* compare up to n string characters */
extern char *strncpy( );   /* copy up to n string characters */
```

See also sections 34, 39, 55, 59, 62, 64, 74, 83, 91, 114, 180, 193, 204, 211, 220, 224, 236, and 245

This code is used in section 1

**3.** 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(ac, av)
    int ac;     /* argument count */
    char **av;  /* argument values */
{
  argc ← ac;
  argv ← av;
  program ← cweave;
  make_xrefs ← force_lines ← 1;     /* controlled by command-line options */
  common_init( );
  ⟨ Set initial values 20 ⟩;
  if (show_banner) printf(banner);     /* print a "banner line" */
  ⟨ Store all the reserved words 28 ⟩;
  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 */
  return wrap_up( );     /* and exit gracefully */
}
```

**4.** The following parameters were sufficient in the original WEAVE to handle TEX, so they should be sufficient for most applications of CWEAVE. If you change *max_bytes*, *max_names*, *hash_size* or *buf_size* you have to change them also in the file "common.w".

```
#define max_bytes 90000 /* the number of bytes in identifiers, index entries, and section names */
#define max_names 4000 /* number of identifiers, strings, section names; must be less than 10240;
    used in "common.w" */
#define max_sections 2000 /* greater than the total number of sections */
#define hash_size 353 /* should be prime */
#define buf_size 100 /* maximum length of input line, plus one */
#define longest_name 1000 /* section names and strings shouldn't be longer than this */
#define long_buf_size (buf_size + longest_name)
#define line_length 80
    /* lines of TEX output have at most this many characters; should be less than 256 */
#define max_refs 20000 /* number of cross-references; must be less than 65536 */
#define max_toks 20000 /* number of symbols in C texts being parsed; must be less than 65536 */
#define max_texts 4000 /* number of phrases in C texts being parsed; must be less than 10240 */
#define max_scraps 2000 /* number of tokens in C texts being parsed */
#define stack_size 400 /* number of simultaneous output levels */
```

**5.** 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 needs to be updated when "common.w" changes.

First comes general stuff:

```
#define ctangle 0
#define cweave 1
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ ≡
  typedef short boolean;
  typedef char unsigned eight_bits;
  extern boolean program; /* CWEAVE or CTANGLE? */
  extern int phase; /* which phase are we in? */
```

See also sections 7, 8, 9, 10, 11, 12, 13, 14, and 15

This code is used in section 1

**6.** ⟨ Include files 6 ⟩ ≡

```
#include <stdio.h>
```

See also section 38

This code is used in section 1

**7.** 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 not_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 ϖ */
#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 ↰ */
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  char section_text[longest_name + 1]; /* name being sought for */
  char *section_text_end ← section_text + longest_name; /* end of section_text */
  char *id_first; /* where the current identifier begins in the buffer */
  char *id_loc; /* just after the current identifier in the buffer */
```

**8.** Code related to input routines:

```
#define xisalpha(c) (isalpha(c) ∧ ((eight_bits) c < °200))
#define xisdigit(c) (isdigit(c) ∧ ((eight_bits) c < °200))
#define xisspace(c) (isspace(c) ∧ ((eight_bits) c < °200))
#define xislower(c) (islower(c) ∧ ((eight_bits) c < °200))
#define xisupper(c) (isupper(c) ∧ ((eight_bits) c < °200))
#define xisxdigit(c) (isxdigit(c) ∧ ((eight_bits) c < °200))
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  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 */
```

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

```
#define length(c) (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 */
#define chunk_marker 0
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  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;
    char *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 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 byte_start */
  extern char *byte_ptr; /* first unused position in byte_mem */
  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 */
  extern name_pointer id_lookup(); /* looks up a string in the identifier table */
  extern name_pointer section_lookup(); /* finds section name */
  extern void print_section_name(), sprint_section_name();
```

**10.** 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 5 ⟩ +≡
  extern history; /* indicates how bad this run was */
  extern err_print(); /* print error message and context */
  extern wrap_up(); /* indicate history and exit */
  extern void fatal(); /* issue error message and die */
  extern void overflow(); /* succumb because a table has overflowed */
```

**11.** Code related to file handling:

```
  format line x /* make line an unreserved word */
#define max_file_name_length 60
#define cur_file file[include_depth] /* current file */
#define cur_file_name file_name[include_depth] /* current file name */
#define web_file_name file_name[0] /* main source file name */
#define cur_line line[include_depth] /* number of current line in current file */
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  extern include_depth; /* current level of nesting */
  extern FILE *file[]; /* stack of non-change files */
  extern FILE *change_file; /* change file */
  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 char file_name[][max_file_name_length]; /* stack of non-change file names */
  extern char change_file_name[]; /* name of change file */
  extern line[]; /* number of current line in the stacked files */
  extern change_line; /* number of current line in change file */
  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 */
  extern reset_input(); /* initialize to read the web file and change file */
  extern get_line(); /* inputs the next line */
  extern check_complete(); /* checks that all changes were picked up */
```

**12.** Code related to section numbers:

```
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  typedef unsigned short sixteen_bits;
  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 */
```

**13.** 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? */
⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡
  extern int argc; /* copy of ac parameter to main */
  extern char **argv; /* copy of av parameter to main */
  extern boolean flags[]; /* an option for each 7-bit code */
```

**14.** Code relating 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)
```

⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡

```
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 */
```

**15.** The procedure that gets everything rolling:

⟨ Common code for CWEAVE and CTANGLE 5 ⟩ +≡

```
extern void common_init();
```

**16. Data structures exclusive to CWEAVE.** As explained in common.w, the field of a **name_info** structure that contains the *rlink* of a section name is used for a completely different purpose in the case of identifiers. It is then called the *ilk* of the identifier, and it is used to distinguish between various types of identifiers, as follows:

*normal* identifiers are part of the C program and will appear in italic type.

*roman* identifiers are index entries that appear after @^ in the CWEB file.

*wildcard* identifiers are index entries that appear after @: in the CWEB file.

*typewriter* identifiers are index entries that appear after @. in the CWEB file.

*else_like*, ..., *typedef_like* identifiers are C reserved words whose *ilk* explains how they are to be treated when C code is being formatted.

```
#define ilk dummy.Ilk
#define normal 0 /* ordinary identifiers have normal ilk */
#define roman 1 /* normal index entries have roman ilk */
#define wildcard 2 /* user-formatted index entries have wildcard ilk */
#define typewriter 3 /* 'typewriter type' entries have typewriter ilk */
#define abnormal(a) (a→ilk > typewriter) /* tells if a name is special */
#define custom 4 /* identifiers with user-given control sequence */
#define unindexed(a) (a→ilk > custom) /* tells if uses of a name are to be indexed */
#define quoted 5 /* NULL */
#define else_like 26 /* else */
#define public_like 40 /* public, private, protected */
#define operator_like 41 /* operator */
#define new_like 42 /* new */
#define catch_like 43 /* catch */
#define for_like 45 /* for, switch, while */
#define do_like 46 /* do */
#define if_like 47 /* if, ifdef, endif, pragma, ... */
#define raw_rpar 48 /* ')' or ']' when looking for const following */
#define raw_unorbin 49 /* '&' or '*' when looking for const following */
#define const_like 50 /* const, volatile */
#define raw_int 51 /* int, char, extern, ... */
#define int_like 52 /* same, when not followed by left parenthesis */
#define case_like 53 /* case, return, goto, break, continue */
#define sizeof_like 54 /* sizeof */
#define struct_like 55 /* struct, union, enum, class */
#define typedef_like 56 /* typedef */
#define define_like 57 /* define */
```

**17.** We keep track of the current section number in *section_count*, which is the total number of sections that have started. Sections which have been altered by a change file entry have their *changed_section* flag turned on during the first phase.

```
⟨ Global variables 17 ⟩ ≡
  boolean change_exists; /* has any section changed? */
```

See also sections 19, 25, 31, 37, 41, 43, 58, 68, 73, 77, 97, 104, 108, 167, 186, 190, 206, 215, 226, 228, 232, 234, and 243

This code is used in section 1

**18.** The other large memory area in CWEAVE keeps the cross-reference data. All uses of the name $p$ are recorded in a linked list beginning at $p\rightarrow xref$, which points into the *xmem* array. The elements of *xmem* are structures consisting of an integer, *num*, and a pointer *xlink* to another element of *xmem*. If $x \leftarrow p\rightarrow xref$ is a pointer into *xmem*, the value of $x\rightarrow num$ is either a section number where $p$ is used, or *cite_flag* plus a section number where $p$ is mentioned, or *def_flag* plus a section number where $p$ is defined; and $x\rightarrow xlink$ points to the next such cross-reference for $p$, if any. This list of cross-references is in decreasing order by section number. The next unused slot in *xmem* is *xref_ptr*. The linked list ends at $\&xmem[0]$.

The global variable *xref_switch* is set either to *def_flag* or to zero, depending on whether the next cross-reference to an identifier is to be underlined or not in the index. This switch is set to *def_flag* when @! or @d is scanned, and it is cleared to zero when the next identifier or index entry cross-reference has been made. Similarly, the global variable *section_xref_switch* is either *def_flag* or *cite_flag* or zero, depending on whether a section name is being defined, cited or used in C text.

```
⟨ Typedef declarations 18 ⟩ ≡
  typedef struct xref_info {
    sixteen_bits num;      /* section number plus zero or def_flag */
    struct xref_info *xlink;      /* pointer to the previous cross-reference */
  } xref_info;
  typedef xref_info *xref_pointer;
```

See also sections 24, 103, and 185

This code is used in section 1

**19.**
```
⟨ Global variables 17 ⟩ +≡
  xref_info xmem[max_refs];      /* contains cross-reference information */
  xref_pointer xmem_end ← xmem + max_refs − 1;
  xref_pointer xref_ptr;      /* the largest occupied position in xmem */
  sixteen_bits xref_switch, section_xref_switch;      /* either zero or def_flag */
```

**20.** A section that is used for multi-file output (with the @( feature) has a special first cross-reference whose *num* field is *file_flag*.

```
#define file_flag (3 * cite_flag)
#define def_flag (2 * cite_flag)
#define cite_flag 10240      /* must be strictly larger than max_sections */
#define xref equiv_or_xref
⟨ Set initial values 20 ⟩ ≡
  xref_ptr ← xmem;
  name_dir→xref ← (char *) xmem;
  xref_switch ← 0;
  section_xref_switch ← 0;
  xmem→num ← 0;      /* sentinel value */
```

See also sections 26, 32, 52, 80, 82, 98, 105, 187, 233, and 235

This code is used in section 3

**21.** 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 (flags['x'] ≡ 0)
#define make_xrefs flags['x']      /* should cross references be output? */
#define is_tiny(p) ((p + 1)→byte_start ≡ (p)→byte_start + 1)
  void new_xref(p)
      name_pointer p;
  {
    xref_pointer q;      /* pointer to previous cross-reference */
    sixteen_bits m, n;      /* new and previous cross-reference value */
    if (no_xref) return;
    if ((unindexed(p) ∨ is_tiny(p)) ∧ xref_switch ≡ 0) return;
    m ← section_count + xref_switch;
    xref_switch ← 0;
    q ← (xref_pointer) p→xref;
    if (q ≠ xmem) {
      n ← q→num;
      if (n ≡ m ∨ n ≡ m + def_flag) return;
      else if (m ≡ n + def_flag) {
        q→num ← m;
        return;
      }
    }
    append_xref(m);
    xref_ptr→xlink ← q;
    p→xref ← (char *) xref_ptr;
  }
```

**22.** The cross-reference lists for section names are slightly different. Suppose that a section name is defined in sections $m_1, \ldots, m_k$, cited in sections $n_1, \ldots, n_l$, and used in sections $p_1, \ldots, p_j$. Then its list will contain $m_1 + def\_flag, \ldots, m_k + def\_flag, n_1 + cite\_flag, \ldots, n_l + cite\_flag, p_1, \ldots, p_j$, in this order.

Although this method of storage take quadratic time on the length of the list, under foreseeable uses of CWEAVE this inefficiency is insignificant.

```
void new_section_xref(p)
    name_pointer p;
{
  xref_pointer q, r;    /* pointers to previous cross-references */
  q ← (xref_pointer) p→xref;
  r ← xmem;
  if (q > xmem)
    while (q→num > section_xref_switch) {
      r ← q;
      q ← q→xlink;
    }
  if (r→num ≡ section_count + section_xref_switch) return;    /* don't duplicate entries */
  append_xref(section_count + section_xref_switch);
  xref_ptr→xlink ← q;
  section_xref_switch ← 0;
  if (r ≡ xmem) p→xref ← (char *) xref_ptr;
  else r→xlink ← xref_ptr;
}
```

**23.** The cross-reference list for a section name may also begin with *file_flag*. Here's how that flag gets put in.

```
void set_file_flag(p)
    name_pointer p;
{
  xref_pointer q;
  q ← (xref_pointer) p→xref;
  if (q→num ≡ file_flag) return;
  append_xref(file_flag);
  xref_ptr→xlink ← q;
  p→xref ← (char *) xref_ptr;
}
```

**24.** A third large area of memory is used for sixteen-bit 'tokens', which appear in short lists similar to the strings of characters in *byte_mem*. Token lists are used to contain the result of C code translated into TEX form; further details about them will be explained later. A *text_pointer* variable is an index into *tok_start*.

```
⟨ Typedef declarations 18 ⟩ +≡
  typedef sixteen_bits token;
  typedef token *token_pointer;
  typedef token_pointer *text_pointer;
```

**25.** 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 $*text\_ptr \equiv tok\_ptr$.

```
⟨ Global variables 17 ⟩ +≡
  token tok_mem[max_toks];      /* tokens */
  token_pointer tok_mem_end ← tok_mem + max_toks − 1;      /* end of tok_mem */
  token_pointer tok_start[max_texts];      /* directory into tok_mem */
  token_pointer tok_ptr;      /* first unused position in tok_mem */
  text_pointer text_ptr;      /* first unused position in tok_start */
  text_pointer tok_start_end ← tok_start + max_texts − 1;      /* end of tok_start */
  token_pointer max_tok_ptr;      /* largest value of tok_ptr */
  text_pointer max_text_ptr;      /* largest value of text_ptr */
```

**26.**

```
⟨ Set initial values 20 ⟩ +≡
  tok_ptr ← tok_mem + 1;
  text_ptr ← tok_start + 1;
  tok_start[0] ← tok_mem + 1;
  tok_start[1] ← tok_mem + 1;
  max_tok_ptr ← tok_mem + 1;
  max_text_ptr ← tok_start + 1;
```

**27.** Here are the three procedures needed to complete *id_lookup*:

```
int names_match(p, first, l, t)
      name_pointer p;      /* points to the proposed match */
      char *first;      /* position of first character of string */
      int l;      /* length of identifier */
      eight_bits t;      /* desired ilk */
{
  if (length(p) ≠ l) return 0;
  if (p→ilk ≠ t ∧ ¬(t ≡ normal ∧ abnormal(p))) return 0;
  return ¬strncmp(first, p→byte_start, l);
}
void init_p(p, t)
      name_pointer p;
      eight_bits t;
{
  p→ilk ← t;
  p→xref ← (char *) xmem;
}
void init_node(p)
      name_pointer p;
{
  p→xref ← (char *) xmem;
}
```

**28.** We have to get C's reserved words into the hash table, and the simplest way to do this is to insert them every time CWEAVE is run. Fortunately there are relatively few reserved words. (Some of these are not strictly "reserved," but are defined in header files of the ISO Standard C Library.)

⟨ Store all the reserved words 28 ⟩ ≡

```
id_lookup("asm", Λ, sizeof_like);
id_lookup("auto", Λ, int_like);
id_lookup("break", Λ, case_like);
id_lookup("case", Λ, case_like);
id_lookup("catch", Λ, catch_like);
id_lookup("char", Λ, raw_int);
id_lookup("class", Λ, struct_like);
id_lookup("clock_t", Λ, raw_int);
id_lookup("const", Λ, const_like);
id_lookup("continue", Λ, case_like);
id_lookup("default", Λ, case_like);
id_lookup("define", Λ, define_like);
id_lookup("defined", Λ, sizeof_like);
id_lookup("delete", Λ, sizeof_like);
id_lookup("div_t", Λ, raw_int);
id_lookup("do", Λ, do_like);
id_lookup("double", Λ, raw_int);
id_lookup("elif", Λ, if_like);
id_lookup("else", Λ, else_like);
id_lookup("endif", Λ, if_like);
id_lookup("enum", Λ, struct_like);
id_lookup("error", Λ, if_like);
id_lookup("extern", Λ, int_like);
id_lookup("FILE", Λ, raw_int);
id_lookup("float", Λ, raw_int);
id_lookup("for", Λ, for_like);
id_lookup("fpos_t", Λ, raw_int);
id_lookup("friend", Λ, int_like);
id_lookup("goto", Λ, case_like);
id_lookup("if", Λ, if_like);
id_lookup("ifdef", Λ, if_like);
id_lookup("ifndef", Λ, if_like);
id_lookup("include", Λ, if_like);
id_lookup("inline", Λ, int_like);
id_lookup("int", Λ, raw_int);
id_lookup("jmp_buf", Λ, raw_int);
id_lookup("ldiv_t", Λ, raw_int);
id_lookup("line", Λ, if_like);
id_lookup("long", Λ, raw_int);
id_lookup("new", Λ, new_like);
id_lookup("NULL", Λ, quoted);
id_lookup("offsetof", Λ, sizeof_like);
id_lookup("operator", Λ, operator_like);
id_lookup("pragma", Λ, if_like);
id_lookup("private", Λ, public_like);
id_lookup("protected", Λ, public_like);
id_lookup("ptrdiff_t", Λ, raw_int);
id_lookup("public", Λ, public_like);
id_lookup("register", Λ, int_like);
id_lookup("return", Λ, case_like);
id_lookup("short", Λ, raw_int);
```

```
id_lookup("sig_atomic_t", Λ, raw_int);
id_lookup("signed", Λ, raw_int);
id_lookup("size_t", Λ, raw_int);
id_lookup("sizeof", Λ, sizeof_like);
id_lookup("static", Λ, int_like);
id_lookup("struct", Λ, struct_like);
id_lookup("switch", Λ, for_like);
id_lookup("template", Λ, int_like);
id_lookup("TeX", Λ, custom);
id_lookup("this", Λ, quoted);
id_lookup("throw", Λ, case_like);
id_lookup("time_t", Λ, raw_int);
id_lookup("try", Λ, else_like);
id_lookup("typedef", Λ, typedef_like);
id_lookup("undef", Λ, if_like);
id_lookup("union", Λ, struct_like);
id_lookup("unsigned", Λ, raw_int);
id_lookup("va_dcl", Λ, decl);      /* Berkeley's variable-arg-list convention */
id_lookup("va_list", Λ, raw_int);      /* ditto */
id_lookup("virtual", Λ, int_like);
id_lookup("void", Λ, raw_int);
id_lookup("volatile", Λ, const_like);
id_lookup("wchar_t", Λ, raw_int);
id_lookup("while", Λ, for_like);
```

This code is used in section 3

**29. Lexical scanning.** Let us now consider the subroutines that read the CWEB source file and break it into meaningful units. There are four such procedures: One simply skips to the next '@␣' or '@*' that begins a section; another passes over the TEX text at the beginning of a section; the third passes over the TEX text in a C comment; and the last, which is the most interesting, gets the next token of a C text. They all use the pointers *limit* and *loc* into the line of input currently being studied.

**30.** Control codes in CWEB, which begin with '@', are converted into a numeric code designed to simplify CWEAVE's logic; for example, larger numbers are given to the control codes that denote more significant milestones, and the code of *new_section* should be the largest of all. Some of these numeric control codes take the place of **char** control codes that will not otherwise appear in the output of the scanning routines.

```
#define ignore °0 /* control code of no interest to CWEAVE */
#define verbatim °2 /* takes the place of extended ASCII α */
#define begin_short_comment °3 /* C++ short comment */
#define begin_comment '\t' /* tab marks will not appear */
#define underline '\n' /* this code will be intercepted without confusion */
#define noop °177 /* takes the place of ASCII delete */
#define xref_roman °203 /* control code for '@^' */
#define xref_wildcard °204 /* control code for '@:' */
#define xref_typewriter °205 /* control code for '@.' */
#define TEX_string °206 /* control code for '@t' */
  format TeX_string TeX
#define ord °207 /* control code for '@'' */
#define join °210 /* control code for '@&' */
#define thin_space °211 /* control code for '@,' */
#define math_break °212 /* control code for '@|' */
#define line_break °213 /* control code for '@/' */
#define big_line_break °214 /* control code for '@#' */
#define no_line_break °215 /* control code for '@+' */
#define pseudo_semi °216 /* control code for '@;' */
#define macro_arg_open °220 /* control code for '@[' */
#define macro_arg_close °221 /* control code for '@]' */
#define trace °222 /* control code for '@0', '@1' and '@2' */
#define translit_code °223 /* control code for '@l' */
#define output_defs_code °224 /* control code for '@h' */
#define format_code °225 /* control code for '@f' and '@s' */
#define definition °226 /* control code for '@d' */
#define begin_C °227 /* control code for '@c' */
#define section_name °230 /* control code for '@<' */
#define new_section °231 /* control code for '@␣' and '@*' */
```

**31.** Control codes are converted to CWEAVE's internal representation by means of the table *ccode*.

