The WEAVE processor

(Version 4.5)
1* Introduction. This program converts a WEB file to a \LaTeX{} 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 \LaTeX{}, 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.

\begin{verbatim}
define my_name ≡ 'weave'
define banner ≡ 'This is WEAVE, Version 4.5'
\end{verbatim}

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 \LaTeX{} 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.

\begin{verbatim}
(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;
\end{verbatim}

8* The following parameters are set big enough to handle \LaTeX{}, so they should be sufficient for most applications of WEAVE.

\begin{verbatim}
(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 \LaTeX{} 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\_scrap\_s = 10000; \{ number of tokens in Pascal texts being parsed \}
  stack\_size = 2000; \{ number of simultaneous output levels \}
\end{verbatim}

This code is used in section 2*.
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 \( \text{chr}(\text{first_text_char}) \) through \( \text{chr}(\text{last_text_char}) \), inclusive. The following definitions should be adjusted if necessary.

\[
\begin{align*}
\text{define} & \quad \text{text_char} \equiv \text{ASCII code} \quad \{ \text{the data type of characters in text files} \} \\
\text{define} & \quad \text{first_text_char} = 0 \quad \{ \text{ordinal number of the smallest element of text_char} \} \\
\text{define} & \quad \text{last_text_char} = 255 \quad \{ \text{ordinal number of the largest element of text_char} \}
\end{align*}
\]

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 \TeX{}book, the first line of code in this module should be changed to

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

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 <>.

(Web’s character set is essentially identical to \TeX{}’s, even with respect to characters less than ‘40.)

(The present file \texttt{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.)

\[
\begin{align*}
\text{Set initial values} & \quad \equiv \\
\text{for } i \leftarrow 1 \text{ to } '37 \text{ do } xchr[i] \leftarrow \text{chr}(i);
\end{align*}
\]
20* Terminal output is done by writing on file `term_out`, which is assumed to consist of characters of type `text_char`:

```plaintext
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 ⟩ +≡

```plaintext
{ 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.

```plaintext
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.

```plaintext
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 ⟩ +≡

```plaintext
rewrite(tex_file, tex_name);
```
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 ≤ 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.

```plaintext
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] ≠ "\n" 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
            getc
        loc ← 0; error;
    end;
    end;
read Ln(f); limit ← final_limit; input Ln ← true;
end;
end;
```
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.

```plaintext
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
        ⟨Print the job history 263⟩;
        new_line;
        if (history ≠ spotless) ∧ (history ≠ harmless_message) then uexit(1)
        else uexit(0);
    end;
```
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 pdf\TeX{} 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 \(ww = 1\).

\begin{verbatim}
define \(ww = 3\) { \{ we multiply the byte capacity by approximately this amount \}
⟨Globals in the outer block \(9\)⟩ +≡
byte_mem: packed array [0 .. \(ww - 1\), 0 .. max_bytes] of ASCII_code;  \{ characters of names \}
byte_start: array [0 .. max_names] of sixteen_bits;  \{ directory into byte_mem \}
link: array [0 .. max_names] of sixteen_bits;  \{ hash table or tree links \}
ilk: array [0 .. max_names] of sixteen_bits;  \{ type codes or tree links \}
xref: array [0 .. max_names] of sixteen_bits;  \{ heads of cross-reference lists \}
\end{verbatim}

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.

\begin{verbatim}
define append_xref(#) ≡
   if xref_ptr = max.refs then overflow(`cross_reference')
   else begin incr(xref_ptr); num(xref_ptr) ← #;
   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) ∨ (byte_start[p] + 1 = byte_start[p + \(ww\)]) ∨ (xref_switch = 0)) then return;
   m ← module_count + xref_switch; xref_switch ← 0; q ← xref[p];
if q > 0 then
   begin n ← num(q);
   if (n = m) ∨ (n = m + def_flag) then return
   else if m = n + def_flag then
      begin num(q) ← m; return;
      end;
   end;
append_xref(m); xlink(xref_ptr) ← q; xref[p] ← xref_ptr;
exit: end;
\end{verbatim}
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 ‘\( \text{simp math} \rightarrow \text{math} \)’ would be represented by ‘\((pp, 2, \text{math}, -1)\)’; in this case the pointer \( pp \) should decrease by 1 after the production has been applied, because some productions with \( \text{math} \) in their second positions might now match, but no productions have \( \text{math} \) 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 ‘\( \text{var head math colon} \rightarrow \text{var head intro} \)’, which is represented by ‘\((pp + 1, 2, \text{intro}, +1)\)’; the +1 here is deduced by looking at the grammar and seeing that no matches could possibly occur at positions \( \leq pp \) 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 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 ‘\( \text{reduce} (pp + 1, 2, \text{intro}, +1)(52) \)’ when it implements the production just mentioned.

