The WEAVE processor

(Version 4.5)

Editor's Note: The present variant of this C/WEB source file has been modified for use in the TEX Live system.

The following sections were changed by the change file: 1, 2, 8, 12, 17, 20, 21, 22, 24, 26, 28, 33, 37, 50, 124, 127, 148, 151, 157, 161, 162, 166, 167, 169, 170, 172, 173, 174, 239, 258, 259, 261, 264, 265, 266, 267, 268, 269, 270, 271, 272.

1*. Introduction. This program converts a `WEB` file to a TeX file. It was written by D. E. Knuth in October, 1981; a somewhat similar SAIL program had been developed in March, 1979, although the earlier program used a top-down parsing method that is quite different from the present scheme.

The code uses a few features of the local Pascal compiler that may need to be changed in other installations:

1) Case statements have a default.
2) Input-output routines may need to be adapted for use with a particular character set and/or for printing messages on the user's terminal.

These features are also present in the Pascal version of TeX, where they are used in a similar (but more complex) way. System-dependent portions of `WEAVE` can be identified by looking at the entries for 'system dependencies' in the index below.

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

define *my_name* ≡ `´weave´`
define *banner* ≡ `´This␣is␣WEAVE,␣Version␣4.5´`

2*. The program begins with a fairly normal header, made up of pieces that will mostly be filled in later. The `WEB` input comes from files *web_file* and *change_file*, and the TeX output goes to file *tex_file*.

If it is necessary to abort the job because of a fatal error, the program calls the '*jump_out*' procedure.

⟨ Compiler directives 4 ⟩
program *WEAVE*(*web_file*, *change_file*, *tex_file*);
  const ⟨ Constants in the outer block 8* ⟩
  type ⟨ Types in the outer block 11 ⟩
  var ⟨ Globals in the outer block 9 ⟩
    ⟨ Define *parse_arguments* 264* ⟩
    ⟨ Error handling procedures 30 ⟩
  procedure *initialize*;
    var ⟨ Local variables for initialization 16 ⟩
    begin *kpse_set_program_name*(*argv*[0], *my_name*); *parse_arguments*; ⟨ Set initial values 10 ⟩
    end;

8*. The following parameters are set big enough to handle TeX, so they should be sufficient for most applications of `WEAVE`.

⟨ Constants in the outer block 8* ⟩ ≡
  max_bytes = 65535; { 1/*ww* times the number of bytes in identifiers, index entries, and module names; must be less than 65536 }
  max_names = 10239; { number of identifiers, index entries, and module names; must be less than 10240 }
  max_modules = 4000; { greater than the total number of modules }
  hash_size = 8501; { should be prime }
  buf_size = 1000; { maximum length of input line }
  longest_name = 10000; { module names shouldn't be longer than this }
  long_buf_size = *buf_size* + *longest_name*; { C arithmetic in Pascal constant }
  line_length = 80; { lines of TeX output have at most this many characters, should be less than 256 }
  max_refs = 65535; { number of cross references; must be less than 65536 }
  max_toks = 65535; { number of symbols in Pascal texts being parsed; must be less than 65536 }
  max_texts = 10239; { number of phrases in Pascal texts being parsed; must be less than 10240 }
  max_scraps = 10000; { number of tokens in Pascal texts being parsed }
  stack_size = 2000; { number of simultaneous output levels }

This code is used in section 2*.

12* The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, so `WEB` assumes that it is being used with a Pascal whose character set contains at least the characters of standard ASCII as listed above. Some Pascal compilers use the original name *char* for the data type associated with the characters in text files, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name.