```
⟨ Global variables 17 ⟩ +≡
  eight_bits ccode[256]; /* meaning of a char following @ */
```

**32.** ⟨ Set initial values 20 ⟩ +≡

```
{
  int c;
  for (c ← 0; c < 256; c++) ccode[c] ← 0;
}
ccode['␣'] ← ccode['\t'] ← ccode['\n'] ← ccode['\v'] ← ccode['\r'] ← ccode['\f'] ←
    ccode['*'] ← new_section;
ccode['@'] ← '@';     /* 'quoted' at sign */
ccode['='] ← verbatim;
ccode['d'] ← ccode['D'] ← definition;
ccode['f'] ← ccode['F'] ← ccode['s'] ← ccode['S'] ← format_code;
ccode['c'] ← ccode['C'] ← ccode['p'] ← ccode['P'] ← begin_C;
ccode['t'] ← ccode['T'] ← TEX_string;
ccode['l'] ← ccode['L'] ← translit_code;
ccode['q'] ← ccode['Q'] ← noop;
ccode['h'] ← ccode['H'] ← output_defs_code;
ccode['&'] ← join;
ccode['<'] ← ccode['('] ← section_name;
ccode['!'] ← underline;
ccode['^'] ← xref_roman;
ccode[':'] ← xref_wildcard;
ccode['.'] ← xref_typewriter;
ccode[','] ← thin_space;
ccode['|'] ← math_break;
ccode['/'] ← line_break;
ccode['#'] ← big_line_break;
ccode['+'] ← no_line_break;
ccode[';'] ← pseudo_semi;
ccode['['] ← macro_arg_open;
ccode[']'] ← macro_arg_close;
ccode['\''] ← ord;
⟨ Special control codes for debugging 33 ⟩
```

**33.** Users can write `@2`, `@1`, and `@0` to turn tracing fully on, partly on, and off, respectively.

⟨ Special control codes for debugging 33 ⟩ ≡

```
ccode['0'] ← ccode['1'] ← ccode['2'] ← trace;
```

This code is used in section 32

**34.** The *skip_limbo* routine is used on the first pass to skip through portions of the input that are not in any sections, i.e., that precede the first section. After this procedure has been called, the value of *input_has_ended* will tell whether or not a section has actually been found.

There's a complication that we will postpone until later: If the `@s` operation appears in limbo, we want to use it to adjust the default interpretation of identifiers.

⟨ Predeclaration of procedures 2 ⟩ +≡

```
void skip_limbo( );
```

**35.**
```
void skip_limbo( )
{
  while (1) {
    if (loc > limit ∧ get_line( ) ≡ 0) return;
    *(limit + 1) ← '@';
    while (*loc ≠ '@') loc++;      /* look for '@', then skip two chars */
    if (loc++ ≤ limit) {
      int c ← ccode[(eight_bits) *loc++];
      if (c ≡ new_section) return;
      if (c ≡ noop) skip_restricted( );
      else if (c ≡ format_code) ⟨ Process simple format in limbo 71 ⟩;
    }
  }
}
```

**36.** The *skip_TEX* routine is used on the first pass to skip through the TEX code at the beginning of a section. It returns the next control code or '|' found in the input. A *new_section* is assumed to exist at the very end of the file.

```
format skip_TeX TeX
unsigned skip_TEX( )      /* skip past pure TEX code */
{
  while (1) {
    if (loc > limit ∧ get_line( ) ≡ 0) return (new_section);
    *(limit + 1) ← '@';
    while (*loc ≠ '@' ∧ *loc ≠ '|') loc++;
    if (*loc++ ≡ '|') return ('|');
    if (loc ≤ limit) return (ccode[(eight_bits) *(loc++)]);
  }
}
```

**37. Inputting the next token.** As stated above, CWEAVE's most interesting lexical scanning routine is the *get_next* function that inputs the next token of C input. However, *get_next* is not especially complicated.

The result of *get_next* is either a **char** code for some special character, or it is a special code representing a pair of characters (e.g., '!='), or it is the numeric value computed by the *ccode* table, or it is one of the following special codes:

*identifier*: In this case the global variables *id_first* and *id_loc* will have been set to the beginning and ending-plus-one locations in the buffer, as required by the *id_lookup* routine.

*string*: The string will have been copied into the array *section_text*; *id_first* and *id_loc* are set as above (now they are pointers into *section_text*).

*constant*: The constant is copied into *section_text*, with slight modifications; *id_first* and *id_loc* are set.

Furthermore, some of the control codes cause *get_next* to take additional actions:

*xref_roman*, *xref_wildcard*, *xref_typewriter*, *TEX_string*, *verbatim*: The values of *id_first* and *id_loc* will have been set to the beginning and ending-plus-one locations in the buffer.

*section_name*: In this case the global variable *cur_section* will point to the *byte_start* entry for the section name that has just been scanned. The value of *cur_section_char* will be '(' if the section name was preceded by @( instead of @<.

If *get_next* sees '@!' it sets *xref_switch* to *def_flag* and goes on to the next token.

```
#define constant °200 /* C constant */
#define string °201 /* C string */
#define identifier °202 /* C identifier or reserved word */
⟨ Global variables 17 ⟩ +≡
  name_pointer cur_section; /* name of section just scanned */
  char cur_section_char; /* the character just before that name */
```

**38.** ⟨ Include files 6 ⟩ +≡

```
#include <ctype.h> /* definition of isalpha, isdigit and so on */
#include <stdlib.h> /* definition of exit */
```

**39.** As one might expect, *get_next* consists mostly of a big switch that branches to the various special cases that can arise.

```
#define isxalpha(c) ((c) ≡ '_') /* non-alpha character allowed in identifier */
#define ishigh(c) ((eight_bits) (c) > °177)
⟨ Predeclaration of procedures 2 ⟩ +≡
  eight_bits get_next( );
```

**40.**

```
eight_bits get_next( ) /* produces the next input token */
{ eight_bits c; /* the current character */
  while (1) {
    ⟨ Check if we're at the end of a preprocessor command 45 ⟩;
    if (loc > limit ∧ get_line( ) ≡ 0) return (new_section);
    c ← *(loc++);
    if (xisdigit(c) ∨ c ≡ '\\' ∨ c ≡ '.') ⟨ Get a constant 48 ⟩
    else if (c ≡ '\'' ∨ c ≡ '"' ∨ (c ≡ 'L' ∧ (*loc ≡ '\'' ∨ *loc ≡ '"'))
        ∨ (c ≡ '<' ∧ sharp_include_line ≡ 1)) ⟨ Get a string 49 ⟩
    else if (xisalpha(c) ∨ isxalpha(c) ∨ ishigh(c)) ⟨ Get an identifier 47 ⟩
    else if (c ≡ '@') ⟨ Get control code and possible section name 50 ⟩
    else if (xisspace(c)) continue; /* ignore spaces and tabs */
    if (c ≡ '#' ∧ loc ≡ buffer + 1) ⟨ Raise preprocessor flag 42 ⟩;
  mistake: ⟨ Compress two-symbol operator 46 ⟩
    return (c);
  }
}
```

**41.** Because preprocessor commands do not fit in with the rest of the syntax of C, we have to deal with them separately. One solution is to enclose such commands between special markers. Thus, when a `#` is seen as the first character of a line, *get_next* returns a special code *left_preproc* and raises a flag *preprocessing*.

We can use the same internal code number for *left_preproc* as we do for *ord*, since *get_next* changes *ord* into a string.

```
#define left_preproc ord      /* begins a preprocessor command */
#define right_preproc °217    /* ends a preprocessor command */
⟨ Global variables 17 ⟩ +≡
  boolean preprocessing ← 0;      /* are we scanning a preprocessor command? */
```

**42.**
```
⟨ Raise preprocessor flag 42 ⟩ ≡
  {
    preprocessing ← 1;
    ⟨ Check if next token is include 44 ⟩;
    return (left_preproc);
  }
```
This code is used in section 40

**43.** An additional complication is the freakish use of `<` and `>` to delimit a file name in lines that start with `#include`. We must treat this file name as a string.

```
⟨ Global variables 17 ⟩ +≡
  boolean sharp_include_line ← 0;      /* are we scanning a # include line? */
```

**44.**
```
⟨ Check if next token is include 44 ⟩ ≡
  while (loc ≤ buffer_end − 7 ∧ xisspace(∗loc)) loc++;
  if (loc ≤ buffer_end − 6 ∧ strncmp(loc, "include", 7) ≡ 0) sharp_include_line ← 1;
```
This code is used in section 42

**45.** When we get to the end of a preprocessor line, we lower the flag and send a code *right_preproc*, unless the last character was a `\`.

```
⟨ Check if we're at the end of a preprocessor command 45 ⟩ ≡
  while (loc ≡ limit − 1 ∧ preprocessing ∧ ∗loc ≡ '\\')
    if (get_line( ) ≡ 0) return (new_section);      /* still in preprocessor mode */
  if (loc ≥ limit ∧ preprocessing) {
    preprocessing ← sharp_include_line ← 0;
    return (right_preproc);
  }
```
This code is used in section 40

**46.** The following code assigns values to the combinations `++`, `--`, `->`, `>=`, `<=`, `==`, `<<`, `>>`, `!=`, `||`, and `&&`, and to the C++ combinations `...`, `::`, `.*` and `->*`. The compound assignment operators (e.g., `+=`) are treated as separate tokens.

```
#define compress(c) if (loc++ ≤ limit) return (c)
⟨ Compress two-symbol operator 46 ⟩ ≡
  switch (c) {
  case '/':
    if (*loc ≡ '*') {
      compress(begin_comment);
    }
    else if (*loc ≡ '/') compress(begin_short_comment);
    break;
  case '+':
    if (*loc ≡ '+') compress(plus_plus);
    break;
  case '-':
    if (*loc ≡ '-') {
      compress(minus_minus);
    }
    else if (*loc ≡ '>')
      if (*(loc + 1) ≡ '*') {
        loc++;
        compress(minus_gt_ast);
      }
      else compress(minus_gt);
    break;
  case '.':
    if (*loc ≡ '*') {
      compress(period_ast);
    }
    else if (*loc ≡ '.' ∧ *(loc + 1) ≡ '.') {
      loc++;
      compress(dot_dot_dot);
    }
    break;
  case ':':
    if (*loc ≡ ':') compress(colon_colon);
    break;
  case '=':
    if (*loc ≡ '=') compress(eq_eq);
    break;
  case '>':
    if (*loc ≡ '=') {
      compress(gt_eq);
    }
    else if (*loc ≡ '>') compress(gt_gt);
    break;
  case '<':
    if (*loc ≡ '=') {
      compress(lt_eq);
    }
    else if (*loc ≡ '<') compress(lt_lt);
    break;
  case '&':
    if (*loc ≡ '&') compress(and_and);
```

```
    break;
  case '|':
    if (*loc ≡ '|') compress(or_or);
    break;
  case '!':
    if (*loc ≡ '=') compress(not_eq);
    break;
  }
```

This code is used in section 40

**47.** ⟨ Get an identifier 47 ⟩ ≡

```
  {
    id_first ← --loc;
    while (isalpha(*++loc) ∨ isdigit(*loc) ∨ isxalpha(*loc) ∨ ishigh(*loc)) ;
    id_loc ← loc;
    return (identifier);
  }
```

This code is used in section 40

**48.** Different conventions are followed by TEX and C to express octal and hexadecimal numbers; it is reasonable to stick to each convention within its realm. Thus the C part of a CWEB file has octals introduced by `0` and hexadecimals by `0x`, but CWEAVE will print in italics or typewriter font, respectively, and introduced by single or double quotes. In order to simplify the TEX macro used to print such constants, we replace some of the characters.

Notice that in this section and the next, *id_first* and *id_loc* are pointers into the array *section_text*, not into *buffer*.

```
⟨ Get a constant 48 ⟩ ≡
  {
    id_first ← id_loc ← section_text + 1;
    if (*(loc − 1) ≡ '\\') {
      *id_loc++ ← '~';
      while (xisdigit(*loc)) *id_loc++ ← *loc++;
    }     /* octal constant */
    else if (*(loc − 1) ≡ '0') {
      if (*loc ≡ 'x' ∨ *loc ≡ 'X') {
        *id_loc++ ← '^';
        loc++;
        while (xisxdigit(*loc)) *id_loc++ ← *loc++;
      }     /* hex constant */
      else if (xisdigit(*loc)) {
        *id_loc++ ← '~';
        while (xisdigit(*loc)) *id_loc++ ← *loc++;
      }     /* octal constant */
      else goto dec;      /* decimal constant */
    }
    else {      /* decimal constant */
      if (*(loc − 1) ≡ '.' ∧ ¬xisdigit(*loc)) goto mistake;      /* not a constant */
    dec: *id_loc++ ← *(loc − 1);
      while (xisdigit(*loc) ∨ *loc ≡ '.') *id_loc++ ← *loc++;
      if (*loc ≡ 'e' ∨ *loc ≡ 'E') {      /* float constant */
        *id_loc++ ← '_';
        loc++;
        if (*loc ≡ '+' ∨ *loc ≡ '-') *id_loc++ ← *loc++;
        while (xisdigit(*loc)) *id_loc++ ← *loc++;
      }
    }
    while (*loc ≡ 'u' ∨ *loc ≡ 'U' ∨ *loc ≡ 'l' ∨ *loc ≡ 'L' ∨ *loc ≡ 'f' ∨ *loc ≡ 'F') {
      *id_loc++ ← '$';
      *id_loc++ ← toupper(*loc++);
    }
    return (constant);
  }
```

This code is used in section 40

**49.** 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 49 ⟩ ≡
  {
    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') {      /* wide character constant */
      delim ← *loc++;
      *++id_loc ← delim;
    }
    if (delim ≡ '<') delim ← '>';      /* for file names in # include lines */
    while (1) {
      if (loc ≥ limit) {
        if (*(limit − 1) ≠ '\\') {
          err_print("! String didn't end");
          loc ← limit;
          break;
        }
        if (get_line( ) ≡ 0) {
          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) {
      printf("\n! String too long: ");
      term_write(section_text + 1, 25);
      printf("...");
      mark_error;
    }
    id_loc++;
    return (string);
  }
```

This code is used in sections 40 and 50

**50.** 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 50 ⟩ ≡
  {
    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[c];
      skip_restricted( );
      return (c);
    case section_name: ⟨ Scan the section name and make cur_section point to it 51 ⟩;
    case verbatim: ⟨ Scan a verbatim string 57 ⟩;
    case ord: ⟨ Get a string 49 ⟩;
    default: return (ccode[(eight_bits) c]);
    }
  }
```

This code is used in section 40

**51.** The occurrence of a section name sets *xref_switch* to zero, because the section name might (for example) follow **int**.

```
⟨ Scan the section name and make cur_section point to it 51 ⟩ ≡
  {
    char *k;      /* pointer into section_text */
    cur_section_char ← *(loc − 1);
    ⟨ Put section name into section_text 53 ⟩;
    if (k − section_text > 3 ∧ strncmp(k − 2, "...", 3) ≡ 0)
      cur_section ← section_lookup(section_text + 1, k − 3, 1);      /* 1 indicates a prefix */
    else cur_section ← section_lookup(section_text + 1, k, 0);
    xref_switch ← 0;
    return (section_name);
  }
```

This code is used in section 50

**52.** Section names are placed into the *section_text* array with consecutive spaces, tabs, and carriage-returns replaced by single spaces. There will be no spaces at the beginning or the end. (We set *section_text*[0] ← ' ' to facilitate this, since the *section_lookup* routine uses *section_text*[1] as the first character of the name.)

```
⟨ Set initial values 20 ⟩ +≡
  section_text[0] ← ' ';
```

**53.** ⟨ Put section name into *section_text* 53 ⟩ ≡

```
k ← section_text;
while (1) {
  if (loc > limit ∧ get_line( ) ≡ 0) {
    err_print("! Input ended in section name");
    loc ← buffer + 1;
    break;
  }
  c ← ∗loc;
  ⟨ If end of name or erroneous control code, break 54 ⟩;
  loc++;
  if (k < section_text_end) k++;
  if (xisspace(c)) {
    c ← ' ';
    if (∗(k − 1) ≡ ' ') k−−;
  }
  ∗k ← c;
}
if (k ≥ section_text_end) {
  printf("\n! Section name too long: ");
  term_write(section_text + 1, 25);
  printf("...");
  mark_harmless;
}
if (∗k ≡ ' ' ∧ k > section_text) k−−;
```

This code is used in section 51

**54.** ⟨ If end of name or erroneous control code, **break** 54 ⟩ ≡

```
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 53

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

⟨ Predeclaration of procedures 2 ⟩ +≡

```
void skip_restricted( );
```

**56.** **void** *skip_restricted*( )
```
{
  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");
  }
}
```

**57.** 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 57 ⟩ ≡
```
{
  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 50

**58. Phase one processing.** We now have accumulated enough subroutines to make it possible to carry out CWEAVE's first pass over the source file. If everything works right, both phase one and phase two of CWEAVE will assign the same numbers to sections, and these numbers will agree with what CTANGLE does.

The global variable *next_control* often contains the most recent output of *get_next*; in interesting cases, this will be the control code that ended a section or part of a section.

```
⟨ Global variables 17 ⟩ +≡
  eight_bits next_control;      /* control code waiting to be acting upon */
```

**59.** The overall processing strategy in phase one has the following straightforward outline.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void phase_one( );
```

**60.**

```
void phase_one( )
{
  phase ← 1;
  reset_input( );
  section_count ← 0;
  skip_limbo( );
  change_exists ← 0;
  while (¬input_has_ended) ⟨ Store cross-reference data for the current section 61 ⟩;
  changed_section[section_count] ← change_exists;      /* the index changes if anything does */
  phase ← 2;      /* prepare for second phase */
  ⟨ Print error messages about unused or undefined section names 76 ⟩;
}
```

**61.**

```
⟨ Store cross-reference data for the current section 61 ⟩ ≡
{
  if (++section_count ≡ max_sections) overflow("section␣number");
  changed_section[section_count] ← changing;      /* it will become 1 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 66 ⟩;
  ⟨ Store cross-references in the definition part of a section 69 ⟩;
  ⟨ Store cross-references in the C part of a section 72 ⟩;
  if (changed_section[section_count]) change_exists ← 1;
}
```

This code is used in section 60

**62.** The *C_xref* subroutine stores references to identifiers in C text material beginning with the current value of *next_control* and continuing until *next_control* is '{' or '|', or until the next "milestone" is passed (i.e., *next_control* $\geq$ *format_code*). If *next_control* $\geq$ *format_code* when *C_xref* is called, nothing will happen; but if *next_control* $\equiv$ '|' upon entry, the procedure assumes that this is the '|' preceding C text that is to be processed.

The parameter *spec_ctrl* is used to change this behavior. In most cases *C_xref* is called with *spec_ctrl* $\equiv$ *ignore*, which triggers the default processing described above. If *spec_ctrl* $\equiv$ *section_name*, section names will be gobbled. This is used when C text in the TEX part or inside comments is parsed: It allows for section names to appear in | ... |, but these strings will not be entered into the cross reference lists since they are not definitions of section names.

The program uses the fact that our internal code numbers satisfy the relations *xref_roman* $\equiv$ *identifier* + *roman* and *xref_wildcard* $\equiv$ *identifier* + *wildcard* and *xref_typewriter* $\equiv$ *identifier* + *typewriter* and *normal* $\equiv$ 0.

⟨ Predeclaration of procedures 2 ⟩ +≡
```
void C_xref();
```

**63.**
```
void C_xref(spec_ctrl)      /* makes cross-references for C identifiers */
  eight_bits spec_ctrl;
{
  name_pointer p;      /* a referenced name */
  while (next_control < format_code ∨ next_control ≡ spec_ctrl) {
    if (next_control ≥ identifier ∧ next_control ≤ xref_typewriter) {
      if (next_control > identifier) ⟨ Replace "@@" by "@" 67 ⟩
      p ← id_lookup(id_first, id_loc, next_control − identifier);
      new_xref(p);
    }
    if (next_control ≡ section_name) {
      section_xref_switch ← cite_flag;
      new_section_xref(cur_section);
    }
    next_control ← get_next();
    if (next_control ≡ '|' ∨ next_control ≡ begin_comment ∨ next_control ≡ begin_short_comment)
      return;
  }
}
```

**64.** The *outer_xref* subroutine is like *C_xref* except that it begins with *next_control* $\neq$ '|' and ends with *next_control* $\geq$ *format_code*. Thus, it handles C text with embedded comments.

⟨ Predeclaration of procedures 2 ⟩ +≡
```
void outer_xref();
```

**65.** **void** *outer_xref*( ) /* extension of *C_xref* */
```
{
  int bal; /* brace level in comment */
  while (next_control < format_code)
    if (next_control ≠ begin_comment ∧ next_control ≠ begin_short_comment) C_xref(ignore);
    else {
      boolean is_long_comment ← (next_control ≡ begin_comment);
      bal ← copy_comment(is_long_comment, 1);
      next_control ← '|';
      while (bal > 0) {
        C_xref(section_name); /* do not reference section names in comments */
        if (next_control ≡ '|') bal ← copy_comment(is_long_comment, bal);
        else bal ← 0; /* an error message will occur in phase two */
      }
    }
}
```

**66.** 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 66 ⟩ ≡
```
while (1) {
  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 "@" 67 ⟩
      new_xref(id_lookup(id_first, id_loc, next_control − identifier));
    }
    break;
  }
  if (next_control ≥ format_code) break;
}
```
This code is used in section 61

**67.** ⟨ Replace "@@" by "@" 67 ⟩ ≡
```
{
  char *src ← id_first, *dst ← id_first;
  while (src < id_loc) {
    if (*src ≡ '@') src++;
    *dst++ ← *src++;
  }
  id_loc ← dst;
  while (dst < src) *dst++ ← ' '; /* clean up in case of error message display */
}
```
This code is used in sections 63 and 66

**68.** During the definition and C parts of a section, cross-references are made for all identifiers except reserved words. However, the right identifier in a format definition is not referenced, and the left identifier is referenced only if it has been explicitly underlined (preceded by `@!`). The TEX code in comments is, of course, ignored, except for C portions enclosed in | ... |; the text of a section name is skipped entirely, even if it contains | ... | constructions.

The variables *lhs* and *rhs* point to the respective identifiers involved in a format definition.

```
⟨ Global variables 17 ⟩ +≡
  name_pointer lhs, rhs;      /* pointers to byte_start for format identifiers */
```

**69.** When we get to the following code we have *next_control* ≥ *format_code*.

```
⟨ Store cross-references in the definition part of a section 69 ⟩ ≡
  while (next_control ≤ definition) {      /* format_code or definition */
    if (next_control ≡ definition) {
      xref_switch ← def_flag;      /* implied @! */
      next_control ← get_next( );
    }
    else ⟨ Process a format definition 70 ⟩;
    outer_xref( );
  }
```

This code is used in section 61

**70.** Error messages for improper format definitions will be issued in phase two. Our job in phase one is to define the *ilk* of a properly formatted identifier, and to remove cross-references to identifiers that we now discover should be unindexed.

```
⟨ Process a format definition 70 ⟩ ≡
  {
    next_control ← get_next( );
    if (next_control ≡ identifier) {
      lhs ← id_lookup(id_first, id_loc, normal);
      lhs→ilk ← normal;
      if (xref_switch) new_xref(lhs);
      next_control ← get_next( );
      if (next_control ≡ identifier) {
        rhs ← id_lookup(id_first, id_loc, normal);
        lhs→ilk ← rhs→ilk;
        if (unindexed(lhs)) {      /* retain only underlined entries */
          xref_pointer q, r ← Λ;
          for (q ← (xref_pointer) lhs→xref; q > xmem; q ← q→xlink)
            if (q→num < def_flag)
              if (r) r→xlink ← q→xlink;
              else lhs→xref ← (char *) q→xlink;
            else r ← q;
        }
        next_control ← get_next( );
      }
    }
  }
```

This code is used in section 69

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

```
⟨ Process simple format in limbo 71 ⟩ ≡
  {
    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 35

**72.** Finally, when the TEX and definition parts have been treated, we have *next_control* ≥ *begin_C*.

