Common code for CTANGLE and CWEAVE

(Version 4.2)

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 3⟩
⟨Preprocessor definitions⟩
⟨Common code for CWEAVE and CTANGLE 2⟩
⟨Global variables 18⟩
⟨Predeclaration of procedures 7⟩

2.   The details will be filled in due course. The interface of this module is included first. It is also used by the main programs.

First comes general stuff:

#**define**   *ctangle*   *false*
#**define**   *cweave*   *true*

⟨Common code for CWEAVE and CTANGLE 2⟩ ≡
  typedef bool boolean;
  typedef uint8_t eight_bits;
  typedef uint16_t sixteen_bits;
  extern boolean *program*;   ▷ CWEAVE or CTANGLE? ◁
  extern int *phase*;   ▷ which phase are we in? ◁

See also sections 4, 5, 6, 8, 9, 11, 13, and 14.

This code is used in section 1.

3.   Interface to the standard C library:

⟨Include files 3⟩ ≡
#**include** <ctype.h>   ▷ definition of *isalpha*, *isdigit* and so on ◁
#**include** <stdbool.h>   ▷ definition of **bool**, *true* and *false* ◁
#**include** <stddef.h>   ▷ definition of **ptrdiff_t** ◁
#**include** <stdint.h>   ▷ definition of **uint8_t** and **uint16_t** ◁
#**include** <stdlib.h>   ▷ definition of *getenv* and *exit* ◁
#**include** <stdio.h>   ▷ definition of *printf* and friends ◁
#**include** <string.h>   ▷ definition of *strlen*, *strcmp* and so on ◁

This code is used in section 1.

4.   Code related to the character set:

#define *and_and* °*4*   ▷ '`&&`'; corresponds to MIT's ∧ ◁
#define *lt_lt* °*20*   ▷ '`<<`'; corresponds to MIT's ⊂ ◁
#define *gt_gt* °*21*   ▷ '`>>`'; corresponds to MIT's ⊃ ◁
#define *plus_plus* °*13*   ▷ '`++`'; corresponds to MIT's ↑ ◁
#define *minus_minus* °*1*   ▷ '`--`'; corresponds to MIT's ↓ ◁
#define *minus_gt* °*31*   ▷ '`->`'; corresponds to MIT's → ◁
#define *non_eq* °*32*   ▷ '`!=`'; corresponds to MIT's ≠ ◁
#define *lt_eq* °*34*   ▷ '`<=`'; corresponds to MIT's ≤ ◁
#define *gt_eq* °*35*   ▷ '`>=`'; corresponds to MIT's ≥ ◁
#define *eq_eq* °*36*   ▷ '`==`'; corresponds to MIT's ≡ ◁
#define *or_or* °*37*   ▷ '`||`'; corresponds to MIT's ∨ ◁
#define *dot_dot_dot* °*16*   ▷ '`...`'; corresponds to MIT's ∞ ◁
#define *colon_colon* °*6*   ▷ '`::`'; corresponds to MIT's ∈ ◁
#define *period_ast* °*26*   ▷ '`.*`'; corresponds to MIT's ⊗ ◁
#define *minus_gt_ast* °*27*   ▷ '`->*`'; corresponds to MIT's ⇆ ◁

⟨ Common code for CWEAVE and CTANGLE 2 ⟩ +≡
  extern char *section_text*[ ];   ▷ text being sought for ◁
  extern char *∗section_text_end*;   ▷ end of *section_text* ◁
  extern char *∗id_first*;   ▷ where the current identifier begins in the buffer ◁
  extern char *∗id_loc*;   ▷ just after the current identifier in the buffer ◁

5.   Code related to input routines:

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

⟨ Common code for CWEAVE and CTANGLE 2 ⟩ +≡
  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 ◁

6. Code related to file handling:

   format *line*   *x*   ▷ make *line* an unreserved word ◁
#**define** *max_include_depth* 10
          ▷ maximum number of source files open simultaneously, not counting the change file ◁
#**define** *max_file_name_length* 1024
#**define** *cur_file* *file*[*include_depth*]   ▷ current file ◁
#**define** *cur_file_name* *file_name*[*include_depth*]   ▷ current file name ◁
#**define** *cur_line* *line*[*include_depth*]   ▷ number of current line in current file ◁
#**define** *web_file* *file*[0]   ▷ main source file ◁
#**define** *web_file_name* *file_name*[0]   ▷ main source file name ◁
⟨ Common code for CWEAVE and CTANGLE 2 ⟩ +≡
  extern int *include_depth*;   ▷ current level of nesting ◁
  extern FILE *∗file*[ ];   ▷ stack of non-change files ◁
  extern FILE *∗change_file*;   ▷ change file ◁
  extern char *file_name*[ ][*max_file_name_length*];   ▷ stack of non-change file names ◁
  extern char *change_file_name*[ ];   ▷ name of change file ◁
  extern int *line*[ ];   ▷ number of current line in the stacked files ◁
  extern int *change_line*;   ▷ number of current line in change file ◁
  extern int *change_depth*;   ▷ where @y originated during a change ◁
  extern boolean *input_has_ended*;   ▷ if there is no more input ◁
  extern boolean *changing*;   ▷ if the current line is from *change_file* ◁
  extern boolean *web_file_open*;   ▷ if the web file is being read ◁

7. ⟨ Predeclaration of procedures 7 ⟩ ≡
  extern boolean *get_line*(**void**);   ▷ inputs the next line ◁
  extern void *check_complete*(**void**);   ▷ checks that all changes were picked up ◁
  extern void *reset_input*(**void**);   ▷ initialize to read the web file and change file ◁
See also sections 10, 12, 15, 24, 28, 33, 55, 64, and 76.
This code is used in section 1.