In order to accommodate this difference, we shall use the name *text_char* to stand for the data type of the characters in the input and output files. We shall also assume that *text_char* consists of the elements *chr*(*first_text_char*) through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define *text_char* ≡ *ASCII_code* { the data type of characters in text files }
define *first_text_char* = 0 { ordinal number of the smallest element of *text_char* }
define *last_text_char* = 255 { ordinal number of the largest element of *text_char* }

⟨ Types in the outer block 11 ⟩ +≡
text_file = **packed file of** *text_char*;

17* Here now is the system-dependent part of the character set. If `WEB` is being implemented on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, you don't need to make any changes here. But if you have, for example, an extended character set like the one in Appendix C of *The TEXbook*, the first line of code in this module should be changed to

$$\textbf{for } i \leftarrow 1 \textbf{ to } '37 \textbf{ do } xchr[i] \leftarrow chr(i);$$

`WEB`'s character set is essentially identical to TEX's, even with respect to characters less than ´*40*.

Changes to the present module will make `WEB` more friendly on computers that have an extended character set, so that one can type things like ≠ instead of `<>`. If you have an extended set of characters that are easily incorporated into text files, you can assign codes arbitrarily here, giving an *xchr* equivalent to whatever characters the users of `WEB` are allowed to have in their input files, provided that unsuitable characters do not correspond to special codes like *carriage_return* that are listed above.

(The present file `WEAVE.WEB` does not contain any of the non-ASCII characters, because it is intended to be used with all implementations of `WEB`. It was originally created on a Stanford system that has a convenient extended character set, then "sanitized" by applying another program that transliterated all of the non-standard characters into standard equivalents.)

⟨ Set initial values 10 ⟩ +≡
for $i \leftarrow 1$ **to** ´*37* **do** $xchr[i] \leftarrow chr(i)$;
for $i \leftarrow$ ´*200* **to** ´*377* **do** $xchr[i] \leftarrow chr(i)$;

20* Terminal output is done by writing on file *term_out*, which is assumed to consist of characters of type *text_char*:

define *term_out* ≡ *stdout*
define *print*(#) ≡ *write*(*term_out*, #) { '*print*' means write on the terminal }
define *print_ln*(#) ≡ *write_ln*(*term_out*, #) { '*print*' and then start new line }
define *new_line* ≡ *write_ln*(*term_out*) { start new line }
define *print_nl*(#) ≡ { print information starting on a new line }
  begin *new_line*; *print*(#);
  end

21* Different systems have different ways of specifying that the output on a certain file will appear on the user's terminal.

⟨ Set initial values 10 ⟩ +≡
  { nothing need be done }

22* 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*(*term_out*) { empty the terminal output buffer }

24* The following code opens the input files. This is called after the filename variables have been set appropriately.

procedure *open_input*; { prepare to read *web_file* and *change_file* }
begin *web_file* ← *kpse_open_file*(*web_name*, *kpse_web_format*);
if *chg_name* **then** *change_file* ← *kpse_open_file*(*chg_name*, *kpse_web_format*);
end;

26* The following code opens *tex_file*. Since this file was listed in the program header, we assume that the Pascal runtime system has checked that a suitable external file name has been given.

⟨ Set initial values 10 ⟩ +≡
  rewrite(*tex_file*, *tex_name*);

28* The *input_ln* procedure brings the next line of input from the specified file into the *buffer* array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false*. The conventions of TeX are followed; i.e., *ASCII_code* numbers representing the next line of the file are input into *buffer*[0], *buffer*[1], ..., *buffer*[*limit* − 1]; trailing blanks are ignored; and the global variable *limit* is set to the length of the line. The value of *limit* must be strictly less than *buf_size*.

We assume that none of the *ASCII_code* values of *buffer*[j] for $0 \le j < limit$ is equal to 0, ´177, *line_feed*, *form_feed*, or *carriage_return*. Since *buf_size* is strictly less than *long_buf_size*, some of WEAVE's routines use the fact that it is safe to refer to *buffer*[*limit* + 2] without overstepping the bounds of the array.

```
function input_ln(var f : text_file): boolean;   { inputs a line or returns false }
  var final_limit: 0 .. buf_size;   { limit without trailing blanks }
  begin limit ← 0; final_limit ← 0;
  if eof(f) then input_ln ← false
  else begin while ¬eoln(f) do
      begin buffer[limit] ← xord[getc(f)]; incr(limit);
      if buffer[limit − 1] ≠ "␣" then final_limit ← limit;
      if limit = buf_size then
        begin while ¬eoln(f) do vgetc(f);
        decr(limit);   { keep buffer[buf_size] empty }
        if final_limit > limit then final_limit ← limit;
        print_nl(´!␣Input␣line␣too␣long´); loc ← 0; error;
        end;
      end;
    read_ln(f); limit ← final_limit; input_ln ← true;
    end;
  end;
```