```
⟨ Store cross-references in the C part of a section 72 ⟩ ≡
  if (next_control ≤ section_name) {      /* begin_C or section_name */
    if (next_control ≡ begin_C) section_xref_switch ← 0;
    else {
      section_xref_switch ← def_flag;
      if (cur_section_char ≡ '(' ∧ cur_section ≠ name_dir) set_file_flag(cur_section);
    }
    do {
      if (next_control ≡ section_name ∧ cur_section ≠ name_dir) new_section_xref(cur_section);
      next_control ← get_next( );
      outer_xref( );
    } while (next_control ≤ section_name);
  }
```

This code is used in section 61

**73.** After phase one has looked at everything, we want to check that each section name was both defined and used. The variable *cur_xref* will point to cross-references for the current section name of interest.

```
⟨ Global variables 17 ⟩ +≡
  xref_pointer cur_xref;      /* temporary cross-reference pointer */
  boolean an_output;      /* did file_flag precede cur_xref? */
```

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

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void section_check( );
```

**75.** **void** *section_check*(*p*)
**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* ← 1;
*cur_xref* ← *cur_xref*→*xlink*;
}
**else** *an_output* ← 0;
**if** (*cur_xref*→*num* < *def_flag*) {
*printf*("\n!␣Never␣defined:␣<");
*print_section_name*(*p*);
*putchar*('>');
*mark_harmless*;
}
**while** (*cur_xref*→*num* ≥ *cite_flag*) *cur_xref* ← *cur_xref*→*xlink*;
**if** (*cur_xref* ≡ *xmem* ∧ ¬*an_output*) {
*printf*("\n!␣Never␣used:␣<");
*print_section_name*(*p*);
*putchar*('>');
*mark_harmless*;
}
*section_check*(*p*→*rlink*);
}
}

**76.** ⟨ Print error messages about unused or undefined section names 76 ⟩ ≡
*section_check*(*root*)

This code is used in section 60

**77. Low-level output routines.** The TEX output is supposed to appear in lines at most *line_length* characters long, so we place it into an output buffer. During the output process, *out_line* will hold the current line number of the line about to be output.

```
⟨ Global variables 17 ⟩ +≡
  char out_buf[line_length + 1];      /* assembled characters */
  char *out_ptr;      /* just after last character in out_buf */
  char *out_buf_end ← out_buf + line_length;      /* end of out_buf */
  int out_line;      /* number of next line to be output */
```

**78.** The *flush_buffer* routine empties the buffer up to a given breakpoint, and moves any remaining characters to the beginning of the next line. If the *per_cent* parameter is 1 a `'%'` is appended to the line that is being output; in this case the breakpoint $b$ should be strictly less than *out_buf_end*. If the *per_cent* parameter is 0, trailing blanks are suppressed. The characters emptied from the buffer form a new line of output; if the *carryover* parameter is true, a `"%"` in that line will be carried over to the next line (so that TEX will ignore the completion of commented-out text).

```
#define c_line_write(c) fflush(active_file), fwrite(out_buf + 1, sizeof(char), c, active_file)
#define tex_putc(c) putc(c, active_file)
#define tex_new_line putc('\n', active_file)
#define tex_printf(c) fprintf(active_file, c)
  void flush_buffer(b, per_cent, carryover)
      char *b;      /* outputs from out_buf + 1 to b,where b ≤ out_ptr */
      boolean per_cent, carryover;
  {
    char *j;
    j ← b;      /* pointer into out_buf */
    if (¬per_cent)      /* remove trailing blanks */
      while (j > out_buf ∧ *j ≡ '␣') j--;
    c_line_write(j − out_buf);
    if (per_cent) tex_putc('%');
    tex_new_line;
    out_line++;
    if (carryover)
      while (j > out_buf)
        if (*j-- ≡ '%' ∧ (j ≡ out_buf ∨ *j ≠ '\\')) {
          *b-- ← '%';
          break;
        }
    if (b < out_ptr) strncpy(out_buf + 1, b + 1, out_ptr − b);
    out_ptr −= b − out_buf;
  }
```

**79.** When we are copying TEX source material, we retain line breaks that occur in the input, except that an empty line is not output when the TEX source line was nonempty. For example, a line of the TEX file that contains only an index cross-reference entry will not be copied. The *finish_line* routine is called just before *get_line* inputs a new line, and just after a line break token has been emitted during the output of translated C text.

```
void finish_line( )      /* do this at the end of a line */
{
  char *k;      /* pointer into buffer */
  if (out_ptr > out_buf) flush_buffer(out_ptr, 0, 0);
  else {
    for (k ← buffer; k ≤ limit; k++)
      if (¬(xisspace(*k))) return;
    flush_buffer(out_buf, 0, 0);
  }
}
```

**80.** 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 '`\input cwebmac`'.

```
⟨ Set initial values 20 ⟩ +≡
  out_ptr ← out_buf + 1;
  out_line ← 1;
  active_file ← tex_file;
  *out_ptr ← 'c';
  tex_printf("\\input␣cwebma");
```

**81.** When we wish to append one character $c$ to the output buffer, we write '$out(c)$'; this will cause the buffer to be emptied if it was already full. If we want to append more than one character at once, we say $out\_str(s)$, where $s$ is a string containing the characters.

A line break will occur at a space or after a single-nonletter TEX control sequence.

```
#define out(c)
        {
          if (out_ptr ≥ out_buf_end) break_out( );
          *(++out_ptr) ← c;
        }
void out_str(s)      /* output characters from s to end of string */
    char *s;
{
  while (*s) out(*s++);
}
```

**82.** The *break_out* routine is called just before the output buffer is about to overflow. To make this routine a little faster, we initialize position 0 of the output buffer to '\'; this character isn't really output.

```
⟨ Set initial values 20 ⟩ +≡
  out_buf[0] ← '\\';
```

**83.** A long line is broken at a blank space or just before a backslash that isn't preceded by another backslash. In the latter case, a `'%'` is output at the break.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void break_out( );
```

**84.** **void** *break_out*( ) /* finds a way to break the output line */
```
{
  char *k ← out_ptr;      /* pointer into out_buf */
  while (1) {
    if (k ≡ out_buf) ⟨ Print warning message, break the line, return 85 ⟩;
    if (*k ≡ '␣') {
      flush_buffer(k, 0, 1);
      return;
    }
    if (*(k−−) ≡ '\\' ∧ *k ≠ '\\') {      /* we've decreased k */
      flush_buffer(k, 1, 1);
      return;
    }
  }
}
```

**85.** 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** 85 ⟩ ≡
```
{
  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, 1, 1);
  return;
}
```
This code is used in section 84

**86.** Here is a macro that outputs a section number in decimal notation. The number to be converted by *out_section* is known to be less than *def_flag*, so it cannot have more than five decimal digits. If the section is changed, we output '`\*`' just after the number.
```
void out_section(n)
    sixteen_bits n;
{
  char s[6];
  sprintf(s, "%d", n);
  out_str(s);
  if (changed_section[n]) out_str("\\*");
}
```

**87.** The *out_name* procedure is used to output an identifier or index entry, enclosing it in braces.
```
void out_name(p)
    name_pointer p;
{
  char *k, *k_end ← (p + 1)→byte_start;      /* pointers into byte_mem */
  out('{');
  for (k ← p→byte_start; k < k_end; k++) {
    if (isxalpha(*k)) out('\\');
    out(*k);
  }
  out('}');
}
```

**88. Routines that copy TEX material.** During phase two, we use subroutines *copy_limbo*, *copy_TEX*, and *copy_comment* in place of the analogous *skip_limbo*, *skip_TEX*, and *skip_comment* that were used in phase one. (Well, *copy_comment* was actually written in such a way that it functions as *skip_comment* in phase one.)

The *copy_limbo* routine, for example, takes TEX material that is not part of any section and transcribes it almost verbatim to the output file. The use of '@' signs is severely restricted in such material: '@@' pairs are replaced by singletons; '@l' and '@q' and '@s' are interpreted.

```
void copy_limbo( )
{
  char c;
  while (1) {
    if (loc > limit ∧ (finish_line( ), get_line( ) ≡ 0)) 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('@');
      }
    }
  }
}
```

**89.** The *copy_TEX* routine processes the TEX code at the beginning of a section; for example, the words you are now reading were copied in this way. It returns the next control code or '|' found in the input. We don't copy spaces or tab marks into the beginning of a line. This makes the test for empty lines in *finish_line* work.

**90. format** *copy_TeX* *TeX*

```
eight_bits copy_TEX( )
{
  char c;      /* current character being copied */
  while (1) {
    if (loc > limit ∧ (finish_line( ), get_line( ) ≡ 0)) return (new_section);
    *(limit + 1) ← '@';
    while ((c ← *(loc++)) ≠ '|' ∧ c ≠ '@') {
      out(c);
      if (out_ptr ≡ out_buf + 1 ∧ (xisspace(c))) out_ptr−−;
    }
    if (c ≡ '|') return ('|');
    if (loc ≤ limit) return (ccode[(eight_bits) *(loc++)]);
  }
}
```

**91.** 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;
        }
⟨ Predeclaration of procedures 2 ⟩ +≡
  int copy_comment( );
```

**92.**

```
int copy_comment(is_long_comment, bal)      /* 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 (1) {
    if (loc > limit) {
      if (is_long_comment) {
        if (get_line( ) ≡ 0) {
          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 93 ⟩;
    if (phase ≡ 2) {
      if (ishigh(c)) app_tok(quoted_char);
      app_tok(c);
    }
    ⟨ Copy special things when c ≡ '@', '\\' 94 ⟩;
    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 95 ⟩;
}
```

**93.** ⟨ Check for end of comment 93 ⟩ ≡

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

This code is used in section 92

**94.** ⟨ Copy special things when $c \equiv$ '@', '\\' 94 ⟩ ≡

```
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 92

**95.** We output enough right braces to keep TEX happy.

⟨ Clear *bal* and **return** 95 ⟩ ≡

```
if (phase ≡ 2)
  while (bal-- > 0) app_tok('}');
return (0);
```

This code is used in section 92

**96. Parsing.** The most intricate part of CWEAVE is its mechanism for converting C-like code into TEX code, and we might as well plunge into this aspect of the program now. A "bottom up" approach is used to parse the C-like material, since CWEAVE must deal with fragmentary constructions whose overall "part of speech" is not known.

At the lowest level, the input is represented as a sequence of entities that we shall call *scraps*, where each scrap of information consists of two parts, its *category* and its *translation*. The category is essentially a syntactic class, and the translation is a token list that represents TEX code. Rules of syntax and semantics tell us how to combine adjacent scraps into larger ones, and if we are lucky an entire C text that starts out as hundreds of small scraps will join together into one gigantic scrap whose translation is the desired TEX code. If we are unlucky, we will be left with several scraps that don't combine; their translations will simply be output, one by one.

The combination rules are given as context-sensitive productions that are applied from left to right. Suppose that we are currently working on the sequence of scraps $s_1\, s_2 \ldots s_n$. We try first to find the longest production that applies to an initial substring $s_1\, s_2 \ldots$; but if no such productions exist, we find to find the longest production applicable to the next substring $s_2\, s_3 \ldots$; and if that fails, we try to match $s_3\, s_4 \ldots$, etc.

A production applies if the category codes have a given pattern. For example, one of the productions (see rule 3) is

$$exp \left\{ \begin{array}{c} binop \\ unorbinop \end{array} \right\} exp \;\rightarrow\; exp$$

and it means that three consecutive scraps whose respective categories are *exp*, *binop* (or *unorbinop*), and *exp* are converted to one scrap whose category is *exp*. The translations of the original scraps are simply concatenated. The case of

$$exp\ comma\ exp \rightarrow exp \qquad\qquad E_1 C\ opt9\ E_2$$

(rule 4) is only slightly more complicated: Here the resulting *exp* translation consists not only of the three original translations, but also of the tokens *opt* and 9 between the translations of the *comma* and the following *exp*. In the TEX file, this will specify an optional line break after the comma, with penalty 90.

At each opportunity the longest possible production is applied. For example, if the current sequence of scraps is *int_like cast lbrace*, rule 31 is applied; but if the sequence is *int_like cast* followed by anything other than *lbrace*, rule 32 takes effect.

Translation rules such as '$E_1 C\ opt9\ E_2$' above use subscripts to distinguish between translations of scraps whose categories have the same initial letter; these subscripts are assigned from left to right.

**97.** Here is a list of the category codes that scraps can have. (A few others, like *int_like*, have already been defined; the *cat_name* array contains a complete list.)

```
#define exp 1 /* denotes an expression, including perhaps a single identifier */
#define unop 2 /* denotes a unary operator */
#define binop 3 /* denotes a binary operator */
#define unorbinop 4 /* denotes an operator that can be unary or binary, depending on context */
#define cast 5 /* denotes a cast */
#define question 6 /* denotes a question mark and possibly the expressions flanking it */
#define lbrace 7 /* denotes a left brace */
#define rbrace 8 /* denotes a right brace */
#define decl_head 9 /* denotes an incomplete declaration */
#define comma 10 /* denotes a comma */
#define lpar 11 /* denotes a left parenthesis or left bracket */
#define rpar 12 /* denotes a right parenthesis or right bracket */
#define prelangle 13 /* denotes '<' before we know what it is */
#define prerangle 14 /* denotes '>' before we know what it is */
#define langle 15 /* denotes '<' when it's used as angle bracket in a template */
#define colcol 18 /* denotes '::' */
#define base 19 /* denotes a colon that introduces a base specifier */
#define decl 20 /* denotes a complete declaration */
#define struct_head 21 /* denotes the beginning of a structure specifier */
#define stmt 23 /* denotes a complete statement */
#define function 24 /* denotes a complete function */
#define fn_decl 25 /* denotes a function declarator */
#define semi 27 /* denotes a semicolon */
#define colon 28 /* denotes a colon */
#define tag 29 /* denotes a statement label */
#define if_head 30 /* denotes the beginning of a compound conditional */
#define else_head 31 /* denotes a prefix for a compound statement */
#define if_clause 32 /* pending if together with a condition */
#define lproc 35 /* begins a preprocessor command */
#define rproc 36 /* ends a preprocessor command */
#define insert 37 /* a scrap that gets combined with its neighbor */
#define section_scrap 38 /* section name */
#define dead 39 /* scrap that won't combine */
#define begin_arg 58 /* @[ */
#define end_arg 59 /* @] */
```

⟨ Global variables 17 ⟩ +≡

```
char cat_name[256][12];
eight_bits cat_index;
```

**98.** ⟨ Set initial values 20 ⟩ +≡
**for** (*cat_index* ← 0; *cat_index* < 255; *cat_index*++) *strcpy*(*cat_name*[*cat_index*], "UNKNOWN");
*strcpy*(*cat_name*[*exp*], "exp");
*strcpy*(*cat_name*[*unop*], "unop");
*strcpy*(*cat_name*[*binop*], "binop");
*strcpy*(*cat_name*[*unorbinop*], "unorbinop");
*strcpy*(*cat_name*[*cast*], "cast");
*strcpy*(*cat_name*[*question*], "?");
*strcpy*(*cat_name*[*lbrace*], "{");
*strcpy*(*cat_name*[*rbrace*], "}");
*strcpy*(*cat_name*[*decl_head*], "decl_head");
*strcpy*(*cat_name*[*comma*], ",");
*strcpy*(*cat_name*[*lpar*], "(");
*strcpy*(*cat_name*[*rpar*], ")");
*strcpy*(*cat_name*[*prelangle*], "<");
*strcpy*(*cat_name*[*prerangle*], ">");
*strcpy*(*cat_name*[*langle*], "\\<");
*strcpy*(*cat_name*[*colcol*], "::");
*strcpy*(*cat_name*[*base*], "\\:");
*strcpy*(*cat_name*[*decl*], "decl");
*strcpy*(*cat_name*[*struct_head*], "struct_head");
*strcpy*(*cat_name*[*stmt*], "stmt");
*strcpy*(*cat_name*[*function*], "function");
*strcpy*(*cat_name*[*fn_decl*], "fn_decl");
*strcpy*(*cat_name*[*else_like*], "else_like");
*strcpy*(*cat_name*[*semi*], ";");
*strcpy*(*cat_name*[*colon*], ":");
*strcpy*(*cat_name*[*tag*], "tag");
*strcpy*(*cat_name*[*if_head*], "if_head");
*strcpy*(*cat_name*[*else_head*], "else_head");
*strcpy*(*cat_name*[*if_clause*], "if()");
*strcpy*(*cat_name*[*lproc*], "#{");
*strcpy*(*cat_name*[*rproc*], "#}");
*strcpy*(*cat_name*[*insert*], "insert");
*strcpy*(*cat_name*[*section_scrap*], "section");
*strcpy*(*cat_name*[*dead*], "@d");
*strcpy*(*cat_name*[*public_like*], "public");
*strcpy*(*cat_name*[*operator_like*], "operator");
*strcpy*(*cat_name*[*new_like*], "new");
*strcpy*(*cat_name*[*catch_like*], "catch");
*strcpy*(*cat_name*[*for_like*], "for");
*strcpy*(*cat_name*[*do_like*], "do");
*strcpy*(*cat_name*[*if_like*], "if");
*strcpy*(*cat_name*[*raw_rpar*], ")?");
*strcpy*(*cat_name*[*raw_unorbin*], "unorbinop?");
*strcpy*(*cat_name*[*const_like*], "const");
*strcpy*(*cat_name*[*raw_int*], "raw");
*strcpy*(*cat_name*[*int_like*], "int");
*strcpy*(*cat_name*[*case_like*], "case");
*strcpy*(*cat_name*[*sizeof_like*], "sizeof");
*strcpy*(*cat_name*[*struct_like*], "struct");
*strcpy*(*cat_name*[*typedef_like*], "typedef");
*strcpy*(*cat_name*[*define_like*], "define");
*strcpy*(*cat_name*[*begin_arg*], "@[");
*strcpy*(*cat_name*[*end_arg*], "@]");

*strcpy*(*cat_name*[0], "zero");

**99.** This code allows CWEAVE to display its parsing steps.

```
void print_cat(c)      /* symbolic printout of a category */
    eight_bits c;
{
  printf(cat_name[c]);
}
```

**100.** The token lists for translated TEX output contain some special control symbols as well as ordinary characters. These control symbols are interpreted by CWEAVE before they are written to the output file.

*break_space* denotes an optional line break or an en space;

*force* denotes a line break;

*big_force* denotes a line break with additional vertical space;

*preproc_line* denotes that the line will be printed flush left;

*opt* denotes an optional line break (with the continuation line indented two ems with respect to the normal starting position)—this code is followed by an integer $n$, and the break will occur with penalty $10n$;

*backup* denotes a backspace of one em;

*cancel* obliterates any *break_space*, *opt*, *force*, or *big_force* tokens that immediately precede or follow it and also cancels any *backup* tokens that follow it;

*indent* causes future lines to be indented one more em;

*outdent* causes future lines to be indented one less em.

All of these tokens are removed from the TEX output that comes from C text between | ... | signs; *break_space* and *force* and *big_force* become single spaces in this mode. The translation of other C texts results in TEX control sequences \1, \2, \3, \4, \5, \6, \7, \8 corresponding respectively to *indent*, *outdent*, *opt*, *backup*, *break_space*, *force*, *big_force* and *preproc_line*. However, a sequence of consecutive '␣', *break_space*, *force*, and/or *big_force* tokens is first replaced by a single token (the maximum of the given ones).

The token *math_rel* will be translated into \MRL{, and it will get a matching } later. Other control sequences in the TEX output will be '\\{...}' surrounding identifiers, '\&{...}' surrounding reserved words, '\.{...}' surrounding strings, '\C{...} *force*' surrounding comments, and '\X$n$:...\X' surrounding section names, where $n$ is the section number.