8. Code related to section numbers:

⟨ Common code for CWEAVE and CTANGLE 2 ⟩ +≡
  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 ◁

9.   Code related to identifier and section name storage:

#define $length(c)$ (**size_t**)$((c+1)\rightarrow byte\_start - (c)\rightarrow byte\_start)$ ▷ the length of a name ◁
#define $print\_id(c)$ $term\_write((c)\rightarrow byte\_start, length((c)))$ ▷ 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 $root$ $name\_dir\rightarrow rlink$ ▷ the root of the binary search tree for section names ◁

⟨ Common code for CWEAVE and CTANGLE 2 ⟩ +≡
　typedef struct name_info {
　　char $*byte\_start$;   ▷ beginning of the name in $byte\_mem$ ◁
　　struct name_info $*link$;
　　union {
　　　struct name_info $*Rlink$;   ▷ right link in binary search tree for section names ◁
　　　char Ilk;   ▷ used by identifiers in CWEAVE only ◁
　　} $dummy$;
　　void $*equiv\_or\_xref$;   ▷ info corresponding to names ◁
　} **name_info**;   ▷ contains information about an identifier or section name ◁
　typedef name_info $*$**name_pointer**;   ▷ pointer into array of **name_infos** ◁
　typedef name_pointer $*$**hash_pointer**;
　extern char $byte\_mem[\,]$;   ▷ characters of names ◁
　extern char $*byte\_mem\_end$;   ▷ end of $byte\_mem$ ◁
　extern char $*byte\_ptr$;   ▷ first unused position in $byte\_mem$ ◁
　extern name_info $name\_dir[\,]$;   ▷ information about names ◁
　extern name_pointer $name\_dir\_end$;   ▷ end of $name\_dir$ ◁
　extern name_pointer $name\_ptr$;   ▷ first unused position in $name\_dir$ ◁
　extern name_pointer $hash[\,]$;   ▷ heads of hash lists ◁
　extern hash_pointer $hash\_end$;   ▷ end of $hash$ ◁
　extern hash_pointer h;   ▷ index into hash-head array ◁

10.   ⟨ Predeclaration of procedures 7 ⟩ +≡
　extern boolean $names\_match$(**name_pointer**, **const char** $*$, **size_t**, **eight_bits**);
　extern name_pointer id_lookup(**const char** $*$, **const char** $*$, **char**);
　　▷ looks up a string in the identifier table ◁
　extern name_pointer $section\_lookup$(**char** $*$, **char** $*$, **int**);   ▷ finds section name ◁
　extern void $init\_node$(**name_pointer**);
　extern void $init\_p$(**name_pointer**, **eight_bits**);
　extern void $print\_prefix\_name$(**name_pointer**);
　extern void $print\_section\_name$(**name_pointer**);
　extern void $sprint\_section\_name$(**char** $*$, **name_pointer**);

11.   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 2 ⟩ +≡
   extern int *history*;   ▷ indicates how bad this run was ◁

12.   ⟨ Predeclaration of procedures 7 ⟩ +≡
   extern int *wrap_up*(**void**);   ▷ indicate *history* and exit ◁
   extern void *err_print*(**const char** ∗);   ▷ print error message and context ◁
   extern void *fatal*(**const char** ∗, **const char** ∗);   ▷ issue error message and die ◁
   extern void *overflow*(**const char** ∗);   ▷ succumb because a table has overflowed ◁

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_stats* *flags*['s']   ▷ should statistics be printed at end of run? ◁
#define *show_happiness* *flags*['h']   ▷ should lack of errors be announced? ◁
#define *make_xrefs* *flags*['x']   ▷ should cross references be output? ◁
⟨ Common code for CWEAVE and CTANGLE 2 ⟩ +≡
   extern int *argc*;   ▷ copy of *ac* parameter to *main* ◁
   extern char ∗∗*argv*;   ▷ copy of *av* parameter to *main* ◁
   extern char *C_file_name*[ ];   ▷ name of *C_file* ◁
   extern char *tex_file_name*[ ];   ▷ name of *tex_file* ◁
   extern char *idx_file_name*[ ];   ▷ name of *idx_file* ◁
   extern char *scn_file_name*[ ];   ▷ name of *scn_file* ◁
   extern boolean *flags*[ ];   ▷ an option for each 7-bit code ◁

14.   Code related to output:

#define *update_terminal* *fflush*(*stdout*)   ▷ empty the terminal output buffer ◁
#define *new_line* *putchar*('\n')
#define *putxchar* *putchar*
#define *term_write*(*a*, *b*) *fflush*(*stdout*), *fwrite*(*a*, **sizeof**(**char**), *b*, *stdout*)
#define *C_printf*(*c*, *a*) *fprintf*(*C_file*, *c*, *a*)
#define *C_putc*(*c*) *putc*(*c*, *C_file*)   ▷ isn't C wonderfully consistent? ◁
⟨ Common code for CWEAVE and CTANGLE 2 ⟩ +≡
   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:

⟨ Predeclaration of procedures 7 ⟩ +≡
  extern void *common_init*(**void**);
  extern void *print_stats*(**void**);

16.   The following parameters were sufficient in the original WEB to handle TEX, so they should be sufficient for most applications of CWEB.