33* The *jump_out* procedure just cuts across all active procedure levels and jumps out of the program. It is used when no recovery from a particular error has been provided.

define *fatal_error*(#) ≡
 begin *new_line*; *write*(*stderr*, #); *error*; *mark_fatal*; *jump_out*;
 end

⟨ Error handling procedures 30 ⟩ +≡
procedure *jump_out*;
 begin stat ⟨ Print statistics about memory usage 262 ⟩; **tats**
{ here files should be closed if the operating system requires it }
 ⟨ Print the job *history* 263 ⟩;
 new_line;
 if $(history \neq spotless) \land (history \neq harmless\_message)$ **then** *uexit*(1)
 else *uexit*(0);
 end;

37* `WEAVE` has been designed to avoid the need for indices that are more than sixteen bits wide, so that it can be used on most computers. But there are programs that need more than 65536 bytes; TEX is one of these (and the pdfTEX variant even requires more than twice that amount when its "final" change file is applied). To get around this problem, a slight complication has been added to the data structures: *byte_mem* is a two-dimensional array, whose first index is either 0, 1 or 2. (For generality, the first index is actually allowed to run between 0 and $ww - 1$, where ww is defined to be 3; the program will work for any positive value of ww, and it can be simplified in obvious ways if $ww = 1$.)

define $ww = 3$ { we multiply the byte capacity by approximately this amount }

⟨ Globals in the outer block 9 ⟩ +≡
byte_mem: **packed array** $[0 \mathinner{..} ww - 1, 0 \mathinner{..} max\_bytes]$ **of** *ASCII_code*; { characters of names }
byte_start: **array** $[0 \mathinner{..} max\_names]$ **of** *sixteen_bits*; { directory into *byte_mem* }
link: **array** $[0 \mathinner{..} max\_names]$ **of** *sixteen_bits*; { hash table or tree links }
ilk: **array** $[0 \mathinner{..} max\_names]$ **of** *sixteen_bits*; { type codes or tree links }
xref: **array** $[0 \mathinner{..} max\_names]$ **of** *sixteen_bits*; { heads of cross-reference lists }

50* 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 Pascal's reserved words.

If the user has sent the *no_xref* flag (the '`-x`' option of the command line), then it is unnecessary to keep track of cross references for identifiers. If one were careful, one could probably make more changes around module 100 to avoid a lot of identifier looking up.

define *append_xref*(#) ≡
    if $xref\_ptr = max\_refs$ **then** *overflow*(´`cross␣reference`´)
    else begin *incr*(*xref_ptr*); $num(xref\_ptr) \gets$ #;
      end

procedure *new_xref*(*p* : *name_pointer*);
  label *exit*;
  var *q*: *xref_number*; { pointer to previous cross-reference }
    m, *n*: *sixteen_bits*; { new and previous cross-reference value }
  begin if *no_xref* **then return**;
  if $(reserved(p) \lor (byte\_start[p] + 1 = byte\_start[p + ww])) \land (xref\_switch = 0)$ **then return**;
  $m \gets module\_count + xref\_switch$; $xref\_switch \gets 0$; $q \gets xref[p]$;
  if $q > 0$ **then**
    begin $n \gets num(q)$;
    if $(n = m) \lor (n = m + def\_flag)$ **then return**
    else if $m = n + def\_flag$ **then**
        begin $num(q) \gets m$; **return**;
        end;
    end;
  append_xref(*m*); $xlink(xref\_ptr) \gets q$; $xref[p] \gets xref\_ptr$;
exit: **end**;

124* 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 webmac`'.

If the user has sent the *pdf_output* flag (the '`-p`' option of the command line), then we use alternative TEX macros from '`\input pwebmac`'.