```
#define math_rel °206
#define big_cancel °210      /* like cancel, also overrides spaces */
#define cancel °211      /* overrides backup, break_space, force, big_force */
#define indent °212      /* one more tab (\1) */
#define outdent °213      /* one less tab (\2) */
#define opt °214      /* optional break in mid-statement (\3) */
#define backup °215      /* stick out one unit to the left (\4) */
#define break_space °216      /* optional break between statements (\5) */
#define force °217      /* forced break between statements (\6) */
#define big_force °220      /* forced break with additional space (\7) */
#define preproc_line °221      /* begin line without indentation (\8) */
#define quoted_char °222      /* introduces a character token in the range °200–°377 */
#define end_translation °223      /* special sentinel token at end of list */
#define inserted °224      /* sentinel to mark translations of inserts */
```

**101.** 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 |
| ++ | *binop*: \PP | yes |
| -- | *binop*: \MM | yes |
| -> | *binop*: \MG | yes |
| >> | *binop*: \GG | yes |
| << | *binop*: \LL | yes |
| :: | *colcol*: \DC | maybe |
| .* | *binop*: \PA | yes |
| ->* | *binop*: \MGA | yes |
| ... | *exp*: \,\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 |
| 77 | *exp*: \T{77} | maybe |
| 77L | *exp*: \T{77\$L} | maybe |
| 0.1E5 | *exp*: \T{0.1_5} | maybe |
| + | *unorbinop*: + | yes |
| - | *unorbinop*: - | yes |
| * | *raw_unorbin*: * | yes |
| / | *binop*: / | yes |
| < | *binop*: < | yes |
| = | *binop*: \K | yes |
| > | *binop*: > | yes |
| . | *binop*: . | yes |
| \| | *binop*: \OR | yes |
| ^ | *binop*: \XOR | yes |
| % | *binop*: \MOD | yes |
| ? | *question*: \? | yes |
| ! | *unop*: \R | yes |
| ~ | *unop*: \CM | yes |
| & | *raw_unorbin*: \AND | yes |
| ( | *lpar*: ( | maybe |
| [ | *lpar*: [ | maybe |
| ) | *raw_rpar*: ) | maybe |
| ] | *raw_rpar*: ] | maybe |
| { | *lbrace*: { | yes |
| } | *lbrace*: } | yes |
| , | *comma*: , | yes |
| ; | *semi*: ; | maybe |
| : | *colon*: : | maybe |

| | | |
|---|---|---|
| # (within line) | *unorbinop*: \# | yes |
| # (at beginning) | *lproc*: *force preproc_line* \# | no |
| end of # line | *rproc*: *force* | no |
| identifier | *exp*: \\{identifier with underlines quoted} | maybe |
| asm | *sizeof_like*: ** | maybe |
| auto | *int_like*: ** | maybe |
| break | *case_like*: ** | maybe |
| case | *case_like*: ** | maybe |
| catch | *catch_like*: ** | maybe |
| char | *raw_int*: ** | maybe |
| class | *struct_like*: ** | maybe |
| clock_t | *raw_int*: ** | maybe |
| const | *const_like*: ** | maybe |
| continue | *case_like*: ** | maybe |
| default | *case_like*: ** | maybe |
| define | *define_like*: ** | maybe |
| defined | *sizeof_like*: ** | maybe |
| delete | *sizeof_like*: ** | maybe |
| div_t | *raw_int*: ** | maybe |
| do | *do_like*: ** | maybe |
| double | *raw_int*: ** | maybe |
| elif | *if_like*: ** | maybe |
| else | *else_like*: ** | maybe |
| endif | *if_like*: ** | maybe |
| enum | *struct_like*: ** | maybe |
| error | *if_like*: ** | maybe |
| extern | *int_like*: ** | maybe |
| FILE | *raw_int*: ** | 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 |
| 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 |
| new | *new_like*: ** | maybe |
| NULL | *exp*: \NULL | yes |
| offsetof | *sizeof_like*: ** | maybe |
| operator | *operator_like*: ** | maybe |
| 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 |
| 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 |
| `struct` | *struct_like*: ** | maybe |
| `switch` | *if_like*: ** | maybe |
| `template` | *int_like*: ** | maybe |
| `TeX` | *exp*: \TeX | yes |
| `this` | *exp*: \this | yes |
| `throw` | *case_like*: ** | maybe |
| `time_t` | *raw_int*: ** | maybe |
| `try` | *else_like*: ** | maybe |
| `typedef` | *typedef_like*: ** | maybe |
| `undef` | *if_like*: ** | maybe |
| `union` | *struct_like*: ** | maybe |
| `unsigned` | *raw_int*: ** | 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` | *if_like*: ** | maybe |
| `@,` | *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.

**102.** 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* and *bsp* as shorthands for *indent*, *outdent*, *backup* and *break_space*, respectively.

| | LHS | → RHS | Translation | Example |
|---|---|---|---|---|
| 0 | $\left\{\begin{matrix} any \\ any\ any \\ any\ any\ any \end{matrix}\right\}$ *insert* | $\rightarrow \left\{\begin{matrix} any \\ any\ any \\ any\ any\ any \end{matrix}\right\}$ | | stmt; /∗ comment ∗/ |
| 1 | *exp* $\left\{\begin{matrix} lbrace \\ int\_like \\ decl \end{matrix}\right\}$ | → *fn_decl* $\left\{\begin{matrix} lbrace \\ int\_like \\ decl \end{matrix}\right\}$ | $F = E^*\ in\ in$ | *main*(){<br>*main*(*ac*, *av*) **int** *ac*; |
| 2 | *exp unop* | → *exp* | | $x{+}{+}$ |
| 3 | *exp* $\left\{\begin{matrix} binop \\ unorbinop \end{matrix}\right\}$ *exp* | → *exp* | | $x/y$<br>$x + y$ |
| 4 | *exp comma exp* | → *exp* | $EC\ opt9\ E$ | $f(x, y)$ |
| 5 | *exp* $\left\{\begin{matrix} exp \\ cast \end{matrix}\right\}$ | → *exp* | | *time*( ) |
| 6 | *exp semi* | → *stmt* | | $x \leftarrow 0$; |
| 7 | *exp colon* | → *tag* | $E^*C$ | *found*: |
| 8 | *exp base int_like comma* | → *base* | $B_\sqcup IC\ opt9$ | **D** : **C**, |
| 9 | *exp base int_like lbrace* | → *exp lbrace* | $E = E_\sqcup B_\sqcup I$ | **D** : **C** { |
| 10 | *exp rbrace* | → *stmt rbrace* | | end of **enum** list |
| 11 | *lpar* $\left\{\begin{matrix} exp \\ unorbinop \end{matrix}\right\}$ *rpar* | → *exp* | | $(x)$<br>$(*)$ |
| 12 | *lpar rpar* | → *exp* | $L \backslash , R$ | functions, declarations |
| 13 | *lpar* $\left\{\begin{matrix} decl\_head \\ int\_like \end{matrix}\right\}$ *rpar* | → *cast* | | (**char** ∗) |
| 14 | *lpar* $\left\{\begin{matrix} decl\_head \\ int\_like \\ exp \end{matrix}\right\}$ *comma* | → *lpar* | $\mathrm{L}\left\{\begin{matrix} D \\ I \\ E \end{matrix}\right\}\mathrm{C}\ opt9$ | (**int**, |
| 15 | *lpar* $\left\{\begin{matrix} stmt \\ decl \end{matrix}\right\}$ | → *lpar* | $\left\{\begin{matrix} LS_\sqcup \\ LD_\sqcup \end{matrix}\right\}$ | $\left\{\begin{matrix} (k \leftarrow 5; \\ (\mathbf{int}\ k \leftarrow 5; \end{matrix}\right\}$ |
| 16 | *question exp colon* | → *binop* | | ? $x$ : |
| 17 | *unop* $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ | → $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ | | $\neg x$<br>∼**C** |
| 18 | *unorbinop* $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ | → $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ | $\{U\}E$ | $*x$ |
| 19 | *unorbinop binop* | → *binop* | *math_rel* $U\{B\}$ | $*{=}$ |
| 20 | *binop binop* | → *binop* | *math_rel* $\{B_1\}\{B_2\}$ | $\gg{=}$ |
| 21 | *cast exp* | → *exp* | $C_\sqcup E$ | (**double**) $x$ |
| 22 | *cast semi* | → *exp semi* | | (**int**); |
| 23 | *sizeof_like cast* | → *exp* | | **sizeof**(**double**) |
| 24 | *sizeof_like exp* | → *exp* | $S_\sqcup E$ | **sizeof** $x$ |
| 25 | *int_like* $\left\{\begin{matrix} int\_like \\ struct\_like \end{matrix}\right\}$ | → $\left\{\begin{matrix} int\_like \\ struct\_like \end{matrix}\right\}$ | $I_\sqcup\left\{\begin{matrix} I \\ S \end{matrix}\right\}$ | **extern char** |

| | | | | | |
|---|---|---|---|---|---|
| 26 | *int_like exp* $\left\{\begin{matrix}raw\_int\\struct\_like\end{matrix}\right\}$ | $\rightarrow$ *int_like* $\left\{\begin{matrix}int\_like\\struct\_like\end{matrix}\right\}$ | | | **extern**`"Ada"` **int** |
| 27 | *int_like* $\left\{\begin{matrix}exp\\unorbinop\\semi\end{matrix}\right\}$ | $\rightarrow$ *decl_head* $\left\{\begin{matrix}exp\\unorbinop\\semi\end{matrix}\right\}$ | $D = I\left\{\begin{matrix}\sqcup\\\sqcup\end{matrix}\right\}$ | | **int** $x$ |
| 28 | *int_like colon* | $\rightarrow$ *decl_head colon* | $D = I_{\sqcup}$ | | **unsigned** : |
| 29 | *int_like prelangle* | $\rightarrow$ *int_like langle* | | | **C**$\langle$ |
| 30 | *int_like colcol* $\left\{\begin{matrix}exp\\int\_like\end{matrix}\right\}$ | $\rightarrow$ $\left\{\begin{matrix}exp\\int\_like\end{matrix}\right\}$ | | | **C**::$x$<br>**C**::**B** |
| 31 | *int_like cast lbrace* | $\rightarrow$ *fn_decl lbrace* | *IC in in* | | **C**$\langle$**void**$*\rangle\{$ |
| 32 | *int_like cast* | $\rightarrow$ *int_like* | | | **C**$\langle$**class T**$\rangle$ |
| 33 | *decl_head comma* | $\rightarrow$ *decl_head* | $DC_{\sqcup}$ | | **int** $x$, |
| 34 | *decl_head unorbinop* | $\rightarrow$ *decl_head* | $D\{U\}$ | | **int** $*$ |
| †35 | *decl_head exp* | $\rightarrow$ *decl_head* | $DE^*$ | | **int** $x$ |
| 36 | *decl_head* $\left\{\begin{matrix}binop\\colon\end{matrix}\right\}$ *exp* $\left\{\begin{matrix}comma\\semi\\rpar\end{matrix}\right\}$ | $\rightarrow$ *decl_head* $\left\{\begin{matrix}comma\\semi\\rpar\end{matrix}\right\}$ | $D = D\left\{\begin{matrix}B\\C\end{matrix}\right\}E$ | | initialization fields or default argument |
| 37 | *decl_head cast* | $\rightarrow$ *decl_head* | | | **int** $f$(**int**) |
| †38 | *decl_head* $\left\{\begin{matrix}int\_like\\lbrace\\decl\end{matrix}\right\}$ | $\rightarrow$ *fn_decl* $\left\{\begin{matrix}int\_like\\lbrace\\decl\end{matrix}\right\}$ | *F = D in in* | | **long** *time*( ) { |
| 39 | *decl_head semi* | $\rightarrow$ *decl* | | | **int** $n$; |
| 40 | *decl decl* | $\rightarrow$ *decl* | *D force D* | | **int** $n$; **double** $x$; |
| 41 | *decl* $\left\{\begin{matrix}stmt\\function\end{matrix}\right\}$ | $\rightarrow$ $\left\{\begin{matrix}stmt\\function\end{matrix}\right\}$ | *D big_force* $\left\{\begin{matrix}S\\F\end{matrix}\right\}$ | | **extern** $n$; *main* (){ } |
| †42 | *typedef_like decl_head* $\left\{\begin{matrix}exp\\int\_like\end{matrix}\right\}$ | $\rightarrow$ *typedef_like decl_head* | $D = D\left\{\begin{matrix}E^{**}\\I^{**}\end{matrix}\right\}$ | | **typedef char ch**; |
| 43 | *typedef_like decl_head semi* | $\rightarrow$ *decl* | $T_{\sqcup}D$ | | **typedef int x**, **y**; |
| 44 | *struct_like lbrace* | $\rightarrow$ *struct_head* | $S_{\sqcup}L$ | | **struct** { |
| 45 | *struct_like* $\left\{\begin{matrix}exp\\int\_like\end{matrix}\right\}$ *semi* | $\rightarrow$ *decl_head* | $S_{\sqcup}\left\{\begin{matrix}E^{**}\\I^{**}\end{matrix}\right\}$ | | **struct forward**; |
| 46 | *struct_like* $\left\{\begin{matrix}exp\\int\_like\end{matrix}\right\}$ *lbrace* | $\rightarrow$ *struct_head* | $S_{\sqcup}\left\{\begin{matrix}E^{**}\\I^{**}\end{matrix}\right\}_{\sqcup}L$ | | **struct name_info** { |
| 47 | *struct_like* $\left\{\begin{matrix}exp\\int\_like\end{matrix}\right\}$ *colon* | $\rightarrow$ *struct_like* $\left\{\begin{matrix}exp\\int\_like\end{matrix}\right\}$ *base* | | | **class C** : |
| †48 | *struct_like* $\left\{\begin{matrix}exp\\int\_like\end{matrix}\right\}$ | $\rightarrow$ *int_like* | $S_{\sqcup}\left\{\begin{matrix}E\\I\end{matrix}\right\}$ | | **struct name_info** $z$; |
| 49 | *struct_head* $\left\{\begin{matrix}decl\\stmt\\function\end{matrix}\right\}$ *rbrace* | $\rightarrow$ *int_like* | *S in force D out force R* | | **struct** { declaration } |
| 50 | *struct_head rbrace* | $\rightarrow$ *int_like* | $S\backslash, R$ | | **class C** { } |
| 51 | *fn_decl decl* | $\rightarrow$ *fn_decl* | *F force D* | | $f(z)$ **double** $z$; |
| 52 | *fn_decl stmt* | $\rightarrow$ *function* | *F out out force S* | | *main*() ... |
| 53 | *function* $\left\{\begin{matrix}stmt\\decl\\function\end{matrix}\right\}$ | $\rightarrow$ $\left\{\begin{matrix}stmt\\decl\\function\end{matrix}\right\}$ | *F big_force* $\left\{\begin{matrix}S\\D\\F\end{matrix}\right\}$ | | outer block |
| 54 | *lbrace rbrace* | $\rightarrow$ *stmt* | $L\backslash, R$ | | empty statement |
| 55 | *lbrace* $\left\{\begin{matrix}stmt\\decl\\function\end{matrix}\right\}$ *rbrace* | $\rightarrow$ *stmt* | *force L in force S force back R out force* | | compound statement |
| 56 | *lbrace exp* [*comma*] *rbrace* | $\rightarrow$ *exp* | | | initializer |

57 *if_like exp* → *if_clause* $I_{\sqcup}E$ **if** $(z)$
58 *for_like exp* → *else_like* $F_{\sqcup}E$ **while** (1)
59 *else_like lbrace* → *else_head lbrace* **else** {
60 *else_like stmt* → *stmt* *force E in bsp S out force* **else** $x \leftarrow 0$;
61 *else_head* $\left\{\begin{matrix} stmt \\ exp \end{matrix}\right\}$ → *stmt* *force E bsp noop cancel S bsp* **else** { $x \leftarrow 0$; }
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* $E_{\sqcup}I$ **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** $(x)$
66 *if_head* $\left\{\begin{matrix} stmt \\ exp \end{matrix}\right\}$ *else_like if_like* → *if_like force I bsp noop cancel S force* $E_{\sqcup}I$ **if** $(x)$ { $y$; } **else if**
67 *if_head* $\left\{\begin{matrix} stmt \\ exp \end{matrix}\right\}$ *else_like* → *else_like force I bsp noop cancel S force E* **if** $(x)$ { $y$; } **else**
68 *if_head* $\left\{\begin{matrix} stmt \\ exp \end{matrix}\right\}$ → *else_head* $\left\{\begin{matrix} stmt \\ exp \end{matrix}\right\}$ **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 semi* → *stmt* $C_{\sqcup}ES$ **return** 0;
73 *case_like exp colon* → *tag* $C_{\sqcup}EC$ **case** 0:
74 *tag tag* → *tag* $T_1$ *bsp* $T_2$ **case** 0: **case** 1:
75 *tag* $\left\{\begin{matrix} stmt \\ decl \\ function \end{matrix}\right\}$ → $\left\{\begin{matrix} stmt \\ decl \\ function \end{matrix}\right\}$ *force back T bsp S* **case** 0: $z \leftarrow 0$;
†76 *stmt* $\left\{\begin{matrix} stmt \\ decl \\ function \end{matrix}\right\}$ → $\left\{\begin{matrix} stmt \\ decl \\ function \end{matrix}\right\}$ $S\left\{\begin{matrix} force\ S \\ big\_force\ D \\ big\_force\ F \end{matrix}\right\}$ $x \leftarrow 1$; $y \leftarrow 2$;
77 *semi* → *stmt* $_{\sqcup}S$ empty statement
†78 *lproc* $\left\{\begin{matrix} if\_like \\ else\_like \\ define\_like \end{matrix}\right\}$ → *lproc* **#include**
**#else**
**#define**
79 *lproc rproc* → *insert* **#endif**
80 *lproc* $\left\{\begin{matrix} exp\ [exp] \\ function \end{matrix}\right\}$ *rproc* → *insert* $I_{\sqcup}\left\{\begin{matrix} E[_{\sqcup}\backslash 5E] \\ F \end{matrix}\right\}$ **#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 exp prerangle* → *cast* ⟨ 0 ⟩
87 *langle prerangle* → *cast* $L\backslash, P$ ⟨ ⟩
88 *langle* $\left\{\begin{matrix} decl\_head \\ int\_like \end{matrix}\right\}$ *prerangle* → *cast* ⟨**class C**⟩
89 *langle* $\left\{\begin{matrix} decl\_head \\ int\_like \end{matrix}\right\}$ *comma* → *langle* $\mathrm{L}\left\{\begin{matrix} D \\ I \end{matrix}\right\}$ C *opt*9 ⟨**class C**,
90 *public_like colon* → *tag* **private**:
91 *public_like* → *int_like* **private**
92 *colcol* $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ → $\left\{\begin{matrix} exp \\ int\_like \end{matrix}\right\}$ $::x$

| | | | | |
|---|---|---|---|---|
| †93 | *new_like* $\left\{\begin{matrix}exp\\raw\_int\end{matrix}\right\}$ | → *new_like* | $N_{\sqcup}E$ | **new** (1) |
| 94 | *new_like* $\left\{\begin{matrix}raw\_unorbin\\colcol\end{matrix}\right\}$ | → *new_like* | | **new** ::∗ |
| 95 | *new_like cast* | → *exp* | | **new** (∗) |
| †96 | *new_like* | → *exp* | | **new** |
| †97 | *operator_like* $\left\{\begin{matrix}binop\\unop\\unorbinop\end{matrix}\right\}$ | → *exp* | $O\{\left\{\begin{matrix}B\\U\\U\}\end{matrix}\right\}$ | **operator**+ |
| 98 | *operator_like* $\left\{\begin{matrix}new\_like\\sizeof\_like\end{matrix}\right\}$ | → *exp* | $O_{\sqcup}N$ | **operator delete** |
| 99 | *operator_like* | → *new_like* | | conversion operator |
| 100 | *catch_like* $\left\{\begin{matrix}cast\\exp\end{matrix}\right\}$ | → *fn_decl* | *CE in in* | **catch**(...) |
| 101 | *base public_like exp comma* | → *base* | $BP_{\sqcup}EC$ | : **public** *a*, |
| 102 | *base public_like exp* | → *base int_like* | $I = P_{\sqcup}E$ | : **public** *a* |
| 103 | *raw_rpar const_like* | → *raw_rpar* | $R_{\sqcup}C$ | ) **const**; |
| 104 | *raw_rpar* | → *rpar* | | ); |
| 105 | *raw_unorbin const_like* | → *raw_unorbin* | $RC\backslash_{\sqcup}$ | ∗**const** *x* |
| 106 | *raw_unorbin* | → *unorbinop* | | ∗ *x* |
| 107 | *const_like* | → *int_like* | | **const** *x* |
| 108 | *raw_int lpar* | → *exp* | | **complex**(*x*, *y*) |
| 109 | *raw_int* | → *int_like* | | **complex** *z* |
| 110 | *begin_arg end_arg* | → *exp* | | `@[`**char**∗`@]` |
| 111 | *any_other end_arg* | → *end_arg* | | **char**∗`@]` |

**†Notes**

Rule 35: The *exp* must not be immediately followed by *lpar* or *exp*.
Rule 38: The *int_like* must not be immediately followed by *colcol*.
Rule 42: The *exp* must not be immediately followed by *lpar* or *exp*.
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.
Rule 78: The *define_like* case calls *make_underlined* on the following scrap.
Rule 93: The *raw_int* must not be immediately followed by *prelangle* or *langle*.
Rule 96: The *new_like* must not be immediately followed by *lpar*, *raw_int*, or *struct_like*.
Rule 97: The operator after *operator_like* must not be immediately followed by a *binop*.

**103. Implementing the productions.** More specifically, a scrap is a structure consisting of a category *cat* and a **text_pointer** *trans*, which points to the translation in *tok_start*. When C text is to be processed with the grammar above, we form an array *scrap_info* containing the initial scraps. Our production rules have the nice property that the right-hand side is never longer than the left-hand side. Therefore it is convenient to use sequential allocation for the current sequence of scraps. Five pointers are used to manage the parsing:

*pp* is a pointer into *scrap_info*. We will try to match the category codes $pp\rightarrow cat$, $(pp+1)\rightarrow cat$, ... to the left-hand sides of productions.

*scrap_base*, *lo_ptr*, *hi_ptr*, and *scrap_ptr* are such that the current sequence of scraps appears in positions *scrap_base* through *lo_ptr* and *hi_ptr* through *scrap_ptr*, inclusive, in the *cat* and *trans* arrays. Scraps located between *scrap_base* and *lo_ptr* have been examined, while those in positions $\geq$ *hi_ptr* have not yet been looked at by the parsing process.

Initially *scrap_ptr* is set to the position of the final scrap to be parsed, and it doesn't change its value. The parsing process makes sure that $lo\_ptr \geq pp+3$, since productions have as many as four terms, by moving scraps from *hi_ptr* to *lo_ptr*. If there are fewer than $pp+3$ scraps left, the positions up to $pp+3$ are filled with blanks that will not match in any productions. Parsing stops when $pp \equiv lo\_ptr+1$ and $hi\_ptr \equiv scrap\_ptr+1$.

Since the *scrap* structure will later be used for other purposes, we declare its second element as unions.

```
⟨ Typedef declarations 18 ⟩ +≡
  typedef struct {
    eight_bits cat;
    eight_bits mathness;
    union {
      text_pointer Trans;
      ⟨ Rest of trans_plus union 231 ⟩
    } trans_plus;
  } scrap;
  typedef scrap *scrap_pointer;
```

**104.** `#define trans trans_plus.Trans /* translation texts of scraps */`

```
⟨ Global variables 17 ⟩ +≡
  scrap scrap_info[max_scraps]; /* memory array for scraps */
  scrap_pointer scrap_info_end ← scrap_info + max_scraps − 1; /* end of scrap_info */
  scrap_pointer pp; /* current position for reducing productions */
  scrap_pointer scrap_base; /* beginning of the current scrap sequence */
  scrap_pointer scrap_ptr; /* ending of the current scrap sequence */
  scrap_pointer lo_ptr; /* last scrap that has been examined */
  scrap_pointer hi_ptr; /* first scrap that has not been examined */
  scrap_pointer max_scr_ptr; /* largest value assumed by scrap_ptr */
```

**105.**

```
⟨ Set initial values 20 ⟩ +≡
  scrap_base ← scrap_info + 1;
  max_scr_ptr ← scrap_ptr ← scrap_info;
```

**106.** 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 '| ... |' */
  void print_text(p) /* 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 107 ⟩;
        }
      }
    fflush(stdout);
  }
```

**107.** ⟨ Print token $r$ in symbolic form 107 ⟩ ≡

```
switch (r) {
case math_rel: printf("\\mathrel{");
  break;
case big_cancel: printf("[ccancel]");
  break;
case cancel: printf("[cancel]");
  break;
case indent: printf("[indent]");
  break;
case outdent: printf("[outdent]");
  break;
case backup: printf("[backup]");
  break;
case opt: printf("[opt]");
  break;
case break_space: printf("[break]");
  break;
case force: printf("[force]");
  break;
case big_force: printf("[fforce]");
  break;
case preproc_line: printf("[preproc]");
  break;
case quoted_char: j++;
  printf("[%o]",(unsigned) *j);
  break;
case end_translation: printf("[quit]");
  break;
case inserted: printf("[inserted]");
  break;
default: putxchar(r);
}
```

This code is used in section 106

**108.** The production rules listed above are embedded directly into CWEAVE, since it is easier to do this than to write an interpretive system that would handle production systems in general. Several macros are defined here so that the program for each production is fairly short.