#define *max_bytes* 1000000   ▷ the number of bytes in identifiers, index entries, and section names ◁
#define *max_toks* 1000000   ▷ number of bytes in compressed C code ◁
#define *max_names* 10239   ▷ number of identifiers, strings, section names; must be less than 10240 ◁
#define *max_sections* 4000   ▷ greater than the total number of sections ◁
#define *max_texts* 10239   ▷ number of replacement texts, must be less than 10240 ◁
#define *longest_name* 10000   ▷ file and section names and section texts shouldn't be longer than this ◁
#define *stack_size* 500   ▷ number of simultaneous levels of macro expansion ◁
#define *buf_size* 1000   ▷ maximum length of input line, plus one ◁
#define *long_buf_size* (*buf_size* + *longest_name*)   ▷ for CWEAVE ◁

17.   End of COMMON interface.

18.   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*.

⟨ Global variables 18 ⟩ ≡
  boolean *program*;   ▷ CWEAVE or CTANGLE? ◁
See also sections 19, 21, 22, 25, 26, 37, 43, 44, 46, 65, 73, and 83.
This code is used in section 1.

19.   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.

⟨ Global variables 18 ⟩ +≡
  int *phase*;   ▷ which phase are we in? ◁

20.   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*(**void**)
  {
   ⟨ Initialize pointers 45 ⟩
   ⟨ Set the default options common to CTANGLE and CWEAVE 74 ⟩
   ⟨ Scan arguments and open output files 84 ⟩
  }

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

A few character pairs are encoded internally as single characters, using the definitions in the interface sections above. These definitions are consistent with an extension of ASCII code originally developed at MIT and explained in Appendix C of *The T_EXbook*; thus, users who have such a character set can type things like ≠ and ∧ 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 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.

⟨ Global variables 18 ⟩ +≡
   char *section_text*[*longest_name* + 1];   ▷ text 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 ◁

22.   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 *true* if the read is successful and *false* 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.

⟨ Global variables 18 ⟩ +≡
 char *buffer*[*long_buf_size*]; ▷ where each line of input goes ◁
 char ∗*buffer_end* ← *buffer* + *buf_size* − 2; ▷ end of *buffer* ◁
 char ∗*loc* ← *buffer*; ▷ points to the next character to be read from the buffer ◁
 char ∗*limit* ← *buffer*; ▷ points to the last character in the buffer ◁

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

```
static boolean input_ln (     ▷ copies a line into buffer or returns false ◁
    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 false;     ▷ 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 false;     ▷ there was nothing after the last newline ◁
  return true;
}
```

24.   ⟨ Predeclaration of procedures 7 ⟩ +≡   **static boolean** *input_ln*(**FILE** ∗);

25.   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.

⟨ Global variables 18 ⟩ +≡
    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 ◁
    static 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* ← *false*;   ▷ if the web file is being read ◁

26.   When *changing* ≡ *false*, 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*, (**size_t**)(*limit* − *buffer*)))

⟨ Global variables 18 ⟩ +≡
    static char *change_buffer*[*buf_size*];   ▷ next line of *change_file* ◁
    static char **change_limit*;   ▷ points to the last character in *change_buffer* ◁

27.   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 *true*; hence error messages will be reported correctly.

    static void *prime_the_change_buffer*(**void**)
    {
        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 29 ⟩
        ⟨ Skip to the next nonblank line; **return** if end of file 30 ⟩
        ⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 31 ⟩
    }

28.   ⟨ Predeclaration of procedures 7 ⟩ +≡ **static void** *prime_the_change_buffer*(**void**);

29.   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 29 ⟩ ≡
   while (*true*) {
     change_line ++;
     if (¬*input_ln*(*change_file*)) **return**;
     if (*limit* < *buffer* + 2) **continue**;
     if (*buffer*[0] ≠ '@') **continue**;
     if (*xisupper*(*buffer*[1])) *buffer*[1] ← *tolower*((**eight_bits**) *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 27.

30.   Here we are looking at lines following the @x.

⟨ Skip to the next nonblank line; **return** if end of file 30 ⟩ ≡
   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 27.

31.   ⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 31 ⟩ ≡
   {
     change_limit ← *change_buffer* + (**ptrdiff_t**)(*limit* − *buffer*);
     strncpy(*change_buffer*, *buffer*, (**size_t**)(*limit* − *buffer* + 1));
   }
This code is used in sections 27 and 32.