⟨Set initial values 10⟩ +≡
  out_ptr ← 1; *out_line* ← 1; *out_buf*[1] ← "c";
  if *pdf_output* **then** *write*(*tex_file*, ´\input␣pwebma´)
  else *write*(*tex_file*, ´\input␣webma´);

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

procedure *break_out*; { finds a way to break the output line }
  label *exit*;
  var *k*: 0 . . *line_length*; { index into *out_buf* }
    d: *ASCII_code*; { character from the buffer }
  begin *k* ← *out_ptr*;
  loop begin if $k = 0$ **then** ⟨Print warning message, break the line, **return** 128⟩;
    d ← *out_buf*[*k*];
    if *d* = "␣" **then**
      begin *flush_buffer*(*k*, *false*, *true*); **return**;
      end;
    if (*d* = "\") ∧ (*out_buf*[*k* − 1] ≠ "\") ∧ (*out_buf*[*k* − 1] ≠ "%") **then** { in this case $k > 1$ }
      begin *flush_buffer*(*k* − 1, *true*, *true*); **return**;
      end;
    decr(*k*);
    end;
exit: **end**;

148*. The production rules listed above are embedded directly into the `WEAVE` program, 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 '*simp math* $\rightarrow$ *math*' would be represented by '$(pp, 2, math, -1)$'; in this case the pointer *pp* should decrease by 1 after the production has been applied, because some productions with *math* in their second positions might now match, but no productions have *math* in the third or 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. Consider the further example '*var_head math colon* $\rightarrow$ *var_head intro*', which is represented by '$(pp+1, 2, intro, +1)$'; the $+1$ here is deduced by looking at the grammar and seeing that no matches could possibly occur at positions $\le pp$ after this production has been applied. 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 Pascal 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 '$reduce(pp+1, 2, intro, +1)(52)$' 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, $trans[pp]$ and $trans[pp+1]$, to the current token list. If the entire new translation is formed in this way, we write '$squash(j,k,c,d)$' instead of '$reduce(j,k,c,d)$'. For example, '$squash(pp, 2, math, -1)$' is an abbreviation for '$app2(pp)$; $reduce(pp, 2, math, -1)$'.

The code below is an exact translation of the production rules into Pascal, 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.

Caution: The macros *app*, *app1*, *app2*, and *app3* are sequences of statements that are not enclosed with **begin** and **end**, because such delimiters would make the Pascal program much longer. This means that it is necessary to write **begin** and **end** explicitly when such a macro is used as a single statement. Several mysterious bugs in the original programming of `WEAVE` were caused by a failure to remember this fact. Next time the author will know better.

define *production*(#) ≡
  debug *prod*(#)
  gubed;
 goto *found*;
 end
define *reduce*(#) ≡
 begin *red*(#); *production*
define *squash*(#) ≡
 begin *sq*(#); *production*
define *app*(#) ≡ *tok_mem*[*tok_ptr*] ← #; *incr*(*tok_ptr*)
  { this is like *app_tok*, but it doesn't test for overflow }
define *app1*(#) ≡ *tok_mem*[*tok_ptr*] ← *tok_flag* + *trans*[#]; *incr*(*tok_ptr*)
define *app2*(#) ≡ *app1*(#); *app1*(# + 1)
define *app3*(#) ≡ *app2*(#); *app1*(# + 2)

151* 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 *alpha* 151* ⟩ ≡
if $cat[pp+1] = math$ **then**
  begin if $cat[pp+2] = colon$ **then** $squash(pp+1, 2, math, 0)(1)$
  else if $cat[pp+2] = omega$ **then**
    begin $app1(pp)$; $app("␣")$; $app("\$")$; $app1(pp+1)$; $app("\$")$; $app("␣")$; $app(indent)$;
    $app1(pp+2)$; $reduce(pp, 3, clause, -2)(2)$;
    end;
  end
else if $cat[pp+1] = omega$ **then**
  begin $app1(pp)$; $app("␣")$; $app(indent)$; $app1(pp+1)$; $reduce(pp, 2, clause, -2)(3)$;
  end
  else if $cat[pp+1] = simp$ **then** $reduce(pp+1, 0, math, 0)(4)$

This code is used in section 150.