All of our productions conform to the general notion that some $k$ consecutive scraps starting at some position $j$ are to be replaced by a single scrap of some category $c$ whose translation is composed from the translations of the disappearing scraps. After this production has been applied, the production pointer *pp* should change by an amount $d$. Such a production can be represented by the quadruple $(j, k, c, d)$. For example, the production '*exp comma exp* → *exp*' would be represented by '$(pp, 3, exp, -2)$'; in this case the pointer *pp* should decrease by 2 after the production has been applied, because some productions with *exp* in their second or third positions might now match, but no productions have *exp* in the fourth position of their left-hand sides. Note that the value of $d$ is determined by the whole collection of productions, not by an individual one. The determination of $d$ has been done by hand in each case, based on the full set of productions but not on the grammar of C or on the rules for constructing the initial scraps.

We also attach a serial number to each production, so that additional information is available when debugging. For example, the program below contains the statement '$reduce(pp, 3, exp, -2, 4)$' when it implements the production just mentioned.

Before calling *reduce*, the program should have appended the tokens of the new translation to the *tok_mem* array. We commonly want to append copies of several existing translations, and macros are defined to simplify these common cases. For example, *app2*(*pp*) will append the translations of two consecutive scraps, *pp*→*trans* and (*pp* + 1)→*trans*, to the current token list. If the entire new translation is formed in this way, we write '$squash(j, k, c, d, n)$' instead of '$reduce(j, k, c, d, n)$'. For example, '$squash(pp, 3, exp, -2, 3)$' is an abbreviation for '*app3*(*pp*); *reduce*(*pp*, 3, *exp*, −2, 3)'.

A couple more words of explanation: Both *big_app* and *app* append a token (while *big_app1* to *big_app4* append the specified number of scrap translations) to the current token list. The difference between *big_app* and *app* is simply that *big_app* checks whether there can be a conflict between math and non-math tokens, and intercalates a '$' token if necessary. When in doubt what to use, use *big_app*.

The *mathness* is an attribute of scraps that says whether they are to be printed in a math mode context or not. It is separate from the "part of speech" (the *cat*) because to make each *cat* have a fixed *mathness* (as in the original WEAVE) would multiply the number of necessary production rules.

The low two bits (i.e. *mathness* % 4) control the left boundary. (We need two bits because we allow cases *yes_math*, *no_math* and *maybe_math*, which can go either way.) The next two bits (i.e. *mathness*/4) control the right boundary. If we combine two scraps and the right boundary of the first has a different mathness from the left boundary of the second, we insert a $ in between. Similarly, if at printing time some irreducible scrap has a *yes_math* boundary the scrap gets preceded or followed by a $. The left boundary is *maybe_math* if and only if the right boundary is.

The code below is an exact translation of the production rules into C, using such macros, and the reader should have no difficulty understanding the format by comparing the code with the symbolic productions as they were listed earlier.

```
#define no_math 2 /* should be in horizontal mode */
#define yes_math 1 /* should be in math mode */
#define maybe_math 0 /* works in either horizontal or math mode */
#define big_app2(a) big_app1(a); big_app1(a + 1)
#define big_app3(a) big_app2(a); big_app1(a + 2)
#define big_app4(a) big_app3(a); big_app1(a + 3)
#define app(a) *(tok_ptr++) ← a
#define app1(a) *(tok_ptr++) ← tok_flag + (int) ((a)→trans − tok_start)
⟨ Global variables 17 ⟩ +≡
  int cur_mathness, init_mathness;
```

```
109.  void app_str(s)
        char *s;
  {
    while (*s) app_tok(*(s++));
  }
  void big_app(a)
        token a;
  {
    if (a ≡ '␣' ∨ (a ≥ big_cancel ∧ a ≤ big_force))      /* non-math token */
    {
      if (cur_mathness ≡ maybe_math) init_mathness ← no_math;
      else if (cur_mathness ≡ yes_math) app_str("{}$");
      cur_mathness ← no_math;
    }
    else {
      if (cur_mathness ≡ maybe_math) init_mathness ← yes_math;
      else if (cur_mathness ≡ no_math) app_str("${}");
      cur_mathness ← yes_math;
    }
    app(a);
  }
  void big_app1(a)
        scrap_pointer a;
  {
    switch (a→mathness % 4) {      /* left boundary */
    case (no_math):
      if (cur_mathness ≡ maybe_math) init_mathness ← no_math;
      else if (cur_mathness ≡ yes_math) app_str("{}$");
      cur_mathness ← a→mathness/4;      /* right boundary */
      break;
    case (yes_math):
      if (cur_mathness ≡ maybe_math) init_mathness ← yes_math;
      else if (cur_mathness ≡ no_math) app_str("${}");
      cur_mathness ← a→mathness/4;      /* right boundary */
      break;
    case (maybe_math):      /* no changes */
      break;
    }
    app(tok_flag + (int) ((a)→trans − tok_start));
  }
```

**110.** Let us consider the big switch for productions now, before looking at its context. We want to design the program so that this switch works, so we might as well not keep ourselves in suspense about exactly what code needs to be provided with a proper environment.

```
#define cat1 (pp + 1)→cat
#define cat2 (pp + 2)→cat
#define cat3 (pp + 3)→cat
#define lhs_not_simple (pp→cat ≠ semi ∧ pp→cat ≠ raw_int ∧ pp→cat ≠ raw_unorbin ∧ pp→cat ≠
        raw_rpar ∧ pp→cat ≠ const_like)
⟨ Match a production at pp, or increase pp if there is no match 110 ⟩ ≡
  {
    if (cat1 ≡ end_arg ∧ lhs_not_simple)
      if (pp→cat ≡ begin_arg) squash(pp, 2, exp, −2, 110);
      else squash(pp, 2, end_arg, −1, 111);
    else if (cat1 ≡ insert) squash(pp, 2, pp→cat, −2, 0);
    else if (cat2 ≡ insert) squash(pp + 1, 2, (pp + 1)→cat, −1, 0);
    else if (cat3 ≡ insert) squash(pp + 2, 2, (pp + 2)→cat, 0, 0);
    else
      switch (pp→cat) {
      case exp: ⟨ Cases for exp 117 ⟩; break;
      case lpar: ⟨ Cases for lpar 118 ⟩; break;
      case question: ⟨ Cases for question 119 ⟩; break;
      case unop: ⟨ Cases for unop 120 ⟩; break;
      case unorbinop: ⟨ Cases for unorbinop 121 ⟩; break;
      case binop: ⟨ Cases for binop 122 ⟩; break;
      case cast: ⟨ Cases for cast 123 ⟩; break;
      case sizeof_like: ⟨ Cases for sizeof_like 124 ⟩; break;
      case int_like: ⟨ Cases for int_like 125 ⟩; break;
      case decl_head: ⟨ Cases for decl_head 126 ⟩; break;
      case decl: ⟨ Cases for decl 127 ⟩; break;
      case typedef_like: ⟨ Cases for typedef_like 128 ⟩; break;
      case struct_like: ⟨ Cases for struct_like 129 ⟩; break;
      case struct_head: ⟨ Cases for struct_head 130 ⟩; break;
      case fn_decl: ⟨ Cases for fn_decl 131 ⟩; break;
      case function: ⟨ Cases for function 132 ⟩; break;
      case lbrace: ⟨ Cases for lbrace 133 ⟩; break;
      case do_like: ⟨ Cases for do_like 140 ⟩; break;
      case if_like: ⟨ Cases for if_like 134 ⟩; break;
      case for_like: ⟨ Cases for for_like 135 ⟩; break;
      case else_like: ⟨ Cases for else_like 136 ⟩; break;
      case if_clause: ⟨ Cases for if_clause 138 ⟩; break;
      case if_head: ⟨ Cases for if_head 139 ⟩; break;
      case else_head: ⟨ Cases for else_head 137 ⟩; break;
      case case_like: ⟨ Cases for case_like 141 ⟩; break;
      case stmt: ⟨ Cases for stmt 143 ⟩; break;
      case tag: ⟨ Cases for tag 142 ⟩; break;
      case semi: ⟨ Cases for semi 144 ⟩; break;
      case lproc: ⟨ Cases for lproc 145 ⟩; break;
      case section_scrap: ⟨ Cases for section_scrap 146 ⟩; break;
      case insert: ⟨ Cases for insert 147 ⟩; break;
      case prelangle: ⟨ Cases for prelangle 148 ⟩; break;
      case prerangle: ⟨ Cases for prerangle 149 ⟩; break;
      case langle: ⟨ Cases for langle 150 ⟩; break;
      case public_like: ⟨ Cases for public_like 151 ⟩; break;
      case colcol: ⟨ Cases for colcol 152 ⟩; break;
      case new_like: ⟨ Cases for new_like 153 ⟩; break;
```

```
      case operator_like: ⟨ Cases for operator_like 154 ⟩; break;
      case catch_like: ⟨ Cases for catch_like 155 ⟩; break;
      case base: ⟨ Cases for base 156 ⟩; break;
      case raw_rpar: ⟨ Cases for raw_rpar 157 ⟩; break;
      case raw_unorbin: ⟨ Cases for raw_unorbin 158 ⟩; break;
      case const_like: ⟨ Cases for const_like 159 ⟩; break;
      case raw_int: ⟨ Cases for raw_int 160 ⟩; break;
      }
    pp++;      /* if no match was found, we move to the right */
  }
```

This code is used in section 165

**111.** In C, new specifier names can be defined via **typedef**, and we want to make the parser recognize future occurrences of the identifier thus defined as specifiers. This is done by the procedure *make_reserved*, which changes the *ilk* of the relevant identifier.

We first need a procedure to recursively seek the first identifier in a token list, because the identifier might be enclosed in parentheses, as when one defines a function returning a pointer.

```
#define no_ident_found 0      /* distinct from any identifier token */
  token_pointer find_first_ident(p)
      text_pointer p;
  {
    token_pointer q;      /* token to be returned */
    token_pointer j;      /* token being looked at */
    sixteen_bits r;      /* remainder of token after the flag has been stripped off */
    if (p ≥ text_ptr) confusion("find_first_ident");
    for (j ← *p; j < *(p + 1); j++) {
      r ← *j % id_flag;
      switch (*j/id_flag) {
      case 1: case 2: return j;
      case 4: case 5:      /* tok_flag or inner_tok_flag */
        if ((q ← find_first_ident(tok_start + r)) ≠ no_ident_found) return q;
      default: ;      /* char, section_flag, fall thru: move on to next token */
        if (*j ≡ inserted) return no_ident_found;      /* ignore inserts */
      }
    }
    return no_ident_found;
  }
```

**112.** The scraps currently being parsed must be inspected for any occurrence of the identifier that we're making reserved; hence the **for** loop below.

```
void make_reserved(p)      /* make the first identifier in p→trans like int */
    scrap_pointer p;
{
  sixteen_bits tok_value;      /* the name of this identifier, plus its flag */
  token_pointer tok_loc;       /* pointer to tok_value */
  if ((tok_loc ← find_first_ident(p→trans)) ≡ no_ident_found) return;
      /* this should not happen */
  tok_value ← *tok_loc;
  for ( ; p ≤ scrap_ptr; p ≡ lo_ptr ? p ← hi_ptr : p++) {
    if (p→cat ≡ exp) {
      if (**(p→trans) ≡ tok_value) {
        p→cat ← raw_int;
        **(p→trans) ← tok_value % id_flag + res_flag;
      }
    }
  }
  (name_dir + (sixteen_bits) (tok_value % id_flag))→ilk ← raw_int;
  *tok_loc ← tok_value % id_flag + res_flag;
}
```

**113.** In the following situations we want to mark the occurrence of an identifier as a definition: when *make_reserved* is just about to be used; after a specifier, as in **char** ∗∗*argv*; before a colon, as in *found*:; and in the declaration of a function, as in *main*(){...;}. This is accomplished by the invocation of *make_underlined* at appropriate times. Notice that, in the declaration of a function, we only find out that the identifier is being defined after it has been swallowed up by an *exp*.

```
void make_underlined(p)      /* underline the entry for the first identifier in p→trans */
    scrap_pointer p;
{
  token_pointer tok_loc;       /* where the first identifier appears */
  if ((tok_loc ← find_first_ident(p→trans)) ≡ no_ident_found) return;
      /* this happens after parsing the ( ) in double f( ); */
  xref_switch ← def_flag;
  underline_xref(*tok_loc % id_flag + name_dir);
}
```

**114.** We cannot use *new_xref* to underline a cross-reference at this point because this would just make a new cross-reference at the end of the list. We actually have to search through the list for the existing cross-reference.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void underline_xref( );
```

**115.** **void** *underline_xref*(*p*)
**name_pointer** *p*;
{
**xref_pointer** *q* ← (**xref_pointer**) *p*→*xref*; /* pointer to cross-reference being examined */
**xref_pointer** *r*; /* temporary pointer for permuting cross-references */
**sixteen_bits** *m*; /* cross-reference value to be installed */
**sixteen_bits** *n*; /* cross-reference value being examined */
**if** (*no_xref*) **return**;
*m* ← *section_count* + *xref_switch*;
**while** (*q* ≠ *xmem*) {
*n* ← *q*→*num*;
**if** (*n* ≡ *m*) **return**;
**else if** (*m* ≡ *n* + *def_flag*) {
*q*→*num* ← *m*;
**return**;
}
**else if** (*n* ≥ *def_flag* ∧ *n* < *m*) **break**;
*q* ← *q*→*xlink*;
}
⟨ Insert new cross-reference at *q*, not at beginning of list 116 ⟩;
}

**116.** We get to this section only when the identifier is one letter long, so it didn't get a non-underlined entry during phase one. But it may have got some explicitly underlined entries in later sections, so in order to preserve the numerical order of the entries in the index, we have to insert the new cross-reference not at the beginning of the list (namely, at *p*→*xref*), but rather right before *q*.

⟨ Insert new cross-reference at *q*, not at beginning of list 116 ⟩ ≡
*append_xref*(0); /* this number doesn't matter */
*xref_ptr*→*xlink* ← (**xref_pointer**) *p*→*xref*;
*r* ← *xref_ptr*;
*p*→*xref* ← (**char** *) *xref_ptr*;
**while** (*r*→*xlink* ≠ *q*) {
*r*→*num* ← *r*→*xlink*→*num*;
*r* ← *r*→*xlink*;
}
*r*→*num* ← *m*; /* everything from *q* on is left undisturbed */

This code is used in section 115

**117.** 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 117 ⟩ ≡
  if (cat1 ≡ lbrace ∨ cat1 ≡ int_like ∨ cat1 ≡ decl) {
    make_underlined(pp);
    big_app1(pp);
    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 ≡ unorbinop) ∧ 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 ≡ exp ∨ cat1 ≡ cast) squash(pp, 2, exp, −2, 5);
  else if (cat1 ≡ semi) squash(pp, 2, stmt, −1, 6);
  else if (cat1 ≡ colon) {
    make_underlined(pp);
    squash(pp, 2, tag, 0, 7);
  }
  else if (cat1 ≡ base) {
    if (cat2 ≡ int_like ∧ cat3 ≡ comma) {
      big_app1(pp + 1);
      big_app('␣');
      big_app2(pp + 2);
      app(opt);
      app('9');
      reduce(pp + 1, 3, base, 0, 8);
    }
    else if (cat2 ≡ int_like ∧ cat3 ≡ lbrace) {
      big_app1(pp);
      big_app('␣');
      big_app1(pp + 1);
      big_app('␣');
      big_app1(pp + 2);
      reduce(pp, 3, exp, −1, 9);
    }
  }
  else if (cat1 ≡ rbrace) squash(pp, 1, stmt, −1, 10);
```

This code is used in section 110

**118.** ⟨ Cases for *lpar* 118 ⟩ ≡

```
if ((cat1 ≡ exp ∨ cat1 ≡ unorbinop) ∧ cat2 ≡ rpar) squash(pp, 3, exp, −2, 11);
else if (cat1 ≡ rpar) {
  big_app1(pp);
  app('\\');
  app(',');
  big_app1(pp + 1);
  reduce(pp, 2, exp, −2, 12);
}
else if (cat1 ≡ decl_head ∨ cat1 ≡ int_like ∨ cat1 ≡ exp) {
  if (cat2 ≡ rpar) squash(pp, 3, cast, −2, 13);
  else if (cat2 ≡ comma) {
    big_app3(pp);
    app(opt);
    app('9');
    reduce(pp, 3, lpar, 0, 14);
  }
}
else if (cat1 ≡ stmt ∨ cat1 ≡ decl) {
  big_app2(pp);
  big_app('␣');
  reduce(pp, 2, lpar, 0, 15);
}
```

This code is used in section 110

**119.** ⟨ Cases for *question* 119 ⟩ ≡

```
if (cat1 ≡ exp ∧ cat2 ≡ colon) squash(pp, 3, binop, −2, 16);
```

This code is used in section 110

**120.** ⟨ Cases for *unop* 120 ⟩ ≡

```
if (cat1 ≡ exp ∨ cat1 ≡ int_like) squash(pp, 2, cat1, −2, 17);
```

This code is used in section 110

**121.** ⟨ Cases for *unorbinop* 121 ⟩ ≡

```
if (cat1 ≡ exp ∨ cat1 ≡ int_like) {
  big_app('{');
  big_app1(pp);
  big_app('}');
  big_app1(pp + 1);
  reduce(pp, 2, cat1, −2, 18);
}
else if (cat1 ≡ binop) {
  big_app(math_rel);
  big_app1(pp);
  big_app('{');
  big_app1(pp + 1);
  big_app('}');
  big_app('}');
  reduce(pp, 2, binop, −1, 19);
}
```

This code is used in section 110

**122.** ⟨ Cases for *binop* 122 ⟩ ≡

```
if (cat1 ≡ binop) {
  big_app(math_rel);
  big_app('{');
  big_app1(pp);
  big_app('}');
  big_app('{');
  big_app1(pp + 1);
  big_app('}');
  big_app('}');
  reduce(pp, 2, binop, −1, 20);
}
```

This code is used in section 110

**123.** ⟨ Cases for *cast* 123 ⟩ ≡

```
if (cat1 ≡ exp) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, exp, −2, 21);
}
else if (cat1 ≡ semi) squash(pp, 1, exp, −2, 22);
```

This code is used in section 110

**124.** ⟨ Cases for *sizeof_like* 124 ⟩ ≡

```
if (cat1 ≡ cast) squash(pp, 2, exp, −2, 23);
else if (cat1 ≡ exp) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, exp, −2, 24);
}
```

This code is used in section 110

**125.** ⟨ Cases for *int_like* 125 ⟩ ≡

```
if (cat1 ≡ int_like ∨ cat1 ≡ struct_like) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, cat1, −2, 25);
}
else if (cat1 ≡ exp ∧ (cat2 ≡ raw_int ∨ cat2 ≡ struct_like)) squash(pp, 2, int_like, −2, 26);
else if (cat1 ≡ exp ∨ cat1 ≡ unorbinop ∨ cat1 ≡ semi) {
  big_app1(pp);
  if (cat1 ≠ semi) big_app('␣');
  reduce(pp, 1, decl_head, −1, 27);
}
else if (cat1 ≡ colon) {
  big_app1(pp);
  big_app('␣');
  reduce(pp, 1, decl_head, 0, 28);
}
else if (cat1 ≡ prelangle) squash(pp + 1, 1, langle, 1, 29);
else if (cat1 ≡ colcol ∧ (cat2 ≡ exp ∨ cat2 ≡ int_like)) squash(pp, 3, cat2, −2, 30);
else if (cat1 ≡ cast) {
  if (cat2 ≡ lbrace) {
    big_app2(pp);
    big_app(indent);
    big_app(indent);
    reduce(pp, 2, fn_decl, 1, 31);
  }
  else squash(pp, 2, int_like, −2, 32);
}
```

This code is used in section 110

**126.** ⟨ Cases for *decl_head* 126 ⟩ ≡

```
if (cat1 ≡ comma) {
  big_app2(pp);
  big_app('␣');
  reduce(pp, 2, decl_head, −1, 33);
}
else if (cat1 ≡ unorbinop) {
  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) {
  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 ∧ cat2 ≠ colcol) ∨ cat1 ≡ decl) {
  big_app1(pp);
  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 110

**127.** ⟨ Cases for *decl* 127 ⟩ ≡

```
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);
  big_app(big_force);
  big_app1(pp + 1);
  reduce(pp, 2, cat1, −1, 41);
}
```

This code is used in section 110

**128.** ⟨ Cases for *typedef_like* 128 ⟩ ≡

```
if (cat1 ≡ decl_head)
  if ((cat2 ≡ exp ∧ cat3 ≠ lpar ∧ cat3 ≠ exp) ∨ cat2 ≡ int_like) {
    make_underlined(pp + 2);
    make_reserved(pp + 2);
    big_app2(pp + 1);
    reduce(pp + 1, 2, decl_head, 0, 42);
  }
  else if (cat2 ≡ semi) {
    big_app1(pp);
    big_app('␣');
    big_app2(pp + 1);
    reduce(pp, 3, decl, −1, 43);
  }
```

This code is used in section 110

**129.** ⟨ Cases for *struct_like* 129 ⟩ ≡

```
if (cat1 ≡ lbrace) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, struct_head, 0, 44);
}
else if (cat1 ≡ exp ∨ cat1 ≡ int_like) {
  if (cat2 ≡ lbrace ∨ cat2 ≡ semi) {
    make_underlined(pp + 1);
    make_reserved(pp + 1);
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    if (cat2 ≡ semi) reduce(pp, 2, decl_head, 0, 45);
    else {
      big_app('␣');
      big_app1(pp + 2);
      reduce(pp, 3, struct_head, 0, 46);
    }
  }
  else if (cat2 ≡ colon) squash(pp + 2, 1, base, −1, 47);
  else if (cat2 ≠ base) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    reduce(pp, 2, int_like, −2, 48);
  }
}
```

This code is used in section 110

**130.** ⟨ Cases for *struct_head* 130 ⟩ ≡