32.   The following procedure is used to see if the next change entry should go into effect; it is called only when *changing* is *false*. 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*;
          }

   static void *check_change*(**void**)   ▷ switches to *change_file* if the buffers match ◁
   {
     int *n* ← 0;   ▷ the number of discrepancies found ◁
     if (*lines_dont_match*) **return**;
     change_pending ← *false*;
     if (¬*changed_section*[*section_count*]) {
       if_section_start_make_pending(*true*);
       if (¬*change_pending*) *changed_section*[*section_count*] ← *true*;
     }
     while (*true*) {
       changing ← *true*; *print_where* ← *true*; *change_line*++;
       if (¬*input_ln*(*change_file*)) {
         err_print("! Change file ended before @y"); *change_limit* ← *change_buffer*;
         changing ← *false*; **return**;
       }
       if (*limit* > *buffer* + 1 ∧ *buffer*[0] ≡ '@') {
         char *xyz_code* ← *xisupper*(*buffer*[1]) ? *tolower*((**eight_bits**) *buffer*[1]) : *buffer*[1];
         ⟨If the current line starts with @y, report any discrepancies and **return** 34⟩
       }
       ⟨Move *buffer* and *limit* to *change_buffer* and *change_limit* 31⟩
       changing ← *false*; *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* ← *true*; **return**;
         }
         include_depth−−; *cur_line*++;
       }
       if (*lines_dont_match*) *n*++;
     }
   }

33.   ⟨Predeclaration of procedures 7⟩ +≡ **static void** *check_change*(**void**);

34. ⟨ If the current line starts with @y, report any discrepancies and **return** 34 ⟩ ≡

  if $(xyz\_code \equiv \texttt{'x'} \vee xyz\_code \equiv \texttt{'z'})$ {

    $loc \leftarrow buffer + 2$; $err\_print(\texttt{"!\_Where\_is\_the\_matching\_@y?"})$;

  }

  else if $(xyz\_code \equiv \texttt{'y'})$ {

    if $(n > 0)$ {

      $loc \leftarrow buffer + 2$; $printf(\texttt{"\textbackslash n!\_Hmm...\_\%d\_"}, n)$;

      $err\_print(\texttt{"of\_the\_preceding\_lines\_failed\_to\_match"})$;

    }

    $change\_depth \leftarrow include\_depth$; **return**;

  }

This code is used in section 32.

35. 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*(**void**)

  {

    $limit \leftarrow buffer$; $loc \leftarrow buffer + 1$; $buffer[0] \leftarrow \texttt{'\_'}$; ⟨ Open input files 36 ⟩

    $include\_depth \leftarrow 0$; $cur\_line \leftarrow 0$; $change\_line \leftarrow 0$; $change\_depth \leftarrow include\_depth$; $changing \leftarrow 1$;

    $prime\_the\_change\_buffer()$; $changing \leftarrow \neg changing$; $limit \leftarrow buffer$; $loc \leftarrow buffer + 1$;

    $buffer[0] \leftarrow \texttt{'\_'}$; $input\_has\_ended \leftarrow false$;

  }

36. The following code opens the input files.

⟨ Open input files 36 ⟩ ≡

  if $((web\_file \leftarrow fopen(web\_file\_name, \texttt{"r"})) \equiv \Lambda)$ {

    $strcpy(web\_file\_name, alt\_web\_file\_name)$;

    if $((web\_file \leftarrow fopen(web\_file\_name, \texttt{"r"})) \equiv \Lambda)$

      $fatal(\texttt{"!\_Cannot\_open\_input\_file\_"}, web\_file\_name)$;

  }

  $web\_file\_open \leftarrow true$;

  if $((change\_file \leftarrow fopen(change\_file\_name, \texttt{"r"})) \equiv \Lambda)$

    $fatal(\texttt{"!\_Cannot\_open\_change\_file\_"}, change\_file\_name)$;

This code is used in section 35.

37. 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.

⟨ Global variables 18 ⟩ +≡

  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* ← *false*;   ▷ should CTANGLE print line and file info? ◁

38.   **boolean** *get_line*(**void**)   ▷ inputs the next line ◁
  {
  restart:
     if (*changing* ∧ *include_depth* ≡ *change_depth*) ⟨Read from *change_file* and maybe turn off *changing* 41⟩
     if (¬*changing* ∨ *include_depth* > *change_depth*) {
        ⟨Read from *cur_file* and maybe turn on *changing* 40⟩
        if (*changing* ∧ *include_depth* ≡ *change_depth*) **goto** *restart*;
     }
     if (*input_has_ended*) **return** *false*;
     loc ← *buffer*; *limit* ← '␣';
     if (*buffer*[0] ≡ '@' ∧ (*buffer*[1] ≡ 'i' ∨ *buffer*[1] ≡ 'I')) {
        loc ← *buffer* + 2; *limit* ← '"';
        while (*loc* ≡ '␣' ∨ *loc* ≡ '\t') *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* 39⟩
     }
     return *true*;
  }

39.   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*;
          }

⟨ Try to open include file, abort push if unsuccessful, go to *restart* 39 ⟩ ≡
  {
      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 ◁
      if (∗*loc* ≡ '"') {
        loc ++;
          while (∗*loc* ≠ '"' ∧ *k* ≤ *cur_file_name_end*) ∗*k*++ ← ∗*loc*++;
          if (*loc* ≡ *limit*) *k* ← *cur_file_name_end* + 1;   ▷ unmatched quote is 'too long' ◁
      }
      else
          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* ← *true*; **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* ← *true*; **goto** *restart*;   ▷ success ◁
        }
      }
      include_depth −−; *err_print*("!␣Cannot␣open␣include␣file"); **goto** *restart*;
  }

This code is used in section 38.