157* ⟨ Cases for *elsie* 157* ⟩ ≡
$reduce(pp, 0, intro, -3)(14)$

This code is used in section 149.

161* ⟨ Cases for *mod_scrap* 161* ⟩ ≡
if $(cat[pp+1] = terminator) \lor (cat[pp+1] = semi)$ **then**
  begin $app2(pp)$; $app(force)$; $reduce(pp, 2, stmt, -2)(24)$;
  end
else $reduce(pp, 0, simp, -2)(25)$

This code is used in section 149.

162* ⟨Cases for *open* 162*⟩ ≡
if $(cat[pp+1] = case\_head) \land (cat[pp+2] = close)$ **then**
begin $app1(pp)$; $app(\texttt{"\$"})$; $app(cancel)$; $app1(pp+1)$; $app(cancel)$; $app(outdent)$; $app(\texttt{"\$"})$;
$app1(pp+2)$; $reduce(pp, 3, math, -1)(26)$;
end
else if $cat[pp+1] = close$ **then**
begin $app1(pp)$; $app(\texttt{"\textbackslash"})$; $app(\texttt{","})$; $app1(pp+1)$; $reduce(pp, 2, math, -1)(27)$;
end
else if $cat[pp+1] = math$ **then** ⟨Cases for *open math* 163⟩
else if $cat[pp+1] = proc$ **then**
begin if $cat[pp+2] = intro$ **then**
begin $app(math\_op)$; $app(cancel)$; $app1(pp+1)$; $app(\texttt{"\}"})$; $reduce(pp+1, 2, math, 0)(34)$;
end;
end
else if $cat[pp+1] = simp$ **then** $reduce(pp+1, 0, math, 0)(35)$
else if $(cat[pp+1] = stmt) \land (cat[pp+2] = close)$ **then**
begin $app1(pp)$; $app(\texttt{"\$"})$; $app(cancel)$; $app1(pp+1)$; $app(cancel)$; $app(\texttt{"\$"})$;
$app1(pp+2)$; $reduce(pp, 3, math, -1)(36)$;
end
else if $cat[pp+1] = var\_head$ **then**
begin if $cat[pp+2] = intro$ **then**
begin $app(math\_op)$; $app(cancel)$; $app1(pp+1)$; $app(\texttt{"\}"})$;
$reduce(pp+1, 2, math, 0)(37)$;
end;
end

This code is used in section 150.

166* ⟨Cases for *semi* 166*⟩ ≡
$reduce(pp, 0, terminator, -3)(42)$

This code is used in section 149.

167* ⟨Cases for *simp* 167*⟩ ≡
if $cat[pp+1] = close$ **then** $reduce(pp, 0, stmt, -2)(43)$
else if $cat[pp+1] = colon$ **then**
begin $app(force)$; $app(backup)$; $squash(pp, 2, intro, -3)(44)$;
end
else if $cat[pp+1] = math$ **then** $squash(pp, 2, math, -1)(45)$
else if $cat[pp+1] = mod\_scrap$ **then** $squash(pp, 2, mod\_scrap, 0)(46)$
else if $cat[pp+1] = simp$ **then** $squash(pp, 2, simp, -2)(47)$
else if $cat[pp+1] = terminator$ **then** $squash(pp, 2, stmt, -2)(48)$

This code is used in section 150.

169* ⟨Cases for *terminator* 169*⟩ ≡
$reduce(pp, 0, stmt, -2)(50)$

This code is used in section 149.