Before calling \( \text{reduce} \), the program should have appended the tokens of the new translation to the \text{tok_mem} array. We commonly want to append copies of several existing translations, and macros are defined to simplify these common cases. For example, \( \text{app2}(pp) \) will append the translations of two consecutive scraps, \( \text{trans}[pp] \) and \( \text{trans}[pp + 1] \), to the current token list. If the entire new translation is formed in this way, we write ‘\( \text{squash}(j, k, c, d) \)’ instead of ‘\( \text{reduce}(j, k, c, d) \)’. For example, ‘\( \text{squash}(pp, 2, \text{math}, -1) \)’ is an abbreviation for \( \text{app2}(pp); \text{reduce}(pp, 2, \text{math}, -1) \).

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.

\begin{verbatim}
define production(#) ≡
  debug prod(#)
guded;
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)
\end{verbatim}
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.

\[\langle \text{Cases for } \text{alpha } 151^* \rangle \equiv \]
\begin{align*}
\text{if } \text{cat}(pp + 1) & = \text{math} \text{ then} \\
\text{begin } & \text{ if } \text{cat}(pp + 2) = \text{colon} \text{ then } \text{squash}(pp + 1, 2, \text{math}, 0)(1) \\
\text{else if } & \text{cat}(pp + 2) = \text{omega} \text{ then} \\
\text{begin } & \text{app}(pp); \text{app}("\_"); \text{app}("\$"); \text{app}(pp + 1); \text{app}("\$"); \text{app}("\_"); \text{app}(\text{indent}); \\
& \text{app}(pp + 2); \text{reduce}(pp, 3, \text{clause}, -2)(2); \\
\text{end}; \\
\text{end} \text{ else if } \text{cat}(pp + 1) = \text{omega} \text{ then} \\
\text{begin } & \text{app}(pp); \text{app}("\_"); \text{app}(\text{indent}); \text{app}(pp + 1); \text{reduce}(pp, 2, \text{clause}, -2)(3); \\
\text{end} \text{ else if } \text{cat}(pp + 1) = \text{simp} \text{ then } \text{reduce}(pp + 1, 0, \text{math}, 0)(4)
\end{align*}
This code is used in section 150.

\[\langle \text{Cases for } \text{elsie } 157^* \rangle \equiv \]
\begin{align*}
\text{reduce}(pp, 0, \text{intro}, -3)(14)
\end{align*}
This code is used in section 149.