40. ⟨ Read from *cur_file* and maybe turn on *changing* 40 ⟩ ≡
  {
      cur_line ++;
      while (¬*input_ln*(*cur_file*)) {   ▷ pop the stack or quit ◁
        print_where ← *true*;
        if (*include_depth* ≡ 0) {
          input_has_ended ← *true*; **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 38.

41. ⟨ Read from *change_file* and maybe turn off *changing* 41 ⟩ ≡
  {
      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(*false*);
          if (*change_pending*) {
            changed_section[*section_count*] ← *true*; *change_pending* ← *false*;
          }
        }
        ∗limit ← '␣';
        if (*buffer*[0] ≡ '@') {
          if (*xisupper*(*buffer*[1])) *buffer*[1] ← *tolower*((**eight_bits**) *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* ← *true*;
          }
        }
      }
  }

This code is used in section 38.

42.   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(void)
{
  if (change_limit ≠ change_buffer) {      ▷ changing is false ◁
    strncpy(buffer, change_buffer, (size_t)(change_limit − change_buffer + 1));
    limit ← buffer + (ptrdiff_t)(change_limit − change_buffer); changing ← true;
    change_depth ← include_depth; loc ← buffer; err_print("!␣Change␣file␣entry␣did␣not␣match");
  }
}
```

43. 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*. You find the complete layout of **name_info** in the interface sections above.

The actual sequence of characters in the name pointed to by a **name_pointer** p appears in positions $p\text{-}byte\_start$ to $(p + 1)\text{-}byte\_start - 1$, inclusive.

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 $hash[h]$, $hash[h]\text{-}link$, $hash[h]\text{-}link\text{-}link$, ..., until either finding the desired name or encountering the null pointer.

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**.

The last component of **name_info** is different for CTANGLE and CWEAVE. In CTANGLE, if p is a pointer to a section name, $p\text{-}equiv$ 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, $p\text{-}xref$ 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 **void**.

⟨ Global variables 18 ⟩ +≡
  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* ◁

44. 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*.

⟨ Global variables 18 ⟩ +≡
  char *∗byte_ptr*;   ▷ first unused position in *byte_mem* ◁
  name_pointer *name_ptr*;   ▷ first unused position in *name_dir* ◁

45. ⟨ Initialize pointers 45 ⟩ ≡
  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 ◁
  root ← Λ;   ▷ the binary search tree starts out with nothing in it ◁

See also section 47.

This code is used in section 20.

46.   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* 8501   ▷ should be prime ◁

⟨ Global variables 18 ⟩ +≡
  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 ◁

47.   Initially all the hash lists are empty.

⟨ Initialize pointers 45 ⟩ +≡
  for (*h* ← *hash*; *h* ≤ *hash_end*; *∗h*++ ← Λ) ;

48.   Here is the main procedure for finding identifiers:

  name_pointer *id_lookup*(▷ looks up a string in the identifier table ◁
    const char *∗first*,   ▷ first character of string ◁
    const char *∗last*,   ▷ last character of string plus one ◁
    char *t*)   ▷ the *ilk*; used by CWEAVE only ◁
  {
    const 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 ← (**int**)(*last* − *first*);   ▷ compute the length ◁
    ⟨ Compute the hash code *h* 49 ⟩
    ⟨ Compute the name location *p* 50 ⟩
    if (*p* ≡ *name_ptr*) ⟨ Enter a new name into the table at position *p* 51 ⟩
    return *p*;
  }

49.   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* 49 ⟩ ≡
  h ← (**eight_bits**) *∗i*;
  while (++*i* < *last*) *h* ← (*h* + *h* + (**int**)((**eight_bits**) *∗i*)) % *hash_size*;
This code is used in section 48.

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

$\langle$ Compute the name location p 50 $\rangle \equiv$
  $p \leftarrow hash[h]$;
  while $(p \wedge \neg names\_match(p, first, l, t))\ \ p \leftarrow p\text{-}link$;
  if $(p \equiv \Lambda)$ {
    $p \leftarrow name\_ptr$; $\triangleright$ the current identifier is new $\triangleleft$
    $p\text{-}link \leftarrow hash[h]$; $hash[h] \leftarrow p$; $\triangleright$ insert p at beginning of hash list $\triangleleft$
  }

This code is used in section 48.

51.   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.

$\langle$ Enter a new name into the table at position p 51 $\rangle \equiv$
  {
    if $(byte\_ptr + l > byte\_mem\_end)\ overflow("byte_\sqcup memory")$;
    if $(name\_ptr \geq name\_dir\_end)\ overflow("name")$;
    $strncpy(byte\_ptr, first, l)$; $(++name\_ptr)\text{-}byte\_start \leftarrow byte\_ptr += l$; $init\_p(p, t)$;
  }

This code is used in section 48.

52.   If p is a **name_pointer** variable, as we have seen, $p\text{-}byte\_start$ 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 $p\text{-}byte\_start$, 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 $(p + 1)\text{-}link$, 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 $first\_chunk(p)$ $((p)\text{-}byte\_start + 2)$
#define $prefix\_length(p)$ $(\textbf{int})((\textbf{eight\_bits})*((p)\text{-}byte\_start)*256 + (\textbf{eight\_bits})*((p)\text{-}byte\_start + 1))$
#define $set\_prefix\_length(p, m)$ $(*((p)\text{-}byte\_start) \leftarrow (m)/256, *((p)\text{-}byte\_start + 1) \leftarrow (m) \% 256)$

  void $print\_section\_name(\textbf{name\_pointer}\ p)$
  {
    char $*ss, *s \leftarrow first\_chunk(p)$;
    name_pointer $q \leftarrow p + 1$;
    while $(p \neq name\_dir)$ {
      $ss \leftarrow (p + 1)\text{-}byte\_start - 1$;
      if $(*ss \equiv '_\sqcup' \wedge ss \geq s)$ {
        $term\_write(s, (\textbf{size\_t})(ss - s))$; $p \leftarrow q\text{-}link$; $q \leftarrow p$;