170* ⟨ Cases for *var_head* 170* ⟩ ≡

```
if cat[pp + 1] = beginning then reduce(pp, 0, stmt, −2)(51)
else if cat[pp + 1] = math then
      begin if cat[pp + 2] = colon then
        begin app("$"); app1(pp + 1); app("$"); app1(pp + 2); reduce(pp + 1, 2, intro, +1)(52);
        end;
      end
    else if cat[pp + 1] = simp then
        begin if cat[pp + 2] = colon then squash(pp + 1, 2, intro, +1)(53);
        end
      else if cat[pp + 1] = stmt then
          begin app1(pp); app(break_space); app1(pp + 1); reduce(pp, 2, var_head, −2)(54);
          end
```

This code is used in section 149.

172* The '*reduce*' macro used in our code for productions actually calls on a procedure named '*red*', which makes the appropriate changes to the scrap list. This procedure takes advantage of the simplification that occurs when $k = 0$.

```
procedure red(j : sixteen_bits; k : eight_bits; c : eight_bits; d : integer);
  var i: 0 .. max_scraps;   { index into scrap memory }
  begin cat[j] ← c;
  if k > 0 then
    begin trans[j] ← text_ptr; freeze_text;
    end;
  if k > 1 then
    begin for i ← j + k to lo_ptr do
      begin cat[i − k + 1] ← cat[i]; trans[i − k + 1] ← trans[i];
      end;
    lo_ptr ← lo_ptr − k + 1;
    end;
  ⟨ Change pp to max(scrap_base, pp+d) 173* ⟩;
  end;
```

173* ⟨ Change *pp* to max(*scrap_base*, *pp*+*d*) 173* ⟩ ≡

```
if pp + d ≥ scrap_base then pp ← pp + d
else pp ← scrap_base
```

This code is used in section 172*.

174* Similarly, the '*squash*' macro invokes a procedure called '*sq*', which combines app_k and *red* for matching numbers k.

```
procedure sq(j : sixteen_bits; k : eight_bits; c : eight_bits; d : integer);
  begin case k of
  1: begin app1(j); end;
  2: begin app2(j); end;
  3: begin app3(j); end;
  othercases confusion(´squash´)
  endcases;
  red(j, k, c, d);
  end;
```

239* Phase three processing. We are nearly finished! `WEAVE`'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, module name list, and table of contents.

⟨ Phase III: Output the cross-reference index 239* ⟩ ≡

```
if no_xref then
   begin finish_line; out("\"); out5("v")("f")("i")("l")("l"); out4("\")("e")("n")("d"); finish_line;
   end
else begin phase_three ← true; print_nl(´Writing␣the␣index...´);
   if change_exists then
      begin finish_line; ⟨ Tell about changed modules 241 ⟩;
      end;
   finish_line; out4("\")("i")("n")("x"); finish_line; ⟨ Do the first pass of sorting 243 ⟩;
   ⟨ Sort and output the index 250 ⟩;
   out4("\")("f")("i")("n"); finish_line; ⟨ Output all the module names 257 ⟩;
   out4("\")("c")("o")("n"); finish_line;
   end;
print(´Done.´);
```

This code is used in section 261*.

258*. Debugging. The Pascal debugger with which `WEAVE` was developed allows breakpoints to be set, and variables can be read and changed, but procedures cannot be executed. Therefore a '*debug_help*' procedure has been inserted in the main loops of each phase of the program; when *ddt* and *dd* are set to appropriate values, symbolic printouts of various tables will appear.

The idea is to set a breakpoint inside the *debug_help* routine, at the place of '*breakpoint*:' below. Then when *debug_help* is to be activated, set *trouble_shooting* equal to *true*. The *debug_help* routine will prompt you for values of *ddt* and *dd*, discontinuing this when $ddt \le 0$; thus you type $2n + 1$ integers, ending with zero or a negative number. Then control either passes to the breakpoint, allowing you to look at and/or change variables (if you typed zero), or to exit the routine (if you typed a negative value).

Another global variable, *debug_cycle*, can be used to skip silently past calls on *debug_help*. If you set $debug\_cycle > 1$, the program stops only every *debug_cycle* times *debug_help* is called; however, any error stop will set *debug_cycle* to zero.

define $term\_in \equiv stdin$

⟨ Globals in the outer block 9 ⟩ +≡
debug *trouble_shooting*: *boolean*; { is *debug_help* wanted? }
ddt: *integer*; { operation code for the *debug_help* routine }
dd: *integer*; { operand in procedures performed by *debug_help* }
debug_cycle: *integer*; { threshold for *debug_help* stopping }
debug_skipped: *integer*; { we have skipped this many *debug_help* calls }
gubed

259*. The debugging routine needs to read from the user's terminal.