```
if ((cat1 ≡ decl ∨ cat1 ≡ stmt ∨ cat1 ≡ function) ∧ cat2 ≡ rbrace) {
  big_app1(pp);
  big_app(indent);
  big_app(force);
  big_app1(pp + 1);
  big_app(outdent);
  big_app(force);
  big_app1(pp + 2);
  reduce(pp, 3, int_like, −2, 49);
}
else if (cat1 ≡ rbrace) {
  big_app1(pp);
  app_str("\\,");
  big_app1(pp + 1);
  reduce(pp, 2, int_like, −2, 50);
}
```

This code is used in section 110

**131.** ⟨ Cases for *fn_decl* 131 ⟩ ≡

```
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);
  app(outdent);
  app(outdent);
  big_app(force);
  big_app1(pp + 1);
  reduce(pp, 2, function, −1, 52);
}
```

This code is used in section 110

**132.** ⟨ Cases for *function* 132 ⟩ ≡

```
if (cat1 ≡ function ∨ cat1 ≡ decl ∨ cat1 ≡ stmt) {
  big_app1(pp);
  big_app(big_force);
  big_app1(pp + 1);
  reduce(pp, 2, cat1, −1, 53);
}
```

This code is used in section 110

**133.** ⟨ Cases for *lbrace* 133 ⟩ ≡

```
if (cat1 ≡ rbrace) {
  big_app1(pp);
  app('\\');
  app(',');
  big_app1(pp + 1);
  reduce(pp, 2, stmt, −1, 54);
}
else if ((cat1 ≡ stmt ∨ cat1 ≡ decl ∨ cat1 ≡ function) ∧ cat2 ≡ rbrace) {
  big_app(force);
  big_app1(pp);
  big_app(indent);
  big_app(force);
  big_app1(pp + 1);
  big_app(force);
  big_app(backup);
  big_app1(pp + 2);
  big_app(outdent);
  big_app(force);
  reduce(pp, 3, stmt, −1, 55);
}
else if (cat1 ≡ exp) {
  if (cat2 ≡ rbrace) squash(pp, 3, exp, −2, 56);
  else if (cat2 ≡ comma ∧ cat3 ≡ rbrace) squash(pp, 4, exp, −2, 56);
}
```

This code is used in section 110

**134.** ⟨ Cases for *if_like* 134 ⟩ ≡

```
if (cat1 ≡ exp) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, if_clause, 0, 57);
}
```

This code is used in section 110

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

```
if (cat1 ≡ exp) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, else_like, −2, 58);
}
```

This code is used in section 110

**136.** ⟨ Cases for *else_like* 136 ⟩ ≡

```
if (cat1 ≡ lbrace) squash(pp, 1, else_head, 0, 59);
else if (cat1 ≡ stmt) {
  big_app(force);
  big_app1(pp);
  big_app(indent);
  big_app(break_space);
  big_app1(pp + 1);
  big_app(outdent);
  big_app(force);
  reduce(pp, 2, stmt, −1, 60);
}
```

This code is used in section 110

**137.** ⟨ Cases for *else_head* 137 ⟩ ≡

```
if (cat1 ≡ stmt ∨ cat1 ≡ exp) {
  big_app(force);
  big_app1(pp);
  big_app(break_space);
  app(noop);
  big_app(cancel);
  big_app1(pp + 1);
  big_app(force);
  reduce(pp, 2, stmt, −1, 61);
}
```

This code is used in section 110

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

```
if (cat1 ≡ lbrace) squash(pp, 1, if_head, 0, 62);
else if (cat1 ≡ stmt) {
  if (cat2 ≡ else_like) {
    big_app(force);
    big_app1(pp);
    big_app(indent);
    big_app(break_space);
    big_app1(pp + 1);
    big_app(outdent);
    big_app(force);
    big_app1(pp + 2);
    if (cat3 ≡ if_like) {
      big_app('␣');
      big_app1(pp + 3);
      reduce(pp, 4, if_like, 0, 63);
    } else reduce(pp, 3, else_like, 0, 64);
  }
  else squash(pp, 1, else_like, 0, 65);
}
```

This code is used in section 110

**139.** ⟨ Cases for *if_head* 139 ⟩ ≡

```
if (cat1 ≡ stmt ∨ cat1 ≡ exp) {
  if (cat2 ≡ else_like) {
    big_app(force);
    big_app1(pp);
    big_app(break_space);
    app(noop);
    big_app(cancel);
    big_app1(pp + 1);
    big_app(force);
    big_app1(pp + 2);
    if (cat3 ≡ if_like) {
      big_app('␣');
      big_app1(pp + 3);
      reduce(pp, 4, if_like, 0, 66);
    } else reduce(pp, 3, else_like, 0, 67);
  }
  else squash(pp, 1, else_head, 0, 68);
}
```

This code is used in section 110

**140.** ⟨ Cases for *do_like* 140 ⟩ ≡

```
if (cat1 ≡ stmt ∧ cat2 ≡ else_like ∧ cat3 ≡ semi) {
  big_app1(pp);
  big_app(break_space);
  app(noop);
  big_app(cancel);
  big_app1(pp + 1);
  big_app(cancel);
  app(noop);
  big_app(break_space);
  big_app2(pp + 2);
  reduce(pp, 4, stmt, −1, 69);
}
```

This code is used in section 110

**141.** ⟨ Cases for *case_like* 141 ⟩ ≡

```
if (cat1 ≡ semi) squash(pp, 2, stmt, −1, 70);
else if (cat1 ≡ colon) squash(pp, 2, tag, −1, 71);
else if (cat1 ≡ exp) {
  if (cat2 ≡ semi) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    big_app1(pp + 2);
    reduce(pp, 3, stmt, −1, 72);
  }
  else if (cat2 ≡ colon) {
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    big_app1(pp + 2);
    reduce(pp, 3, tag, −1, 73);
  }
}
```

This code is used in section 110

**142.** ⟨ Cases for *tag* 142 ⟩ ≡

```
if (cat1 ≡ tag) {
  big_app1(pp);
  big_app(break_space);
  big_app1(pp + 1);
  reduce(pp, 2, tag, −1, 74);
}
else if (cat1 ≡ stmt ∨ cat1 ≡ decl ∨ cat1 ≡ function) {
  big_app(force);
  big_app(backup);
  big_app1(pp);
  big_app(break_space);
  big_app1(pp + 1);
  reduce(pp, 2, cat1, −1, 75);
}
```

This code is used in section 110

**143.** The user can decide at run-time whether short statements should be grouped together on the same line.

**#define** *force_lines* *flags*['f'] /* should each statement be on its own line? */

⟨ Cases for *stmt* 143 ⟩ ≡

```
if (cat1 ≡ stmt ∨ cat1 ≡ decl ∨ cat1 ≡ function) {
  big_app1(pp);
  if (cat1 ≡ function) big_app(big_force);
  else if (cat1 ≡ decl) big_app(big_force);
  else if (force_lines) big_app(force);
  else big_app(break_space);
  big_app1(pp + 1);
  reduce(pp, 2, cat1, −1, 76);
}
```

This code is used in section 110

**144.** ⟨ Cases for *semi* 144 ⟩ ≡

```
big_app('␣');
big_app1(pp);
reduce(pp, 1, stmt, −1, 77);
```

This code is used in section 110

**145.** ⟨ Cases for *lproc* 145 ⟩ ≡

```
if (cat1 ≡ define_like) make_underlined(pp + 2);
if (cat1 ≡ else_like ∨ cat1 ≡ if_like ∨ cat1 ≡ define_like) squash(pp, 2, lproc, 0, 78);
else if (cat1 ≡ rproc) {
  app(inserted);
  big_app2(pp);
  reduce(pp, 2, insert, −1, 79);
}
else if (cat1 ≡ exp ∨ cat1 ≡ function) {
  if (cat2 ≡ rproc) {
    app(inserted);
    big_app1(pp);
    big_app('␣');
    big_app2(pp + 1);
    reduce(pp, 3, insert, −1, 80);
  }
  else if (cat2 ≡ exp ∧ cat3 ≡ rproc ∧ cat1 ≡ exp) {
    app(inserted);
    big_app1(pp);
    big_app('␣');
    big_app1(pp + 1);
    app_str("␣\\5");
    big_app2(pp + 2);
    reduce(pp, 4, insert, −1, 80);
  }
}
```

This code is used in section 110

**146.** ⟨ Cases for *section_scrap* 146 ⟩ ≡

```
if (cat1 ≡ semi) {
  big_app2(pp);
  big_app(force);
  reduce(pp, 2, stmt, −2, 81);
}
else squash(pp, 1, exp, −2, 82);
```

This code is used in section 110

**147.** ⟨ Cases for *insert* 147 ⟩ ≡

```
if (cat1) squash(pp, 2, cat1, 0, 83);
```

This code is used in section 110

**148.** ⟨ Cases for *prelangle* 148 ⟩ ≡

```
init_mathness ← cur_mathness ← yes_math;
app('<');
reduce(pp, 1, binop, −2, 84);
```

This code is used in section 110

**149.** ⟨ Cases for *prerangle* 149 ⟩ ≡
```
init_mathness ← cur_mathness ← yes_math;
app('>');
reduce(pp, 1, binop, −2, 85);
```
This code is used in section 110

**150.** ⟨ Cases for *langle* 150 ⟩ ≡
```
if (cat1 ≡ exp ∧ cat2 ≡ prerangle) squash(pp, 3, cast, −1, 86);
else if (cat1 ≡ prerangle) {
  big_app1(pp);
  app('\\');
  app(',');
  big_app1(pp + 1);
  reduce(pp, 2, cast, −1, 87);
}
else if (cat1 ≡ decl_head ∨ cat1 ≡ int_like) {
  if (cat2 ≡ prerangle) squash(pp, 3, cast, −1, 88);
  else if (cat2 ≡ comma) {
    big_app3(pp);
    app(opt);
    app('9');
    reduce(pp, 3, langle, 0, 89);
  }
}
```
This code is used in section 110

**151.** ⟨ Cases for *public_like* 151 ⟩ ≡
```
if (cat1 ≡ colon) squash(pp, 2, tag, −1, 90);
else squash(pp, 1, int_like, −2, 91);
```
This code is used in section 110

**152.** ⟨ Cases for *colcol* 152 ⟩ ≡
```
if (cat1 ≡ exp ∨ cat1 ≡ int_like) squash(pp, 2, cat1, −2, 92);
```
This code is used in section 110

**153.** ⟨ Cases for *new_like* 153 ⟩ ≡
```
if (cat1 ≡ exp ∨ (cat1 ≡ raw_int ∧ cat2 ≠ prelangle ∧ cat2 ≠ langle)) {
  big_app1(pp);
  big_app(' ');
  big_app1(pp + 1);
  reduce(pp, 2, new_like, 0, 93);
}
else if (cat1 ≡ raw_unorbin ∨ cat1 ≡ colcol) squash(pp, 2, new_like, 0, 94);
else if (cat1 ≡ cast) squash(pp, 2, exp, −2, 95);
else if (cat1 ≠ lpar ∧ cat1 ≠ raw_int ∧ cat1 ≠ struct_like) squash(pp, 1, exp, −2, 96);
```
This code is used in section 110

**154.** ⟨ Cases for *operator_like* 154 ⟩ ≡

```
if (cat1 ≡ binop ∨ cat1 ≡ unop ∨ cat1 ≡ unorbinop) {
  if (cat2 ≡ binop) break;
  big_app1(pp);
  big_app('{');
  big_app1(pp + 1);
  big_app('}');
  reduce(pp, 2, exp, −2, 97);
}
else if (cat1 ≡ new_like ∨ cat1 ≡ sizeof_like) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, exp, −2, 98);
}
else squash(pp, 1, new_like, 0, 99);
```

This code is used in section 110

**155.** ⟨ Cases for *catch_like* 155 ⟩ ≡

```
if (cat1 ≡ cast ∨ cat1 ≡ exp) {
  big_app2(pp);
  big_app(indent);
  big_app(indent);
  reduce(pp, 2, fn_decl, 0, 100);
}
```

This code is used in section 110

**156.** ⟨ Cases for *base* 156 ⟩ ≡

```
if (cat1 ≡ public_like ∧ cat2 ≡ exp) {
  if (cat3 ≡ comma) {
    big_app2(pp);
    big_app('␣');
    big_app2(pp + 2);
    reduce(pp, 4, base, 0, 101);
  }
  else {
    big_app1(pp + 1);
    big_app('␣');
    big_app1(pp + 2);
    reduce(pp + 1, 2, int_like, −1, 102);
  }
}
```

This code is used in section 110

**157.** ⟨ Cases for *raw_rpar* 157 ⟩ ≡

```
if (cat1 ≡ const_like) {
  big_app1(pp);
  big_app('␣');
  big_app1(pp + 1);
  reduce(pp, 2, raw_rpar, 0, 103);
}
else squash(pp, 1, rpar, −3, 104);
```

This code is used in section 110

**158.** ⟨ Cases for *raw_unorbin* 158 ⟩ ≡

```
if (cat1 ≡ const_like) {
  big_app2(pp);
  app_str("\\␣");
  reduce(pp, 2, raw_unorbin, 0, 105);
}
else squash(pp, 1, unorbinop, −2, 106);
```

This code is used in section 110

**159.** ⟨ Cases for *const_like* 159 ⟩ ≡

```
squash(pp, 1, int_like, −2, 107);
```

This code is used in section 110

**160.** ⟨ Cases for *raw_int* 160 ⟩ ≡

```
if (cat1 ≡ lpar) squash(pp, 1, exp, −2, 108);
else squash(pp, 1, int_like, −3, 109);
```

This code is used in section 110

**161.** The '*freeze_text*' macro is used to give official status to a token list. Before saying *freeze_text*, items are appended to the current token list, and we know that the eventual number of this token list will be the current value of *text_ptr*. But no list of that number really exists as yet, because no ending point for the current list has been stored in the *tok_start* array. After saying *freeze_text*, the old current token list becomes legitimate, and its number is the current value of *text_ptr* − 1 since *text_ptr* has been increased. The new current token list is empty and ready to be appended to. Note that *freeze_text* does not check to see that *text_ptr* hasn't gotten too large, since it is assumed that this test was done beforehand.

```
#define freeze_text ∗(++text_ptr) ← tok_ptr
```

**162.** Here's the *reduce* procedure used in our code for productions:

```
void reduce(j, k, c, d, n)
    scrap_pointer j;
    eight_bits c;
    short k, d, n;
{
  scrap_pointer i, i1;      /∗ pointers into scrap memory ∗/
  j→cat ← c;
  j→trans ← text_ptr;
  j→mathness ← 4 ∗ cur_mathness + init_mathness;
  freeze_text;
  if (k > 1) {
    for (i ← j + k, i1 ← j + 1; i ≤ lo_ptr; i++, i1 ++) {
      i1→cat ← i→cat;
      i1→trans ← i→trans;
      i1→mathness ← i→mathness;
    }
    lo_ptr ← lo_ptr − k + 1;
  }
  ⟨ Change pp to max(scrap_base, pp + d) 163 ⟩;
  ⟨ Print a snapshot of the scrap list if debugging 168 ⟩;
  pp−−;      /∗ we next say pp++ ∗/
}
```

**163.** ⟨ Change $pp$ to $\max(scrap\_base, pp + d)$ 163 ⟩ ≡

```
if (pp + d ≥ scrap_base) pp ← pp + d;
else pp ← scrap_base;
```

This code is used in sections 162 and 164

**164.** Here's the *squash* procedure, which takes advantage of the simplification that occurs when $k \equiv 1$.

```
void squash(j, k, c, d, n)
    scrap_pointer j;
    eight_bits c;
    short k, d, n;
{
  scrap_pointer i;      /* pointers into scrap memory */
  if (k ≡ 1) {
    j→cat ← c;
    ⟨ Change pp to max(scrap_base, pp + d) 163 ⟩;
    ⟨ Print a snapshot of the scrap list if debugging 168 ⟩;
    pp−−;      /* we next say pp++ */
    return;
  }
  for (i ← j; i < j + k; i++) big_app1(i);
  reduce(j, k, c, d, n);
}
```

**165.** 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 165 ⟩ ≡

```
while (1) {
  ⟨ Make sure the entries pp through pp + 3 of cat are defined 166 ⟩;
  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 110 ⟩;
}
```

This code is used in section 169

**166.** If we get to the end of the scrap list, category codes equal to zero are stored, since zero does not match anything in a production.

```
⟨ Make sure the entries pp through pp + 3 of cat are defined 166 ⟩ ≡
  if (lo_ptr < pp + 3) {
    while (hi_ptr ≤ scrap_ptr ∧ lo_ptr ≠ pp + 3) {
      (++lo_ptr)→cat ← hi_ptr→cat;
      lo_ptr→mathness ← (hi_ptr)→mathness;
      lo_ptr→trans ← (hi_ptr++)→trans;
    }
    for (i ← lo_ptr + 1; i ≤ pp + 3; i++) i→cat ← 0;
  }
```

This code is used in section 165

**167.** If CWEAVE is being run in debugging mode, the production numbers and current stack categories will be printed out when *tracing* is set to 2; a sequence of two or more irreducible scraps will be printed out when *tracing* is set to 1.

```
⟨ Global variables 17 ⟩ +≡
  int tracing;      /* can be used to show parsing details */
```

**168.**

```
⟨ Print a snapshot of the scrap list if debugging 168 ⟩ ≡
  {
    scrap_pointer k;      /* pointer into scrap_info */
    if (tracing ≡ 2) {
      printf("\n%d:", n);
      for (k ← scrap_base; k ≤ lo_ptr; k++) {
        if (k ≡ pp) putxchar('*');
        else putxchar('␣');
        if (k→mathness % 4 ≡ yes_math) putchar('+');
        else if (k→mathness % 4 ≡ no_math) putchar('-');
        print_cat(k→cat);
        if (k→mathness/4 ≡ yes_math) putchar('+');
        else if (k→mathness/4 ≡ no_math) putchar('-');
      }
      if (hi_ptr ≤ scrap_ptr) printf("..."); /* indicate that more is coming */
    }
  }
```

This code is used in sections 162 and 164

**169.** The *translate* function assumes that scraps have been stored in positions *scrap_base* through *scrap_ptr* of *cat* and *trans*. It applies productions as much as possible. The result is a token list containing the translation of the given sequence of scraps.

After calling *translate*, we will have $text\_ptr + 3 \le max\_texts$ and $tok\_ptr + 6 \le max\_toks$, so it will be possible to create up to three token lists with up to six tokens without checking for overflow. Before calling *translate*, we should have $text\_ptr < max\_texts$ and $scrap\_ptr < max\_scraps$, since *translate* might add a new text and a new scrap before it checks for overflow.

```
text_pointer translate( )    /* converts a sequence of scraps */
{
  scrap_pointer i,    /* index into cat */
  j;    /* runs through final scraps */
  pp ← scrap_base;
  lo_ptr ← pp − 1;
  hi_ptr ← pp;
  ⟨ If tracing, print an indication of where we are 172 ⟩;
  ⟨ Reduce the scraps using the productions until no more rules apply 165 ⟩;
  ⟨ Combine the irreducible scraps that remain 170 ⟩;
}
```

**170.** 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 170 ⟩ ≡
{
  ⟨ If semi-tracing, show the irreducible scraps 171 ⟩;
  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 169

**171.**

```
⟨ If semi-tracing, show the irreducible scraps 171 ⟩ ≡
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 170

**172.** ⟨ If tracing, print an indication of where we are 172 ⟩ ≡

```
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 169

**173. Initializing the scraps.** If we are going to use the powerful production mechanism just developed, we must get the scraps set up in the first place, given a C text. A table of the initial scraps corresponding to C tokens appeared above in the section on parsing; our goal now is to implement that table. We shall do this by implementing a subroutine called *C_parse* that is analogous to the *C_xref* routine used during phase one.

Like *C_xref*, the *C_parse* procedure starts with the current value of *next_control* and it uses the operation *next_control* ← *get_next*( ) repeatedly to read C text until encountering the next '|' or '/*', or until *next_control* ≥ *format_code*. The scraps corresponding to what it reads are appended into the *cat* and *trans* arrays, and *scrap_ptr* is advanced.