      }
      else {
        $term\_write(s, (\textbf{size\_t})(ss + 1 - s))$; $p \leftarrow name\_dir$; $q \leftarrow \Lambda$;
      }
      $s \leftarrow p\text{-}byte\_start$;
    }
    if $(q)\ term\_write("...", 3)$; $\triangleright$ complete name not yet known $\triangleleft$
  }

53.   **void** *sprint_section_name*(**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*, (**size_t**)(*ss* − *s*)), *dest* += *ss* − *s*; *s* ← *p*⃗*byte_start*;
     }
     ∗*dest* ← '\0';
  }

54.   **void** *print_prefix_name*(**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);
  }

55.   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 ◁

⟨ Predeclaration of procedures 7 ⟩ +≡
  static int *web_strcmp*(**char** ∗, **int**, **char** ∗, **int**);
  static name_pointer *add_section_name*(**name_pointer**, **int**, **char** ∗, **char** ∗, **int**);
  static void *extend_section_name*(**name_pointer**, **char** ∗, **char** ∗, **int**);

56.   **static int** *web_strcmp* (▷ fuller comparison than *strcmp* ◁
    char ∗*j*,   ▷ beginning of first string ◁
    int *j_len*,   ▷ length of first string ◁
    char ∗*k*,   ▷ beginning of second string ◁
    int *k_len*)   ▷ length of second string ◁
{
   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*;
}

57.   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.

   static name_pointer *add_section_name* (▷ 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* ← (**int**)(*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*);
}

58.   **static void** *extend_section_name*(**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 $q \leftarrow p + 1$;
  int *name_len* $\leftarrow$ (**int**)(*last* − *first*) + *ispref*;
  if (*name_ptr* $\geq$ *name_dir_end*) *overflow*("name");
  while (q‑*link* $\neq$ *name_dir*) $q \leftarrow q$‑*link*;
  q‑*link* $\leftarrow$ *name_ptr*; $s \leftarrow$ *name_ptr*‑*byte_start*; *name_ptr*‑*link* $\leftarrow$ *name_dir*;
  if (s + *name_len* > *byte_mem_end*) *overflow*("byte␣memory");
  (++*name_ptr*)‑*byte_start* $\leftarrow$ *byte_ptr* $\leftarrow$ s + *name_len*; *strncpy*(s, *first*, *name_len*);
  if (*ispref*) *(*byte_ptr* − 1) $\leftarrow$ '␣';
}

59.   The *section_lookup* procedure is supposed to find a section name that matches a new name, installing the new name if it 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*(▷ find or install section name in tree ◁
    char $*first$, **char** $*last$,   ▷ first and last characters of new name ◁
    int *ispref*)   ▷ is the new name a prefix or a full name? ◁
{
  int $c \leftarrow 0$;   ▷ comparison between two names; initialized so some compilers won't complain ◁
  name_pointer $p \leftarrow root$;   ▷ current node of the search tree ◁
  name_pointer $q \leftarrow \Lambda$;   ▷ another place to look in the tree ◁
  name_pointer $r \leftarrow \Lambda$;   ▷ where a match has been found ◁
  name_pointer *par* $\leftarrow \Lambda$;   ▷ parent of p, if r is Λ; otherwise parent of r ◁
  int *name_len* $\leftarrow$ (**int**)(*last* − *first*) + 1;
  ⟨ Look for matches for new name among shortest prefixes, complaining if more than one is found 60 ⟩
  ⟨ If no match found, add new name to tree 61 ⟩
  ⟨ If one match found, check for compatibility and return match 62 ⟩
}

60. 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 60 ⟩ ≡
 while (p) { ▷ compare shortest prefix of p with new name ◁
  $c \leftarrow web\_strcmp(first, name\_len, first\_chunk(p), prefix\_length(p))$;
  if $(c \equiv less \vee c \equiv greater)$ { ▷ new name does not match p ◁
   if $(r \equiv \Lambda)$ ▷ no previous matches have been found ◁
    $par \leftarrow p$;
   $p \leftarrow (c \equiv less ? p\text{-}llink : p\text{-}rlink)$;
  }
  else { ▷ new name matches p ◁
   if $(r \neq \Lambda)$ { ▷ and also r: illegal ◁
    $fputs(\texttt{"\\n!\_Ambiguous\_prefix:\_matches\_<"}, stdout)$; $print\_prefix\_name(p)$;
    $fputs(\texttt{">\\n\_and\_<"}, stdout)$; $print\_prefix\_name(r)$; $err\_print(\texttt{">"})$; **return** $name\_dir$;
     ▷ the unsection ◁
   }
   $r \leftarrow p$; ▷ remember match ◁
   $p \leftarrow p\text{-}llink$; ▷ try another ◁
   $q \leftarrow r\text{-}rlink$; ▷ we'll get back here if the new p doesn't match ◁
  }
  if $(p \equiv \Lambda)$ $p \leftarrow q, q \leftarrow \Lambda$; ▷ q held the other branch of r ◁
 }

This code is used in section 59.

61. ⟨ If no match found, add new name to tree 61 ⟩ ≡
 if $(r \equiv \Lambda)$ ▷ no matches were found ◁
  return $add\_section\_name(par, c, first, last + 1, ispref)$;

This code is used in section 59.

62. 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 62 ⟩ ≡
 switch $(section\_name\_cmp(\&first, name\_len, r))$ { ▷ compare all of r with new name ◁
 case $prefix$:
  if $(\neg ispref)$ {
   $fputs(\texttt{"\\n!\_New\_name\_is\_a\_prefix\_of\_<"}, stdout)$; $print\_section\_name(r)$; $err\_print(\texttt{">"})$;
  }
  else if $(name\_len < prefix\_length(r))$ $set\_prefix\_length(r, name\_len)$; ▷ fall through ◁
 case $equal$: **return** r;
 case $extension$:
  if $(\neg ispref \vee first \leq last)$ $extend\_section\_name(r, first, last + 1, ispref)$;
  return r;
 case $bad\_extension$: $fputs(\texttt{"\\n!\_New\_name\_extends\_<"}, stdout)$; $print\_section\_name(r)$; $err\_print(\texttt{">"})$;
  return r;
 default: ▷ no match: illegal ◁
  $fputs(\texttt{"\\n!\_Section\_name\_incompatible\_with\_<"}, stdout)$; $print\_prefix\_name(r)$;
  $fputs(\texttt{">,\\n\_which\_abbreviates\_<"}, stdout)$; $print\_section\_name(r)$; $err\_print(\texttt{">"})$; **return** r;
 }

This code is used in section 59.

63. 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