\[\langle \text{Cases for } \text{mod. scrap } 161^* \rangle \equiv \]
\begin{align*}
\text{if } (\text{cat}(pp + 1) = \text{terminator}) \lor (\text{cat}(pp + 1) = \text{semi}) \text{ then} \\
\text{begin } & \text{app}(pp); \text{app}(\text{force}); \text{reduce}(pp, 2, \text{stmt}, -2)(24); \\
\text{end} \text{ else } \text{reduce}(pp, 0, \text{simp}, -2)(25)
\end{align*}
This code is used in section 149.
\[162^* \ (\text{Cases for open 162}^*) \equiv
\]
\[
\begin{aligned}
\text{if } & (\text{cat}[pp + 1] = \text{case.head}) \land (\text{cat}[pp + 2] = \text{close}) \text{ then} \\
& \begin{aligned}
\text{begin } & \text{app1}(pp) \ ; \ \text{app}("\$") \ ; \ \text{app}(\text{cancel}) \ ; \ \text{app1}(pp + 1) \ ; \ \text{app}(\text{cancel}) \ ; \ \text{app}(\text{outdent}) \ ; \ \text{app}("\$") \ ; \\
& \ \text{app1}(pp + 2) \ ; \ \text{reduce}(pp, 3, \text{math}, -1)(26) \\
\end{aligned} \\
\text{end}
\end{aligned}
\]
\[
\text{else if } \text{cat}[pp + 1] = \text{close} \text{ then} \\
\begin{aligned}
\text{begin } & \text{app1}(pp) \ ; \ \text{app}("\"\") \ ; \ \text{app1}(pp + 1) \ ; \ \text{reduce}(pp, 2, \text{math}, -1)(27) \\
\text{end}
\end{aligned}
\]
\[
\text{else if } \text{cat}[pp + 1] = \text{math} \text{ then } \ (\text{Cases for open math 163})
\begin{aligned}
\text{else if } & \text{cat}[pp + 1] = \text{proc} \text{ then} \\
& \begin{aligned}
\text{begin if } & \text{cat}[pp + 2] = \text{intro} \text{ then} \\
& \begin{aligned}
\text{begin } & \text{app}(\text{math_op}) \ ; \ \text{app}(\text{cancel}) \ ; \ \text{app1}(pp + 1) \ ; \ \text{app}("\})") \ ; \ \text{reduce}(pp + 1, 2, \text{math}, 0)(34) \\
\end{aligned} \\
\text{end}
\end{aligned}
\end{aligned}
\]
\[
\text{else if } \text{cat}[pp + 1] = \text{simp} \text{ then } \text{reduce}(pp + 1, 0, \text{math}, 0)(35)
\end{aligned}
\]
\[
\text{else if } (\text{cat}[pp + 1] = \text{stmt}) \land (\text{cat}[pp + 2] = \text{close}) \text{ then} \\
\begin{aligned}
\text{begin } & \text{app1}(pp) \ ; \ \text{app}("\$") \ ; \ \text{app}(\text{cancel}) \ ; \ \text{app1}(pp + 1) \ ; \ \text{app}(\text{cancel}) \ ; \ \text{app}("\$") \ ; \\
& \ \text{app1}(pp + 2) \ ; \ \text{reduce}(pp, 3, \text{math}, -1)(36) \\
\end{aligned}
\]
\[
\text{end}
\]
\[
\text{else if } \text{cat}[pp + 1] = \text{var.head} \text{ then} \\
\begin{aligned}
\text{begin if } & \text{cat}[pp + 2] = \text{intro} \text{ then} \\
& \begin{aligned}
\text{begin } & \text{app}(\text{math_op}) \ ; \ \text{app}(\text{cancel}) \ ; \ \text{app1}(pp + 1) \ ; \ \text{app}("\})") \\
& \ \text{reduce}(pp + 1, 2, \text{math}, 0)(37) \\
\end{aligned} \\
\text{end}
\end{aligned}
\]
\[
\text{end}
\]
This code is used in section 150.

\[166^* \ (\text{Cases for semi 166}^*) \equiv
\begin{aligned}
\text{reduce}(pp, 0, \text{terminator}, -3)(42)
\end{aligned}
\]
This code is used in section 149.

\[167^* \ (\text{Cases for simp 167}^*) \equiv
\begin{aligned}
\text{if } & \text{cat}[pp + 1] = \text{close} \text{ then } \text{reduce}(pp, 0, \text{stmt}, -2)(43) \\
\text{else if } & \text{cat}[pp + 1] = \text{colon} \text{ then} \\
& \begin{aligned}
\text{begin } & \text{app}(\text{force}) \ ; \ \text{app}(\text{backup}) \ ; \ \text{squash}(pp, 2, \text{intro}, -3)(44) \\
\end{aligned} \\
\text{end}
\end{aligned}
\]
\[
\text{else if } \text{cat}[pp + 1] = \text{math} \text{ then } \text{squash}(pp, 2, \text{math}, -1)(45)
\end{aligned}
\]
\[
\text{else if } \text{cat}[pp + 1] = \text{mod.scrap} \text{ then } \text{squash}(pp, 2, \text{mod.scrap}, 0)(46)
\]
\[
\text{else if } \text{cat}[pp + 1] = \text{simp} \text{ then } \text{squash}(pp, 2, \text{simp}, -2)(47)
\]
\[
\text{else if } \text{cat}[pp + 1] = \text{terminator} \text{ then } \text{squash}(pp, 2, \text{stmt}, -2)(48)
\]
This code is used in section 150.

\[169^* \ (\text{Cases for terminator 169}^*) \equiv
\begin{aligned}
\text{reduce}(pp, 0, \text{stmt}, -2)(50)
\end{aligned}
\]
This code is used in section 149.
WEAVE changes for C

IMPLEMENTING THE PRODUCTIONS

\[170^*\]