```
void C_parse(spec_ctrl)      /* creates scraps from C tokens */
    eight_bits spec_ctrl;
{
  int count;      /* characters remaining before string break */
  while (next_control < format_code ∨ next_control ≡ spec_ctrl) {
    ⟨ Append the scrap appropriate to next_control 175 ⟩;
    next_control ← get_next( );
    if (next_control ≡ '|' ∨ next_control ≡ begin_comment ∨ next_control ≡ begin_short_comment)
      return;
  }
}
```

**174.** The following macro is used to append a scrap whose tokens have just been appended:

```
#define app_scrap(c, b)
        {
          (++scrap_ptr)→cat ← (c);
          scrap_ptr→trans ← text_ptr;
          scrap_ptr→mathness ← 5 ∗ (b);      /* no no, yes yes, or maybe maybe */
          freeze_text;
        }
```

**175.** ⟨ Append the scrap appropriate to *next_control* 175 ⟩ ≡
⟨ Make sure that there is room for the new scraps, tokens, and texts 176 ⟩;
**switch** (*next_control*) {
**case** *section_name*: *app*(*section_flag* + (**int**) (*cur_section* − *name_dir*));
*app_scrap*(*section_scrap*, *maybe_math*);
*app_scrap*(*exp*, *yes_math*); **break**;
**case** *string*: **case** *constant*: **case** *verbatim*: ⟨ Append a string or constant 178 ⟩; **break**;
**case** *identifier*: *app_cur_id*(1); **break**;
**case** *TEX_string* : ⟨ Append a TEX string, without forming a scrap 179 ⟩; **break**;
**case** '/': **case** '.': *app*(*next_control*);
*app_scrap*(*binop*, *yes_math*); **break**;
**case** '<': *app_str*("\\langle"); *app_scrap*(*prelangle*, *yes_math*); **break**;
**case** '>': *app_str*("\\rangle"); *app_scrap*(*prerangle*, *yes_math*); **break**;
**case** '=': *app_str*("\\K");
*app_scrap*(*binop*, *yes_math*); **break**;
**case** '|': *app_str*("\\OR");
*app_scrap*(*binop*, *yes_math*); **break**;
**case** '^': *app_str*("\\XOR");
*app_scrap*(*binop*, *yes_math*); **break**;
**case** '%': *app_str*("\\MOD");
*app_scrap*(*binop*, *yes_math*); **break**;
**case** '!': *app_str*("\\R");
*app_scrap*(*unop*, *yes_math*); **break**;
**case** '~': *app_str*("\\CM");
*app_scrap*(*unop*, *yes_math*); **break**;
**case** '+': **case** '-': *app*(*next_control*);
*app_scrap*(*unorbinop*, *yes_math*); **break**;
**case** '*': *app*(*next_control*);
*app_scrap*(*raw_unorbin*, *yes_math*); **break**;
**case** '&': *app_str*("\\AND");
*app_scrap*(*raw_unorbin*, *yes_math*); **break**;
**case** '?': *app_str*("\\?");
*app_scrap*(*question*, *yes_math*); **break**;
**case** '#': *app_str*("\\#");
*app_scrap*(*unorbinop*, *yes_math*); **break**;
**case** *ignore*: **case** *xref_roman*: **case** *xref_wildcard*: **case** *xref_typewriter*: **case** *noop*: **break**;
**case** '(': **case** '[': *app*(*next_control*);
*app_scrap*(*lpar*, *maybe_math*); **break**;
**case** ')': **case** ']': *app*(*next_control*);
*app_scrap*(*raw_rpar*, *maybe_math*); **break**;
**case** '{': *app_str*("\\{");
*app_scrap*(*lbrace*, *yes_math*); **break**;
**case** '}': *app_str*("\\}");
*app_scrap*(*rbrace*, *yes_math*); **break**;
**case** ',': *app*(',');
*app_scrap*(*comma*, *yes_math*); **break**;
**case** ';': *app*(';');
*app_scrap*(*semi*, *maybe_math*); **break**;
**case** ':': *app*(':');
*app_scrap*(*colon*, *maybe_math*); **break**;
⟨ Cases involving nonstandard characters 177 ⟩
**case** *thin_space*: *app_str*("\\,");
*app_scrap*(*insert*, *maybe_math*); **break**;
**case** *math_break*: *app*(*opt*);
*app_str*("0");

```
    app_scrap(insert, maybe_math); break;
  case line_break: app(force);
    app_scrap(insert, no_math); break;
  case left_preproc: app(force);
    app(preproc_line);
    app_str("\\#");
    app_scrap(lproc, no_math); break;
  case right_preproc: app(force);
    app_scrap(rproc, no_math); break;
  case big_line_break: app(big_force);
    app_scrap(insert, no_math); break;
  case no_line_break: app(big_cancel);
    app(noop);
    app(break_space);
    app(noop);
    app(big_cancel);
    app_scrap(insert, no_math); break;
  case pseudo_semi: app_scrap(semi, maybe_math); break;
  case macro_arg_open: app_scrap(begin_arg, maybe_math); break;
  case macro_arg_close: app_scrap(end_arg, maybe_math); break;
  case join: app_str("\\J");
    app_scrap(insert, no_math); break;
  case output_defs_code: app(force);
    app_str("\\ATH");
    app(force);
    app_scrap(insert, no_math); break;
  default: app(inserted);
    app(next_control);
    app_scrap(insert, maybe_math); break;
  }
```

This code is used in section 173

**176.** ⟨ Make sure that there is room for the new scraps, tokens, and texts 176 ⟩ ≡

```
  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 175 and 183

**177.** Some nonstandard characters may have entered CWEAVE by means of standard ones. They are converted to TEX control sequences so that it is possible to keep CWEAVE from outputting unusual **char** codes.

⟨ Cases involving nonstandard characters 177 ⟩ ≡
**case** *not_eq*: *app_str*("\\I"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *lt_eq*: *app_str*("\\Z"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *gt_eq*: *app_str*("\\G"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *eq_eq*: *app_str*("\\E"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *and_and*: *app_str*("\\W"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *or_or*: *app_str*("\\V"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *plus_plus*: *app_str*("\\PP"); *app_scrap*(*unop*, *yes_math*); **break**;
**case** *minus_minus*: *app_str*("\\MM"); *app_scrap*(*unop*, *yes_math*); **break**;
**case** *minus_gt*: *app_str*("\\MG"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *gt_gt*: *app_str*("\\GG"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *lt_lt*: *app_str*("\\LL"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *dot_dot_dot*: *app_str*("\\,\\ldots\\,"); *app_scrap*(*exp*, *yes_math*); **break**;
**case** *colon_colon*: *app_str*("\\DC"); *app_scrap*(*colcol*, *maybe_math*); **break**;
**case** *period_ast*: *app_str*("\\PA"); *app_scrap*(*binop*, *yes_math*); **break**;
**case** *minus_gt_ast*: *app_str*("\\MGA"); *app_scrap*(*binop*, *yes_math*); **break**;

This code is used in section 175

**178.** 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 178 ⟩ ≡
  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 175

**179.** We do not make the TEX string into a scrap, because there is no telling what the user will be putting into it; instead we leave it open, to be picked up by next scrap. If it comes at the end of a section, it will be made into a scrap when *finish_C* is called.

```
⟨ Append a TEX string, without forming a scrap 179 ⟩ ≡
  app_str("\\hbox{");
  while (id_first < id_loc)
    if ((eight_bits) (*id_first) > °177) {
      app_tok(quoted_char);
      app_tok((eight_bits) (*id_first++));
    }
    else {
      if (*id_first ≡ '@') id_first++;
      app_tok(*id_first++);
    }
  app('}');
```

This code is used in section 175

**180.** The function *app_cur_id* appends the current identifier to the token list; it also builds a new scrap if *scrapping* ≡ 1.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void app_cur_id( );
```

**181.**
```
  void app_cur_id(scrapping)
      boolean scrapping;      /* are we making this into a scrap? */
  {
    name_pointer p ← id_lookup(id_first, id_loc, normal);
    if (p→ilk ≤ quoted) {      /* not a reserved word */
      app(id_flag + (int) (p − name_dir));
      if (scrapping) app_scrap(exp, p→ilk ≥ custom ? yes_math : maybe_math);
    }
    else {
      app(res_flag + (int) (p − name_dir));
      if (scrapping) app_scrap(p→ilk, maybe_math);
    }
  }
```

**182.** 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.

```
  text_pointer C_translate( )
  {
    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);
  }
```

**183.** The *outer_parse* routine is to *C_parse* as *outer_xref* is to *C_xref*: it constructs a sequence of scraps for C text until *next_control* $\geq$ *format_code*. Thus, it takes care of embedded comments.

```
void outer_parse( )      /* makes scraps from C tokens and comments */
{
  int bal;      /* brace level in comment */
  text_pointer p, q;      /* partial comments */
  while (next_control < format_code)
    if (next_control ≠ begin_comment ∧ next_control ≠ begin_short_comment) C_parse(ignore);
    else {
      boolean is_long_comment ← (next_control ≡ begin_comment);
      ⟨ Make sure that there is room for the new scraps, tokens, and texts 176 ⟩;
      app(cancel);
      app(inserted);
      if (is_long_comment) app_str("\\C{");
      else app_str("\\SHC{");
      bal ← copy_comment(is_long_comment, 1);
      next_control ← ignore;
      while (bal > 0) {
        p ← text_ptr;
        freeze_text;
        q ← C_translate( );      /* at this point we have tok_ptr + 6 ≤ max_toks */
        app(tok_flag + (int) (p − tok_start));
        app_str("\\PB{");
        app(inner_tok_flag + (int) (q − tok_start));
        app_tok('}');
        if (next_control ≡ '|') {
          bal ← copy_comment(is_long_comment, bal);
          next_control ← ignore;
        }
        else bal ← 0;      /* an error has been reported */
      }
      app(force);
      app_scrap(insert, no_math);      /* the full comment becomes a scrap */
    }
}
```

**184. Output of tokens.** So far our programs have only built up multi-layered token lists in CWEAVE's internal memory; we have to figure out how to get them into the desired final form. The job of converting token lists to characters in the TEX output file is not difficult, although it is an implicitly recursive process. Four main considerations had to be kept in mind when this part of CWEAVE was designed. (a) There are two modes of output: *outer* mode, which translates tokens like *force* into line-breaking control sequences, and *inner* mode, which ignores them except that blank spaces take the place of line breaks. (b) The *cancel* instruction applies to adjacent token or tokens that are output, and this cuts across levels of recursion since '*cancel*' occurs at the beginning or end of a token list on one level. (c) The TEX output file will be semi-readable if line breaks are inserted after the result of tokens like *break_space* and *force*. (d) The final line break should be suppressed, and there should be no *force* token output immediately after '\Y\B'.

**185.** The output process uses a stack to keep track of what is going on at different "levels" as the token lists are being written out. Entries on this stack have three parts:

*end_field* is the *tok_mem* location where the token list of a particular level will end;

*tok_field* is the *tok_mem* location from which the next token on a particular level will be read;

*mode_field* is the current mode, either *inner* or *outer*.

The current values of these quantities are referred to quite frequently, so they are stored in a separate place instead of in the *stack* array. We call the current values *cur_end*, *cur_tok*, and *cur_mode*.

The global variable *stack_ptr* tells how many levels of output are currently in progress. The end of output occurs when an *end_translation* token is found, so the stack is never empty except when we first begin the output process.

```
#define inner 0 /* value of mode for C texts within TEX texts */
#define outer 1 /* value of mode for C texts in sections */
⟨ Typedef declarations 18 ⟩ +≡
  typedef int mode;
  typedef struct {
    token_pointer end_field; /* ending location of token list */
    token_pointer tok_field; /* present location within token list */
    boolean mode_field; /* interpretation of control tokens */
  } output_state;
  typedef output_state *stack_pointer;
```

**186.**
```
#define cur_end cur_state.end_field /* current ending location in tok_mem */
#define cur_tok cur_state.tok_field /* location of next output token in tok_mem */
#define cur_mode cur_state.mode_field /* current mode of interpretation */
#define init_stack stack_ptr ← stack; cur_mode ← outer /* initialize the stack */
⟨ Global variables 17 ⟩ +≡
  output_state cur_state; /* cur_end, cur_tok, cur_mode */
  output_state stack[stack_size]; /* info for non-current levels */
  stack_pointer stack_ptr; /* first unused location in the output state stack */
  stack_pointer stack_end ← stack + stack_size − 1; /* end of stack */
  stack_pointer max_stack_ptr; /* largest value assumed by stack_ptr */
```

**187.**
```
⟨ Set initial values 20 ⟩ +≡
  max_stack_ptr ← stack;
```

**188.** To insert token-list $p$ into the output, the *push_level* subroutine is called; it saves the old level of output and gets a new one going. The value of *cur_mode* is not changed.

```
void push_level(p)      /* 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);
}
```

**189.** Conversely, the *pop_level* routine restores the conditions that were in force when the current level was begun. This subroutine will never be called when $stack\_ptr \equiv 1$.

```
void pop_level( )
{
  cur_end ← (--stack_ptr)→end_field;
  cur_tok ← stack_ptr→tok_field;
  cur_mode ← stack_ptr→mode_field;
}
```

**190.** The *get_output* function returns the next byte of output that is not a reference to a token list. It returns the values *identifier* or *res_word* or *section_code* if the next token is to be an identifier (typeset in italics), a reserved word (typeset in boldface) or a section name (typeset by a complex routine that might generate additional levels of output). In these cases *cur_name* points to the identifier or section name in question.

⟨ Global variables 17 ⟩ +≡

```
name_pointer cur_name;
```

**191.** **#define** *res_word* °201 /* returned by *get_output* for reserved words */
**#define** *section_code* °200 /* returned by *get_output* for section names */

```
eight_bits get_output( )      /* returns the next token of output */
{
  sixteen_bits a;      /* current item read from tok_mem */
restart:
  while (cur_tok ≡ cur_end) pop_level( );
  a ← *(cur_tok++);
  if (a ≥ °400) {
    cur_name ← a % id_flag + name_dir;
    switch (a/id_flag) {
    case 2: return (res_word);      /* a ≡ res_flag + cur_name */
    case 3: return (section_code);      /* a ≡ section_flag + cur_name */
    case 4: push_level(a % id_flag + tok_start);
      goto restart;      /* a ≡ tok_flag + cur_name */
    case 5: push_level(a % id_flag + tok_start);
      cur_mode ← inner;
      goto restart;      /* a ≡ inner_tok_flag + cur_name */
    default: return (identifier);      /* a ≡ id_flag + cur_name */
    }
  }
  return (a);
}
```

**192.** The real work associated with token output is done by *make_output*. This procedure appends an *end_translation* token to the current token list, and then it repeatedly calls *get_output* and feeds characters to the output buffer until reaching the *end_translation* sentinel. It is possible for *make_output* to be called recursively, since a section name may include embedded C text; however, the depth of recursion never exceeds one level, since section names cannot be inside of section names.

A procedure called *output_C* does the scanning, translation, and output of C text within '| ... |' brackets, and this procedure uses *make_output* to output the current token list. Thus, the recursive call of *make_output* actually occurs when *make_output* calls *output_C* while outputting the name of a section.

The token list created from within '| ... |' brackets is output as an argument to \PB. Although cwebmac ignores \PB, other macro packages might use it to localize the special meaning of the macros that mark up program text.

```
void output_C( )      /* outputs the current token list */
{
  token_pointer save_tok_ptr;
  text_pointer save_text_ptr;
  sixteen_bits save_next_control;      /* values to be restored */
  text_pointer p;      /* translation of the C text */
  save_tok_ptr ← tok_ptr;
  save_text_ptr ← text_ptr;
  save_next_control ← next_control;
  next_control ← ignore;
  p ← C_translate( );
  app(inner_tok_flag + (int) (p − tok_start));
  out_str("\\PB{");
  make_output( );
  out('}');      /* output the list */
  if (text_ptr > max_text_ptr) max_text_ptr ← text_ptr;
  if (tok_ptr > max_tok_ptr) max_tok_ptr ← tok_ptr;
  text_ptr ← save_text_ptr;
  tok_ptr ← save_tok_ptr;      /* forget the tokens */
  next_control ← save_next_control;      /* restore next_control to original state */
}
```

**193.** Here is CWEAVE's major output handler.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void make_output( );
```

```
194. void make_output( )      /* outputs the equivalents of tokens */
  {
    eight_bits a,      /* current output byte */
    b;      /* next output byte */
    int c;      /* count of indent and outdent tokens */
    char *k, *k_limit;      /* indices into byte_mem */
    char *j;      /* index into buffer */
    char delim;      /* first and last character of string being copied */
    char *save_loc, *save_limit;      /* loc and limit to be restored */
    char scratch[longest_name];      /* scratch area for section names */
    name_pointer cur_section_name;      /* name of section being output */
    boolean save_mode;      /* value of cur_mode before a sequence of breaks */
    app(end_translation);      /* append a sentinel */
    freeze_text;
    push_level(text_ptr − 1);
    while (1) {
      a ← get_output( );
    reswitch:
      switch (a) {
      case end_translation: return;
      case identifier: case res_word: ⟨ Output an identifier 195 ⟩;
        break;
      case section_code: ⟨ Output a section name 199 ⟩;
        break;
      case math_rel: out_str("\\MRL{");
      case noop: case inserted: break;
      case cancel: case big_cancel: c ← 0;
        b ← a;
        while (1) {
          a ← get_output( );
          if (a ≡ inserted) continue;
          if ((a < indent ∧ ¬(b ≡ big_cancel ∧ a ≡ ' ')) ∨ a > big_force) break;
          if (a ≡ indent) c++;
          else if (a ≡ outdent) c−−;
          else if (a ≡ opt) a ← get_output( );
        }
        ⟨ Output saved indent or outdent tokens 198 ⟩;
        goto reswitch;
      case indent: case outdent: case opt: case backup: case break_space: case force:
        case big_force: case preproc_line:
        ⟨ Output a control, look ahead in case of line breaks, possibly goto reswitch 196 ⟩;
        break;
      case quoted_char: out(*(cur_tok++));
        break;
      default: out(a);      /* otherwise a is an ordinary character */
      }
    }
  }
```

**195.** An identifier of length one does not have to be enclosed in braces, and it looks slightly better if set in a math-italic font instead of a (slightly narrower) text-italic font. Thus we output '\|a' but '\\{aa}'.

```
⟨ Output an identifier 195 ⟩ ≡
  out('\\'); if (a ≡ identifier) {
  if (cur_name→ilk ≥ custom ∧ cur_name→ilk ≤ quoted ∧ ¬doing_format) {
    for (j ← cur_name→byte_start; j < (cur_name + 1)→byte_start; j++) out(isxalpha(*j) ? 'x' : *j);
    break;
  }
  else if (is_tiny(cur_name)) out('|')
  else {
    delim ← '.';
    for (j ← cur_name→byte_start; j < (cur_name + 1)→byte_start; j++)
      if (xislower(*j)) {      /* not entirely uppercase */
        delim ← '\\';
        break;
      }
    out(delim);
  }
  } else out('&')      /* a ≡ res_word */
  if (is_tiny(cur_name)) {
    if (isxalpha((cur_name→byte_start)[0])) out('\\');
    out((cur_name→byte_start)[0]);
  }
  else out_name(cur_name);
```

This code is used in section 194

**196.** The current mode does not affect the behavior of CWEAVE's output routine except when we are outputting control tokens.

```
⟨ Output a control, look ahead in case of line breaks, possibly goto reswitch 196 ⟩ ≡
  if (a < break_space ∨ a ≡ preproc_line) {
    if (cur_mode ≡ outer) {
      out('\\');
      out(a − cancel + '0');
      if (a ≡ opt) {
        b ← get_output( );      /* opt is followed by a digit */
        if (b ≠ '0' ∨ force_lines ≡ 0) out(b)
        else out_str("{-1}");      /* force_lines encourages more @| breaks */
      }
    }
    else if (a ≡ opt) b ← get_output( );      /* ignore digit following opt */
  }
  else ⟨ Look ahead for strongest line break, goto reswitch 197 ⟩
```

This code is used in section 194

**197.** If several of the tokens *break_space*, *force*, *big_force* occur in a row, possibly mixed with blank spaces (which are ignored), the largest one is used. A line break also occurs in the output file, except at the very end of the translation. The very first line break is suppressed (i.e., a line break that follows '\Y\B').

```
⟨ Look ahead for strongest line break, goto reswitch 197 ⟩ ≡
  {
    b ← a;
    save_mode ← cur_mode;
    c ← 0;
    while (1) {
      a ← get_output( );
      if (a ≡ inserted) continue;
      if (a ≡ cancel ∨ a ≡ big_cancel) {
        ⟨ Output saved indent or outdent tokens 198 ⟩;
        goto reswitch;      /* cancel overrides everything */
      }
      if ((a ≠ '␣' ∧ a < indent) ∨ a ≡ backup ∨ a > big_force) {
        if (save_mode ≡ outer) {
          if (out_ptr > out_buf + 3 ∧ strncmp(out_ptr − 3, "\\Y\\B", 4) ≡ 0) goto reswitch;
          ⟨ Output saved indent or outdent tokens 198 ⟩;
          out('\\');
          out(b − cancel + '0');
          if (a ≠ end_translation) finish_line( );
        }
        else if (a ≠ end_translation ∧ cur_mode ≡ inner) out('␣');
        goto reswitch;
      }
      if (a ≡ indent) c++;
      else if (a ≡ outdent) c−−;
      else if (a ≡ opt) a ← get_output( );
      else if (a > b) b ← a;      /* if a ≡ '␣' we have a < b */
    }
  }
```

This code is used in section 196

**198.** ⟨ Output saved *indent* or *outdent* tokens 198 ⟩ ≡

```
  for ( ; c > 0; c−−) out_str("\\1");
  for ( ; c < 0; c++) out_str("\\2");
```

This code is used in sections 194 and 197

**199.** The remaining part of *make_output* is somewhat more complicated. When we output a section name, we may need to enter the parsing and translation routines, since the name may contain C code embedded in | ... | constructions. This C code is placed at the end of the active input buffer and the translation process uses the end of the active *tok_mem* area.

```
⟨ Output a section name 199 ⟩ ≡
  {
    out_str("\\X");
    cur_xref ← (xref_pointer) cur_name→xref;
    if (cur_xref→num ≡ file_flag) {
      an_output ← 1;
      cur_xref ← cur_xref→xlink;
    }
    else an_output ← 0;
    if (cur_xref→num ≥ def_flag) {
      out_section(cur_xref→num − def_flag);
      if (phase ≡ 3) {
        cur_xref ← cur_xref→xlink;
        while (cur_xref→num ≥ def_flag) {
          out_str(",␣");
          out_section(cur_xref→num − def_flag);
          cur_xref ← cur_xref→xlink;
        }
      }
    }
    else out('0');      /* output the section number, or zero if it was undefined */
    out(':');
    if (an_output) out_str("\\.{");
    ⟨ Output the text of the section name 200 ⟩;
    if (an_output) out_str("␣}");
    out_str("\\X");
  }
```

This code is used in section 194

**200.** ⟨ Output the text of the section name 200 ⟩ ≡

```
sprint_section_name(scratch, cur_name);
k ← scratch;
k_limit ← scratch + strlen(scratch);
cur_section_name ← cur_name; while (k < k_limit) { b ← *(k++);
if (b ≡ '@') ⟨ Skip next character, give error if not '@' 201 ⟩;
if (an_output)
  switch (b) {
  case ' ': case '\\': case '#': case '%': case '$': case '^': case '{': case '}': case '~':
    case '&': case '_': out('\\');      /* falls through */
  default: out(b);
  }
else if (b ≠ '|') out(b)
else {
  ⟨ Copy the C text into the buffer array 202 ⟩;
  save_loc ← loc;
  save_limit ← limit;
  loc ← limit + 2;
  limit ← j + 1;
  *limit ← '|';
  output_C( );
  loc ← save_loc;
  limit ← save_limit;
}
}
```

This code is used in section 199

**201.** ⟨ Skip next character, give error if not '@' 201 ⟩ ≡

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

This code is used in section 200

**202.** 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 202 ⟩ ≡
  j ← limit + 1;
  *j ← '|';
  delim ← 0; while (1) {
  if (k ≥ k_limit) {
    printf("\n!␣C␣text␣in␣section␣name␣didn't␣end:␣<");
    print_section_name(cur_section_name);
    printf(">␣");
    mark_error;
    break;
  }
  b ← *(k++); if (b ≡ '@') ⟨ Copy a control code into the buffer 203 ⟩
  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 200

**203.**

```
⟨ Copy a control code into the buffer 203 ⟩ ≡
  {
    if (j > buffer + long_buf_size − 4) overflow("buffer");
    *(++j) ← '@';
    *(++j) ← *(k++);
  }
```

This code is used in section 202

**204. 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.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void phase_two( );
```

**205.**

```
void phase_two( )
{
  reset_input( );
  if (show_progress) printf("\nWriting␣the␣output␣file...");
  section_count ← 0;
  format_visible ← 1;
  copy_limbo( );
  finish_line( );
  flush_buffer(out_buf, 0, 0);      /* insert a blank line, it looks nice */
  while (¬input_has_ended) ⟨ Translate the current section 207 ⟩;
}
```

**206.** The output file will contain the control sequence \Y between non-null sections of a section, e.g., between the TEX and definition parts if both are nonempty. This puts a little white space between the parts when they are printed. However, we don't want \Y to occur between two definitions within a single section. The variables *out_line* or *out_ptr* will change if a section is non-null, so the following macros '*save_position*' and '*emit_space_if_needed*' are able to handle the situation:

```
#define save_position  save_line ← out_line; save_place ← out_ptr
#define emit_space_if_needed
        if (save_line ≠ out_line ∨ save_place ≠ out_ptr) out_str("\\Y");
        space_checked ← 1
⟨ Global variables 17 ⟩ +≡
  int save_line;      /* former value of out_line */
  char *save_place;      /* former value of out_ptr */
  int sec_depth;      /* the integer, if any, following @* */
  boolean space_checked;      /* have we done emit_space_if_needed? */
  boolean format_visible;      /* should the next format declaration be output? */
  boolean doing_format ← 0;      /* are we outputting a format declaration? */
  boolean group_found ← 0;      /* has a starred section occurred? */
```

**207.**

```
⟨ Translate the current section 207 ⟩ ≡
{
  section_count++;
  ⟨ Output the code for the beginning of a new section 208 ⟩;
  save_position;
  ⟨ Translate the TEX part of the current section 209 ⟩;
  ⟨ Translate the definition part of the current section 210 ⟩;
  ⟨ Translate the C part of the current section 216 ⟩;
  ⟨ Show cross-references to this section 219 ⟩;
  ⟨ Output the code for the end of a section 223 ⟩;
}
```

This code is used in section 205

**208.** Sections beginning with the CWEB control sequence '@␣' start in the output with the TEX control sequence '\M', followed by the section number. Similarly, '@*' sections lead to the control sequence '\N'. In this case there's an additional parameter, representing one plus the specified depth, immediately after the \N. If the section has changed, we put * just after the section number.