```
static int section_name_cmp(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 (true) {
    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 + (ptrdiff_t)(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 −= (int)(ss − s);  s ← q→byte_start;  r ← q;  continue;
      }
      *pfirst ← first; return extension;
    default: return c;
    }
  }
}
```

64. ⟨ Predeclaration of procedures 7 ⟩ +≡ **static int** *section_name_cmp*(**char** ∗∗, **int**, **name_pointer**);

65.   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.

⟨ Global variables 18 ⟩ +≡
  int *history* ← *spotless*;   ▷ indicates how bad this run was ◁

66.   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 "!".

  void *err_print* (▷ prints '.' and location of error message ◁
      const 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 67 ⟩
    update_terminal; *mark_error*;
  }

67.   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 67 ⟩ ≡
  {
    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 66.

68. 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. See the function(s) *print_stats* in the interface above and in the index.

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(void)
{
   if (show_progress) new_line;
   if (show_stats) print_stats();        ▷ print statistics about memory usage ◁
   ⟨Print the job history 69⟩
   if (history > harmless_message) return EXIT_FAILURE;
   else return EXIT_SUCCESS;
}
```

69. ⟨Print the job *history* 69⟩ ≡
```
switch (history) {
case spotless:
   if (show_happiness) puts("(No␣errors␣were␣found.)");
   break;
case harmless_message: puts("(Did␣you␣see␣the␣warning␣message␣above?)"); break;
case error_message: puts("(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)"); break;
case fatal_message: puts("(That␣was␣a␣fatal␣error,␣my␣friend.)");
}        ▷ there are no other cases ◁
```
This code is used in section 68.

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

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

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

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

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

72. 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").

73.   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.

⟨ Global variables 18 ⟩ +≡
    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 ◁

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

⟨ Set the default options common to CTANGLE and CWEAVE 74 ⟩ ≡
    show_banner ← *show_happiness* ← *show_progress* ← *make_xrefs* ← *true*;
    show_stats ← *false*;
This code is used in section 20.