⟨ Set initial values 10 ⟩ +≡
debug $trouble\_shooting \leftarrow true$; $debug\_cycle \leftarrow 1$; $debug\_skipped \leftarrow 0$; $tracing \leftarrow 0$;
$trouble\_shooting \leftarrow false$; $debug\_cycle \leftarrow 99999$; { use these when it almost works }
gubed

261* The main program. Let's put it all together now: `WEAVE` starts and ends here.

The main procedure has been split into three sub-procedures in order to keep certain Pascal compilers from overflowing their capacity.

procedure *Phase_I*;
  begin ⟨ Phase I: Read all the user's text and store the cross references 109 ⟩;
  end;

procedure *Phase_II*;
  begin ⟨ Phase II: Read all the text again and translate it to TeX form 218 ⟩;
  end;

  begin *initialize*; { beginning of the main program }
  print(*banner*); { print a "banner line" }
  print_ln(*version_string*); ⟨ Store all the reserved words 64 ⟩;
  Phase_I; *Phase_II*;
  ⟨ Phase III: Output the cross-reference index 239* ⟩;
  ⟨ Check that all changes have been read 85 ⟩;
  jump_out;
  end.

264*. System-dependent changes. Parse a Unix-style command line.

```
define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
⟨ Define parse_arguments 264* ⟩ ≡
procedure parse_arguments;
  const n_options = 4;  { Pascal won't count array lengths for us. }
  var long_options: array [0 .. n_options] of getopt_struct;
    getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options;
  begin ⟨ Define the option table 265* ⟩;
  repeat getopt_return_val ← getopt_long_only(argc, argv, ´´, long_options, address_of(option_index));
    if getopt_return_val = −1 then
      begin do_nothing;  { End of arguments; we exit the loop below. }
      end
    else if getopt_return_val = "?" then
        begin usage(my_name);
        end
      else if argument_is(´help´) then
          begin usage_help(WEAVE_HELP, nil);
          end
        else if argument_is(´version´) then
            begin print_version_and_exit(banner, nil, ´D.E.␣Knuth´, nil);
            end;  { Else it was a flag; getopt has already done the assignment. }
  until getopt_return_val = −1;  { Now optind is the index of first non-option on the command line. }
  if (optind + 1 > argc) ∨ (optind + 3 < argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣one␣to␣three␣file␣arguments.´); usage(my_name);
    end;  { Supply ".web" and ".ch" extensions if necessary. }
  web_name ← extend_filename(cmdline(optind), ´web´);
  if optind + 2 ≤ argc then
    begin if strcmp(char_to_string(´-´), cmdline(optind + 1)) ≠ 0 then
      chg_name ← extend_filename(cmdline(optind + 1), ´ch´);
    end;  { Change ".web" to ".tex" and use the current directory. }
  if optind + 3 = argc then tex_name ← extend_filename(cmdline(optind + 2), ´tex´)
  else tex_name ← basename_change_suffix(web_name, ´.web´, ´.tex´);
  end;
```

This code is used in section 2*.

265*. Here are the options we allow. The first is one of the standard GNU options.

```
⟨ Define the option table 265* ⟩ ≡
  current_option ← 0; long_options[current_option].name ← ´help´;
  long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
  long_options[current_option].val ← 0; incr(current_option);
```

See also sections 266*, 267*, 268*, and 270*.

This code is used in section 264*.