```
⟨ Output the code for the beginning of a new section 208 ⟩ ≡
  if (*(loc − 1) ≠ '*') out_str("\\M");
  else {
    while (*loc ≡ '␣') loc++;
    if (*loc ≡ '*') {      /* "top" level */
      sec_depth ← −1;
      loc++;
    }
    else {
      for (sec_depth ← 0; xisdigit(*loc); loc++) sec_depth ← sec_depth * 10 + (*loc) − '0';
    }
    while (*loc ≡ '␣') loc++;      /* remove spaces before group title */
    group_found ← 1;
    out_str("\\N");
    { char s[32]; sprintf(s, "{%d}", sec_depth + 1); out_str(s); }
    if (show_progress) printf("*%d", section_count);
    update_terminal;      /* print a progress report */
  }
  out_str("{");
  out_section(section_count);
  out_str("}");
```

This code is used in section 207

**209.** 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 209 ⟩ ≡
  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 207

**210.** When we get to the following code we have *next_control* $\geq$ *format_code*, and the token memory is in its initial empty state.

```
⟨ Translate the definition part of the current section 210 ⟩ ≡
  space_checked ← 0;
  while (next_control ≤ definition) {      /* format_code or definition */
    init_stack;
    if (next_control ≡ definition) ⟨ Start a macro definition 213 ⟩
    else ⟨ Start a format definition 214 ⟩;
    outer_parse( );
    finish_C(format_visible);
    format_visible ← 1;
    doing_format ← 0;
  }
```

This code is used in section 207

**211.** The *finish_C* procedure outputs the translation of the current scraps, preceded by the control sequence '`\B`' and followed by the control sequence '`\par`'. It also restores the token and scrap memories to their initial empty state.

A *force* token is appended to the current scraps before translation takes place, so that the translation will normally end with `\6` or `\7` (the TEX macros for *force* and *big_force*). This `\6` or `\7` is replaced by the concluding `\par` or by `\Y\par`.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void finish_C( );
```

**212.**

```
void finish_C(visible)      /* finishes a definition or a C part */
    boolean visible;      /* nonzero if we should produce TEX output */
{
  text_pointer p;      /* translation of the scraps */
  if (visible) {
    out_str("\\B");
    app_tok(force);
    app_scrap(insert, no_math);
    p ← translate( );
    app(tok_flag + (int) (p − tok_start));
    make_output( );      /* output the list */
    if (out_ptr > out_buf + 1)
      if (*(out_ptr − 1) ≡ '\\')
        if (*out_ptr ≡ '6') out_ptr −= 2;
        else if (*out_ptr ≡ '7') *out_ptr ← 'Y';
    out_str("\\par");
    finish_line( );
  }
  if (text_ptr > max_text_ptr) max_text_ptr ← text_ptr;
  if (tok_ptr > max_tok_ptr) max_tok_ptr ← tok_ptr;
  if (scrap_ptr > max_scr_ptr) max_scr_ptr ← scrap_ptr;
  tok_ptr ← tok_mem + 1;
  text_ptr ← tok_start + 1;
  scrap_ptr ← scrap_info;      /* forget the tokens and the scraps */
}
```

**213.** 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 213 ⟩ ≡
  {
    if (save_line ≠ out_line ∨ save_place ≠ out_ptr) 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('$');
      app_cur_id(0);
      if (*loc ≡ '(')
      reswitch:
        switch (next_control ← get_next()) {
        case '(': case ',': app(next_control);
          goto reswitch;
        case identifier: app_cur_id(0);
          goto reswitch;
        case ')': app(next_control);
          next_control ← get_next();
          break;
        default: err_print("! Improper macro definition");
          break;
        }
      else next_control ← get_next();
      app_str("$ ");
      app(break_space);
      app_scrap(dead, no_math);      /* scrap won't take part in the parsing */
    }
  }
```

This code is used in section 210

**214.** ⟨ Start a format definition 214 ⟩ ≡

```
{
  doing_format ← 1;
  if (∗(loc − 1) ≡ 's' ∨ ∗(loc − 1) ≡ 'S') format_visible ← 0;
  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 210

**215.** Finally, when the TEX and definition parts have been treated, we have *next_control* ≥ *begin_C*. We will make the global variable *this_section* point to the current section name, if it has a name.

⟨ Global variables 17 ⟩ +≡

```
name_pointer this_section;      /∗ the current section name, or zero ∗/
```

**216.** ⟨ Translate the C part of the current section 216 ⟩ ≡

```
this_section ← name_dir;
if (next_control ≤ section_name) {
  emit_space_if_needed;
  init_stack;
  if (next_control ≡ begin_C) next_control ← get_next( );
  else {
    this_section ← cur_section;
    ⟨ Check that '=' or '==' follows this section name, and emit the scraps to start the section
        definition 217 ⟩;
  }
  while (next_control ≤ section_name) {
    outer_parse( );
    ⟨ Emit the scrap for a section name if present 218 ⟩;
  }
  finish_C(1);
}
```

This code is used in section 207

**217.** 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 217 ⟩ ≡
  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 216

**218.** ⟨ Emit the scrap for a section name if present 218 ⟩ ≡

```
  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 216

**219.** Cross references relating to a named section are given after the section ends.

```
⟨ Show cross-references to this section 219 ⟩ ≡
  if (this_section > name_dir) {
    cur_xref ← (xref_pointer) this_section→xref;
    if (cur_xref→num ≡ file_flag) {
      an_output ← 1;
      cur_xref ← cur_xref→xlink;
    }
    else an_output ← 0;
    if (cur_xref→num > def_flag) cur_xref ← cur_xref→xlink;      /* bypass current section number */
    footnote(def_flag);
    footnote(cite_flag);
    footnote(0);
  }
```

This code is used in section 207

**220.** The *footnote* procedure gives cross-reference information about multiply defined section names (if the *flag* parameter is *def_flag*), or about references to a section name (if *flag* ≡ *cite_flag*), or to its uses (if *flag* ≡ 0). It assumes that *cur_xref* points to the first cross-reference entry of interest, and it leaves *cur_xref* pointing to the first element not printed. Typical outputs: '`\A101.`'; '`\Us 370\ET1009.`'; '`\As 8, 27\*\ETs64.`'.

Note that the output of `CWEAVE` is not English-specific; users may supply new definitions for the macros `\A`, `\As`, etc.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void footnote( );
```

**221.**
```
void footnote(flag)      /* outputs section cross-references */
    sixteen_bits flag;
{
  xref_pointer q;      /* cross-reference pointer variable */
  if (cur_xref→num ≤ flag) return;
  finish_line( );
  out('\\');
  out(flag ≡ 0 ? 'U' : flag ≡ cite_flag ? 'Q' : 'A');
  ⟨ Output all the section numbers on the reference list cur_xref 222 ⟩;
  out('.');
}
```

**222.** The following code distinguishes three cases, according as the number of cross-references is one, two, or more than two. Variable $q$ points to the first cross-reference, and the last link is a zero.

```
⟨ Output all the section numbers on the reference list cur_xref 222 ⟩ ≡
  q ← cur_xref;
  if (q→xlink→num > flag) out('s');      /* plural */
  while (1) {
    out_section(cur_xref→num − flag);
    cur_xref ← cur_xref→xlink;      /* point to the next cross-reference to output */
    if (cur_xref→num ≤ flag) break;
    if (cur_xref→xlink→num > flag) out_str(",␣");      /* not the last */
    else {
      out_str("\\ET");      /* the last */
      if (cur_xref ≠ q→xlink) out('s');      /* the last of more than two */
    }
  }
```

This code is used in section 221

**223.**
```
⟨ Output the code for the end of a section 223 ⟩ ≡
  out_str("\\fi");
  finish_line( );
  flush_buffer(out_buf, 0, 0);      /* insert a blank line, it looks nice */
```

This code is used in section 207

**224. 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.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void phase_three( );
```

**225.**

```
void phase_three( )
{
  if (no_xref) {
    finish_line( );
    out_str("\\end");
    finish_line( );
  }
  else {
    phase ← 3;
    if (show_progress) printf("\nWriting␣the␣index...");
    finish_line( );
    if ((idx_file ← fopen(idx_file_name, "w")) ≡ Λ)
      fatal("!␣Cannot␣open␣index␣file␣", idx_file_name);
    if (change_exists) {
      ⟨ Tell about changed sections 227 ⟩;
      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 229 ⟩;
    ⟨ Sort and output the index 238 ⟩;
    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, "w")) ≡ 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 247 ⟩;
    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);
  }
  if (show_happiness) printf("\nDone.");
  check_complete( );      /* was all of the change file used? */
}
```

**226.** Just before the index comes a list of all the changed sections, including the index section itself.

```
⟨ Global variables 17 ⟩ +≡
  sixteen_bits k_section;      /* runs through the sections */
```

**227.** ⟨ Tell about changed sections 227 ⟩ ≡

```
{   /* remember that the index is already marked as changed */
  k_section ← 0;
  while (¬changed_section[++k_section]) ;
  out_str("\\ch␣");
  out_section(k_section);
  while (k_section < section_count) {
    while (¬changed_section[++k_section]) ;
    out_str(",␣");
    out_section(k_section);
  }
  out('.');
}
```

This code is used in section 225

**228.** A left-to-right radix sorting method is used, since this makes it easy to adjust the collating sequence and since the running time will be at worst proportional to the total length of all entries in the index. We put the identifiers into 102 different lists based on their first characters. (Uppercase letters are put into the same list as the corresponding lowercase letters, since we want to have '$t < TeX <$ **to**'.) The list for character $c$ begins at location $bucket[c]$ and continues through the *blink* array.

⟨ Global variables 17 ⟩ +≡

```
name_pointer bucket[256];
name_pointer next_name;      /* successor of cur_name when sorting */
name_pointer blink[max_names];      /* links in the buckets */
```

**229.** To begin the sorting, we go through all the hash lists and put each entry having a nonempty cross-reference list into the proper bucket.

⟨ Do the first pass of sorting 229 ⟩ ≡

```
{
  int c;
  for (c ← 0; c ≤ 255; c++) bucket[c] ← Λ;
  for (h ← hash; h ≤ hash_end; h++) {
    next_name ← *h;
    while (next_name) {
      cur_name ← next_name;
      next_name ← cur_name→link;
      if (cur_name→xref ≠ (char *) xmem) {
        c ← (eight_bits) ((cur_name→byte_start)[0]);
        if (xisupper(c)) c ← tolower(c);
        blink[cur_name − name_dir] ← bucket[c];
        bucket[c] ← cur_name;
      }
    }
  }
}
```

This code is used in section 225

**230.** During the sorting phase we shall use the *cat* and *trans* arrays from CWEAVE's parsing algorithm and rename them *depth* and *head*. They now represent a stack of identifier lists for all the index entries that have not yet been output. The variable *sort_ptr* tells how many such lists are present; the lists are output in reverse order (first *sort_ptr*, then *sort_ptr* − 1, etc.). The $j$th list starts at $head[j]$, and if the first $k$ characters of all entries on this list are known to be equal we have $depth[j] \equiv k$.

**231.** ⟨ Rest of *trans_plus* union 231 ⟩ ≡
**name_pointer** *Head*;

This code is used in section 103

**232.** **#define** *depth* *cat* /* reclaims memory that is no longer needed for parsing */
**#define** *head* *trans_plus.Head* /* ditto */
**format** *sort_pointer* *int*
**#define** **sort_pointer** **scrap_pointer** /* ditto */
**#define** *sort_ptr* *scrap_ptr* /* ditto */
**#define** *max_sorts* *max_scraps* /* ditto */
⟨ Global variables 17 ⟩ +≡
**eight_bits** *cur_depth*; /* depth of current buckets */
**char** **cur_byte*; /* index into *byte_mem* */
**sixteen_bits** *cur_val*; /* current cross-reference number */
**sort_pointer** *max_sort_ptr*; /* largest value of *sort_ptr* */

**233.** ⟨ Set initial values 20 ⟩ +≡
*max_sort_ptr* ← *scrap_info*;

**234.** The desired alphabetic order is specified by the *collate* array; namely, *collate*[0] < *collate*[1] << *collate*[100].

⟨ Global variables 17 ⟩ +≡
**eight_bits** *collate*[102 + 128]; /* collation order */

**235.** We use the order null < ␣ < other characters < `_` < `A` = `a` < ··· < `Z` = `z` < `0` < ··· < `9`. Warning: The collation mapping needs to be changed if ASCII code is not being used.

⟨ Set initial values 20 ⟩ +≡
*collate*[0] ← 0;
*strcpy*(*collate* + 1, "␣\1\2\3\4\5\6\7\10\11\12\13\14\15\16\17\20\21\22\23\24\25\2\
6\27\30\31\32\33\34\35\36\37!\42#$%&'()*+,-./:;<=>?@[\\]^‘{|}~_abcdefghijklmnopq\
rstuvwxyz0123456789\200\201\202\203\204\205\206\207\210\211\212\213\214\215\216\\
217\220\221\222\223\224\225\226\227\230\231\232\233\234\235\236\237\240\241\242\\
243\244\245\246\247\250\251\252\253\254\255\256\257\260\261\262\263\264\265\266\\
267\270\271\272\273\274\275\276\277\300\301\302\303\304\305\306\307\310\311\312\\
313\314\315\316\317\320\321\322\323\324\325\326\327\330\331\332\333\334\335\336\\
337\340\341\342\343\344\345\346\347\350\351\352\353\354\355\356\357\360\361\362\\
363\364\365\366\367\370\371\372\373\374\375\376\377");

**236.** 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) */
⟨ Predeclaration of procedures 2 ⟩ +≡
**void** *unbucket*( );

```
237. void unbucket(d)      /* 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;
    }
}
```

**238.** ⟨ Sort and output the index 238 ⟩ ≡

```
sort_ptr ← scrap_info;
unbucket(1);
while (sort_ptr > scrap_info) {
  cur_depth ← sort_ptr→depth;
  if (blink[sort_ptr→head − name_dir] ≡ 0 ∨ cur_depth ≡ infinity)
    ⟨ Output index entries for the list at sort_ptr 240 ⟩
  else ⟨ Split the list at sort_ptr into further lists 239 ⟩;
}
```

This code is used in section 225

**239.** ⟨ Split the list at *sort_ptr* into further lists 239 ⟩ ≡

```
{
  eight_bits c;
  next_name ← sort_ptr→head;
  do {
    cur_name ← next_name;
    next_name ← blink[cur_name − name_dir];
    cur_byte ← cur_name→byte_start + cur_depth;
    if (cur_byte ≡ (cur_name + 1)→byte_start) c ← 0;      /* hit end of the name */
    else {
      c ← (eight_bits) *cur_byte;
      if (xisupper(c)) c ← tolower(c);
    }
    blink[cur_name − name_dir] ← bucket[c];
    bucket[c] ← cur_name;
  } while (next_name);
  −−sort_ptr;
  unbucket(cur_depth + 1);
}
```

This code is used in section 238

**240.** ⟨ Output index entries for the list at *sort_ptr* 240 ⟩ ≡

```
{
  cur_name ← sort_ptr→head;
  do {
    out_str("\\I");
    ⟨ Output the name at cur_name 241 ⟩;
    ⟨ Output the cross-references at cur_name 242 ⟩;
    cur_name ← blink[cur_name − name_dir];
  } while (cur_name);
  −− sort_ptr;
}
```

This code is used in section 238

**241.** ⟨ Output the name at *cur_name* 241 ⟩ ≡

```
switch (cur_name→ilk) {
case normal:
  if (is_tiny(cur_name)) out_str("\\|");
  else {
    char *j;
    for (j ← cur_name→byte_start; j < (cur_name + 1)→byte_start; j++)
      if (xislower(*j)) goto lowcase;
    out_str("\\.");
    break;
  lowcase: out_str("\\\\");
  }
  break;
case roman: break;
case wildcard: out_str("\\9");
  break;
case typewriter: out_str("\\.");
  break;
case custom: case quoted:
  {
    char *j;
    out_str("$\\");
    for (j ← cur_name→byte_start; j < (cur_name + 1)→byte_start; j++) out(isxalpha(*j) ? 'x' : *j);
    out('$');
    goto name_done;
  }
default: out_str("\\&");
}
out_name(cur_name); name_done:
```

This code is used in section 240

**242.** Section numbers that are to be underlined are enclosed in '\[...]'.

```
⟨ Output the cross-references at cur_name 242 ⟩ ≡
  ⟨ Invert the cross-reference list at cur_name, making cur_xref the head 244 ⟩;
  do {
    out_str(",␣");
    cur_val ← cur_xref→num;
    if (cur_val < def_flag) out_section(cur_val);
    else {
      out_str("\\[");
      out_section(cur_val − def_flag);
      out(']');
    }
    cur_xref ← cur_xref→xlink;
  } while (cur_xref ≠ xmem);
  out('.');
  finish_line( );
```

This code is used in section 240

**243.** List inversion is best thought of as popping elements off one stack and pushing them onto another. In this case *cur_xref* will be the head of the stack that we push things onto.

```
⟨ Global variables 17 ⟩ +≡
  xref_pointer next_xref, this_xref;      /* pointer variables for rearranging a list */
```

**244.**

```
⟨ Invert the cross-reference list at cur_name, making cur_xref the head 244 ⟩ ≡
  this_xref ← (xref_pointer) cur_name→xref;
  cur_xref ← xmem;
  do {
    next_xref ← this_xref→xlink;
    this_xref→xlink ← cur_xref;
    cur_xref ← this_xref;
    this_xref ← next_xref;
  } while (this_xref ≠ xmem);
```

This code is used in section 242

**245.** The following recursive procedure walks through the tree of section names and prints them.

```
⟨ Predeclaration of procedures 2 ⟩ +≡
  void section_print( );
```

**246.** **void** *section_print*(*p*) /* print all section names in subtree *p* */
**name_pointer** *p*;
{
**if** (*p*) {
*section_print*(*p*→*llink*);
*out_str*("\\I");
*tok_ptr* ← *tok_mem* + 1;
*text_ptr* ← *tok_start* + 1;
*scrap_ptr* ← *scrap_info*;
*init_stack*;
*app*(*p* − *name_dir* + *section_flag*);
*make_output*( );
*footnote*(*cite_flag*);
*footnote*(0); /* *cur_xref* was set by *make_output* */
*finish_line*( );
*section_print*(*p*→*rlink*);
}
}

**247.** ⟨ Output all the section names 247 ⟩ ≡
*section_print*(*root*)
This code is used in section 225

**248.** Because on some systems the difference between two pointers is a **long** rather than an **int**, we use `%ld` to print these quantities.
**void** *print_stats*( )
{
*printf*("\nMemory␣usage␣statistics:\n");
*printf*("%ld␣names␣(out␣of␣%ld)\n", (**long**) (*name_ptr* − *name_dir*), (**long**) *max_names*);
*printf*("%ld␣cross-references␣(out␣of␣%ld)\n", (**long**) (*xref_ptr* − *xmem*), (**long**) *max_refs*);
*printf*("%ld␣bytes␣(out␣of␣%ld)\n", (**long**) (*byte_ptr* − *byte_mem*), (**long**) *max_bytes*);
*printf*("Parsing:\n");
*printf*("%ld␣scraps␣(out␣of␣%ld)\n", (**long**) (*max_scr_ptr* − *scrap_info*), (**long**) *max_scraps*);
*printf*("%ld␣texts␣(out␣of␣%ld)\n", (**long**) (*max_text_ptr* − *tok_start*), (**long**) *max_texts*);
*printf*("%ld␣tokens␣(out␣of␣%ld)\n", (**long**) (*max_tok_ptr* − *tok_mem*), (**long**) *max_toks*);
*printf*("%ld␣levels␣(out␣of␣%ld)\n", (**long**) (*max_stack_ptr* − *stack*), (**long**) *stack_size*);
*printf*("Sorting:\n");
*printf*("%ld␣levels␣(out␣of␣%ld)\n", (**long**) (*max_sort_ptr* − *scrap_info*), (**long**) *max_scraps*);
}

**249. 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.

## Index of cweave



### A

**D**



**E**



**F**

**T**