75.   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.

```
static void scan_args(void)
{
   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 ← false, found_change ← false, found_out ← false;
      ▷ have these names been seen? ◁
   strcpy(change_file_name, "/dev/null");
   while (−−argc > 0) {
      if ((**(++argv) ≡ '−' ∨ **argv ≡ '+') ∧ *(*argv + 1)) ⟨Handle flag argument 80⟩
      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 77⟩
         else if (¬found_change) ⟨Make change_file_name from fname 78⟩
         else if (¬found_out) ⟨Override tex_file_name and C_file_name 79⟩
         else ⟨Print usage error message and quit 81⟩
      }
   }
   if (¬found_web) ⟨Print usage error message and quit 81⟩
}
```

76.   ⟨Predeclaration of procedures 7⟩ +≡   **static void** scan_args(**void**);

77.   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 77 ⟩ ≡
  {
    if $(s - *argv > max\_file\_name\_length - 5)$ ⟨ Complain about argument length 82 ⟩
    if $(dot\_pos \equiv \Lambda)$ $sprintf(web\_file\_name, \texttt{"\%s.w"}, *argv)$;
    else {
      $strcpy(web\_file\_name, *argv)$; $*dot\_pos \leftarrow 0$;   ▷ string now ends where the dot was ◁
    }
    $sprintf(alt\_web\_file\_name, \texttt{"\%s.web"}, *argv)$; $sprintf(tex\_file\_name, \texttt{"\%s.tex"}, name\_pos)$;
      ▷ strip off directory name ◁
    $sprintf(idx\_file\_name, \texttt{"\%s.idx"}, name\_pos)$; $sprintf(scn\_file\_name, \texttt{"\%s.scn"}, name\_pos)$;
    $sprintf(C\_file\_name, \texttt{"\%s.c"}, name\_pos)$; $found\_web \leftarrow true$;
  }
This code is used in section 75.

78.   ⟨ Make $change\_file\_name$ from $fname$ 78 ⟩ ≡
  {
    if $(strcmp(*argv, \texttt{"-"}) \neq 0)$ {
      if $(s - *argv > max\_file\_name\_length - 4)$ ⟨ Complain about argument length 82 ⟩
      if $(dot\_pos \equiv \Lambda)$ $sprintf(change\_file\_name, \texttt{"\%s.ch"}, *argv)$;
      else $strcpy(change\_file\_name, *argv)$;
    }
    $found\_change \leftarrow true$;
  }
This code is used in section 75.

79.   ⟨ Override tex_file_name and C_file_name 79 ⟩ ≡
  {
    if $(s - *argv > max\_file\_name\_length - 5)$ ⟨ Complain about argument length 82 ⟩
    if $(dot\_pos \equiv \Lambda)$ {
      $sprintf(tex\_file\_name, \texttt{"\%s.tex"}, *argv)$; $sprintf(idx\_file\_name, \texttt{"\%s.idx"}, *argv)$;
      $sprintf(scn\_file\_name, \texttt{"\%s.scn"}, *argv)$; $sprintf(C\_file\_name, \texttt{"\%s.c"}, *argv)$;
    }
    else {
      $strcpy(tex\_file\_name, *argv)$; $strcpy(C\_file\_name, *argv)$;
      if $(make\_xrefs)$ {   ▷ indexes will be generated ◁
        $*dot\_pos \leftarrow 0$; $sprintf(idx\_file\_name, \texttt{"\%s.idx"}, *argv)$; $sprintf(scn\_file\_name, \texttt{"\%s.scn"}, *argv)$;
      }
    }
    $found\_out \leftarrow true$;
  }
This code is used in section 75.

80.   **#define** *flag_change* $(**argv \neq$ '−'$)$

⟨ Handle flag argument 80 ⟩ ≡
  {
    for $(dot\_pos \leftarrow *argv + 1; *dot\_pos >$ '\0'; $dot\_pos {+}{+})$ $flags[(\textbf{eight\_bits}) *dot\_pos] \leftarrow flag\_change$;
  }
This code is used in section 75.

81.   ⟨ Print usage error message and quit 81 ⟩ ≡
  {
    if $(program \equiv ctangle)$
      $fatal($"!␣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 75.

82.   ⟨ Complain about argument length 82 ⟩ ≡
  $fatal($"!␣Filename␣too␣long\n", $*argv)$;
This code is used in sections 77, 78, and 79.

83.   Output.   Here is the code that opens the output file:

⟨ Global variables 18 ⟩ +≡
  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 ◁

84.   ⟨ Scan arguments and open output files 84 ⟩ ≡
  scan_args( );
  if (*program* ≡ *ctangle*) {
    if ((*C_file* ← *fopen*(*C_file_name*, "wb")) ≡ Λ) *fatal*("!␣Cannot␣open␣output␣file␣", *C_file_name*);
  }
  else {
    if ((*tex_file* ← *fopen*(*tex_file_name*, "wb")) ≡ Λ)
      fatal("!␣Cannot␣open␣output␣file␣", *tex_file_name*);
  }
This code is used in section 20.

85.   Index.