266*. Another of the standard options.

```
⟨ Define the option table 265* ⟩ +≡
  long_options[current_option].name ← ´version´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

267* Use alternative TEX macros more suited for PDF output?

⟨ Define the option table 265* ⟩ +≡
long_options[*current_option*].*name* ← *char_to_string*(´p´); *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← *address_of*(*pdf_output*); *long_options*[*current_option*].*val* ← 1;
incr(*current_option*);

268* Omit cross-referencing?

⟨ Define the option table 265* ⟩ +≡
long_options[*current_option*].*name* ← *char_to_string*(´x´); *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← *address_of*(*no_xref*); *long_options*[*current_option*].*val* ← 1;
incr(*current_option*);

269* ⟨ Globals in the outer block 9 ⟩ +≡
no_xref: *c_int_type*;
pdf_output: *c_int_type*;

270* An element with all zeros always ends the list.

⟨ Define the option table 265* ⟩ +≡
long_options[*current_option*].*name* ← 0; *long_options*[*current_option*].*has_arg* ← 0;
long_options[*current_option*].*flag* ← 0; *long_options*[*current_option*].*val* ← 0;

271* Global filenames.

⟨ Globals in the outer block 9 ⟩ +≡
web_name, *chg_name*, *tex_name*: *const_c_string*;

272* 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 modules 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 module names are not indexed. Underlined entries correspond to where the identifier was declared. Error messages, control sequences put into the output, and a few other things like "recursion" are indexed here too.

The following sections were changed by the change file: 1, 2, 8, 12, 17, 20, 21, 22, 24, 26, 28, 33, 37, 50, 124, 127, 148, 151, 157, 161, 162, 166, 167, 169, 170, 172, 173, 174, 239, 258, 259, 261, 264, 265, 266, 267, 268, 269, 270, 271, 272.

⟨ Reduce the scraps using the productions until no more rules apply 175 ⟩ Used in section 179.
⟨ Scan a verbatim string 107 ⟩ Used in section 100.
⟨ Scan the module name and make *cur_module* point to it 101 ⟩ Used in section 100.
⟨ Scan to the next `@>` 106 ⟩ Used in section 100.
⟨ Set initial values 10, 14, 17*, 18, 21*, 26*, 41, 43, 49, 54, 57, 94, 102, 124*, 126, 145, 203, 245, 248, 259* ⟩ Used in section 2*.
⟨ Set variable c to the result of comparing the given name to name p 68 ⟩ Used in sections 66 and 69.
⟨ Show cross references to this module 233 ⟩ Used in section 220.
⟨ Skip next character, give error if not '`@`' 215 ⟩ Used in section 214.
⟨ Skip over comment lines in the change file; **return** if end of file 76 ⟩ Used in section 75.
⟨ Skip to the next nonblank line; **return** if end of file 77 ⟩ Used in section 75.
⟨ Sort and output the index 250 ⟩ Used in section 239*.
⟨ Special control codes allowed only when debugging 88 ⟩ Used in section 87.
⟨ Split the list at *sort_ptr* into further lists 251 ⟩ Used in section 250.
⟨ Start a format definition 228 ⟩ Used in section 225.
⟨ Start a macro definition 227 ⟩ Used in section 225.
⟨ Store all the reserved words 64 ⟩ Used in section 261*.
⟨ Store cross reference data for the current module 110 ⟩ Used in section 109.
⟨ Store cross references in the definition part of a module 115 ⟩ Used in section 110.
⟨ Store cross references in the Pascal part of a module 117 ⟩ Used in section 110.
⟨ Store cross references in the TEX part of a module 113 ⟩ Used in section 110.
⟨ Tell about changed modules 241 ⟩ Used in section 239*.
⟨ Translate a hexadecimal constant appearing in TEX text 224 ⟩ Used in section 222.
⟨ Translate an octal constant appearing in TEX text 223 ⟩ Used in section 222.
⟨ Translate the current module 220 ⟩ Used in section 218.
⟨ Translate the definition part of the current module 225 ⟩ Used in section 220.
⟨ Translate the Pascal part of the current module 230 ⟩ Used in section 220.
⟨ Translate the TEX part of the current module 222 ⟩ Used in section 220.
⟨ Types in the outer block 11, 12*, 36, 38, 47, 52, 201 ⟩ Used in section 2*.