\(\text{(Cases for } \text{var.head} \ 170^*\) \equiv\)

\[
\begin{align*}
\text{if } & \text{cat} [pp + 1] = \text{beginning} \ \text{then } \text{reduce} (pp, 0, \text{stmt}, -2)(51) \\
\text{else if } & \text{cat} [pp + 1] = \text{math} \ \text{then} \\
& \begin{cases} 
\text{begin if } \text{cat} [pp + 2] = \text{colon} \ \text{then} \\
& \text{app} ("\$") ; \text{app1} (pp + 1) ; \text{app} ("\$") ; \text{app1} (pp + 2) ; \text{reduce} (pp + 1, 2, \text{intro}, +1)(52) \\
& \text{end} ; \\
\text{end} \\
\text{else if } & \text{cat} [pp + 1] = \text{simp} \ \text{then} \\
& \begin{cases} 
\text{begin if } \text{cat} [pp + 2] = \text{colon} \ \text{then} \text{squash} (pp + 1, 2, \text{intro}, +1)(53) ; \\
\text{end} \\
\text{else if } & \text{cat} [pp + 1] = \text{stmt} \ \text{then} \\
& \begin{cases} 
\text{begin} \text{app1} (pp) ; \text{app} (\text{break_space}) ; \text{app1} (pp + 1) ; \text{reduce} (pp, 2, \text{var.head}, -2)(54) ; \\
\text{end} \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\end{align*}
\]

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 \((j : \text{sixteen_bits}; k : \text{eight_bits}; c : \text{eight_bits}; d : \text{integer})\);

\begin{align*}
\text{var} & \ : \ 0 \ldots \text{max_scraps} ; \ \{ \text{index into scrap memory } \} \\
\text{begin} & \ \text{cat} [j] \leftarrow c ; \\
\text{if} & \ k > 0 \ \text{then} \\
& \begin{cases} 
\text{begin} \ \text{trans} [j] \leftarrow \text{text_ptr} ; \ \text{freeze_text} ; \\
\text{end} ; \\
\text{if} & \ k > 1 \ \text{then} \\
& \begin{cases} 
\text{begin} \ \text{for} \ i \leftarrow j + k \ \text{to} \ \text{lo_ptr} \ \text{do} \\
& \begin{cases} 
\text{begin} \ \text{cat} [i - k + 1] \leftarrow \text{cat} [i] ; \ \text{trans} [i - k + 1] \leftarrow \text{trans} [i] ; \\
\text{end} ; \\
\text{lo_ptr} \leftarrow \text{lo_ptr} - k + 1 ; \\
\text{end} ; \\
\langle \text{Change } pp \ \text{to} \ \text{max} (\text{scrap_base}, pp + d) \ 173^* \rangle ; \\
\text{end} ; \\
\end{cases} \\
\end{cases} \\
\end{cases} \\
\text{end} ; \\
\end{align*}

\[173^*\]

\(\langle \text{Change } pp \ \text{to} \ \text{max} (\text{scrap_base}, pp + d) \ 173^* \rangle \equiv \)

\[
\begin{align*}
\text{if} & \ pp + d \geq \text{scrap_base} \ \text{then} \ pp \leftarrow pp + d \\
\text{else} & \ pp \leftarrow \text{scrap_base} \\
\end{align*}
\]

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 \((j : \text{sixteen_bits}; k : \text{eight_bits}; c : \text{eight_bits}; d : \text{integer})\);

\begin{align*}
\text{begin} & \ \text{case} \ k \ \text{of} \\
1 : & \begin{cases} 
\text{begin} \ \text{app1} (j) ; \ \text{end} ; \\
2 : \begin{cases} 
\text{begin} \ \text{app2} (j) ; \ \text{end} ; \\
3 : \begin{cases} 
\text{begin} \ \text{app3} (j) ; \ \text{end} ; \\
\text{othercases} \ \text{confusion} (\text{"squash"}) \\
\text{endcases} ; \\
\text{red} (j, k, c, d) ; \\
\end{cases} \\
\text{endcases} ; \\
\end{cases} \\
\end{cases} \\
\end{align*}

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 \texttt{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.

\begin{verbatim}
⟨Phase III: Output the cross-reference index 239∗⟩ ≡
if \texttt{no_xref} then
  begin finish_line; out("\n"); out5("v")("f")("i")("l")("l"); out4("\n")("e")("n")("d"); finish_line;
  end
else begin phase_three ← true; print_nl(\texttt{\`Writing\ the\ index...});
  if change_exists then
    begin finish_line; (Tell about changed modules 241);
    end;
    finish_line; out4("\n")("i")("n")("x"); finish_line; (Do the first pass of sorting 243);
    (Sort and output the index 250);
    out4("\n")("f")("i")("n"); finish_line; (Output all the module names 257);
    out4("\n")("c")("o")("n"); finish_line;
  end;
  print(\texttt{\`Done.\});
\end{verbatim}

This code is used in section 261∗.
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 \( ddt \leq 0 \); 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.

```plaintext
define term_in ≡ stdin
=set initial values
debug trouble_shooting ← true; debug_cycle ← 1; debug_skipped ← 0; tracing ← 0;
        trouble_shooting ← false; debug_cycle ← 99999;
defined
```

The debugging routine needs to read from the user’s terminal.

```plaintext
define term_in ≡ stdin
=set initial values
debug trouble_shooting ← true; debug_cycle ← 1; debug_skipped ← 0; tracing ← 0;
        trouble_shooting ← false; debug_cycle ← 99999;
defined
```
261* The main program. Let’s put it all together now: \texttt{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.

\begin{verbatim}
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 \textsc{TeX} form\ 218⟩;
    end;

begin initialize;  { beginning of the main program }
    print(banner);  { print a “banner line” }
    printLn(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.
\end{verbatim}
§264* System-dependent changes. Parse a Unix-style command line.

```c
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_uint_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(WEAVEHELP, nil);
            end
      else if argument_is(´version`) then
        begin print_version_and_exit(banner, nil, ´D.E. Knuth´, nil);
            end
  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 writeLn(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.

```c
(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*, and 269*.

This code is used in section 264*.

266* Another of the standard options.

```c
(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* Omit cross-referencing?

\(\langle\text{Define the option table } 265^*\rangle \equiv\)

\[
\begin{align*}
&\text{long_options}[\text{current_option}].\text{name} \leftarrow \text{char_to_string}(\text{"x"}); \text{long_options}[\text{current_option}].\text{has_arg} \leftarrow 0; \\
&\text{long_options}[\text{current_option}].\text{flag} \leftarrow \text{address_of(\text{no_xref})}; \text{long_options}[\text{current_option}].\text{val} \leftarrow 1; \\
&\text{incr(\text{current_option})};
\end{align*}
\]

268* (Globals in the outer block \(9\)) \equiv

\[\text{no_xref}: \text{c_int_type};\]

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

\(\langle\text{Define the option table } 265^*\rangle \equiv\)

\[
\begin{align*}
&\text{long_options}[\text{current_option}].\text{name} \leftarrow 0; \text{long_options}[\text{current_option}].\text{has_arg} \leftarrow 0; \\
&\text{long_options}[\text{current_option}].\text{flag} \leftarrow 0; \text{long_options}[\text{current_option}].\text{val} \leftarrow 0;
\end{align*}
\]

270* Global filenames.

\(\langle\text{Globals in the outer block } 9\rangle \equiv\)

\[\text{web_name, chg_name, tex_name: const c_string};\]
§271 Index. If you have read and understood the code for Phase III above, you know what is in this index and how it got here. All 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, 148, 151, 157, 161, 162, 166, 167, 169, 170, 172, 173, 174, 239, 258, 259, 261, 264, 265, 266, 267, 268, 269, 270, 271.

- help: 265*
- version: 266*
- x: 267*
\): 186.
\*: 130.
\,: 162*, 163, 186.
\.: 189, 253.
\:: 252, 256.
\=: 189.
\[: 254.
\#: 186, 189.
\$: 186, 189.
\%: 186, 189.
\&: 189, 209, 253.
\^: 189.
\^: 189, 209, 253.
\^: 186, 189.
\{: 189.
\{: 189.
\{: 189.
\{: 189, 193.
\[: 186.
\[: 209, 253.
\_: 131, 189.
\A: 236.
\As: 237.
\ast: 186.
\B: 186.
\C: 198.
\con: 239*.
\D: 227.
\E: 186.
\ET: 237.
\ETs: 237.
\F: 228.
\fi: 238.
\fin: 239*
\G: 188.
\H: 196, 224.
\I: 188.
\in: 186.
\input webmac: 124.
\inx: 239*. \\J: 186.
\K: 188.
\L: 188.
\M: 221.
\N: 221.
\O: 196, 223.
\P: 212, 226.
\R: 188.
\S: 188, 228, 231.
\T: 186.
\to: 186.
\U: 236.
\Us: 237.
\V: 188.
\W: 188.
\X: 213.
\Y: 212, 219, 226, 231.
\A: 211, 212.
\B: 211, 212.
\C: 211, 212.
\D: 211, 212.
\E: 211, 212.
\F: 211, 212.
\G: 253.
\H: 88, 177.
\I: 88, 177.
a: 130, 206, 208.
address_of: 264*, 267*
alpha: 140, 142, 143, 149, 192, 193.
alpha_cases: 149, 150.
Ambiguous prefix: 69.
and_sign: 15, 64, 188.
app_comment: 183, 184, 195, 197, 198, 226.
app_hex: 183, 186, 196.
app_octal: 183, 186, 196.
append_xref: 50*, 51.
case_like: 42, 64, 193.
casey: 140, 142, 143, 149, 153, 165, 193.

Change file ended...: 77, 79, 84.
Change file entry did not match: 85.
change_buffer: 73, 74, 75, 78, 79, 85.
change_changing: 72, 79, 81, 84.
change_exists: 45, 109, 110, 239*.
change_file: 2* 23, 24* 32, 71, 73, 76, 77, 79, 84.
change_limit: 73, 74, 75, 78, 79, 83, 85.
change_pending: 71, 79, 84.
changed_module: 45, 71, 79, 84, 109, 110, 130, 241.
changing: 32, 71, 72, 73, 75, 79, 81, 82, 85, 110.
char: 12* 14.
char_like: 42, 64, 191.
char_to_string: 264* 267*.
check_change: 79, 83.
check_sum: 86, 87, 186, 222.
chg_name: 24* 264* 270*.
chr: 12* 13, 17* 18.
clause: 140, 142, 143, 149, 151* 153, 154, 156, 193.
cmdline: 264*.
collate: 246, 247, 248, 249.
colon: 140, 142, 143, 148* 151* 160, 163, 167* 170* 186.

Control codes are forbidden...: 106.
Control text didn't end: 106.
copy_code: 87, 88, 90, 93, 100, 135.
copy_comment: 132, 136, 198.
copy_limbo: 132, 218.
copy_TeX: 132, 134, 222.
count: 69.
cur_bank: 244, 251, 262.
cur_bye: 244, 251.
cur_depth: 244, 250, 251.
cur_end: 201, 202, 204, 205, 206.
cur_mod_name: 208, 214, 215, 216.
cur_mode: 201, 202, 204, 206, 208, 211, 212.
cur_name: 63, 206, 209, 213, 214, 242, 243, 251, 252, 253, 255.
cur_state: 202, 204, 205.
cur_tok: 201, 202, 204, 205, 206.
cur_val: 244, 254.
current_option: 264*265*266*267*269*.
d: 95, 127, 172*174*249.
"d": 258*260.
ddt: 258*260.
debug: 3, 4, 30, 31, 88, 95, 140, 146, 148*177, 178, 179, 181, 182, 206, 252, 258*259*260.
decr: 6, 28*92, 98, 103, 122, 127, 130, 135, 137, 138, 205, 251, 252.
definition: 86, 87, 115, 225.
depth: 244, 249.
dig: 129, 130.
div_like: 42, 64, 191, 192.
do_like: 42, 64, 191, 192.
do_nothing: 6, 95, 113, 149, 150, 186, 189, 222, 253, 264*.
done: 5, 75, 76, 90, 91, 92, 95, 103, 104, 122, 134, 135, 136, 137, 175, 179, 236, 237.
Double & required...: 133.
Double & should be used...: 189.
double_dot: 86, 97, 186.
easy_cases: 183, 185, 186.
eight_bits: 36, 58, 87, 90, 91, 95, 108, 112, 122, 134, 136, 140, 144, 172*174*178, 198, 206, 208, 244, 249.
else: 7.
else_like: 42, 64, 193.
else: 140, 142, 143, 149, 156, 193.
emit_space_if_needed: 219, 225, 230.
end: 3, 7.
end_comment: 86, 87, 97, 186, 222.
end_field: 201, 202.
end_like: 42, 64, 193.
end_translation: 141, 147, 201, 207, 208, 212.
endcases: 7.
eof: 28*.
eoln: 28*.
equal: 66, 67, 68.
equivalence_sign: 15, 97, 116, 188, 228, 231.
error: 28*, 31, 33*.
error_message: 9, 263.
exit: 5, 6, 50*74, 75, 79, 89, 111, 123, 127, 132, 183, 208, 236, 260.
exp: 140, 142, 143, 149, 186.
exponent: 93, 98, 186.
extend_filename: 264*.
extension: 66, 68, 69.
Extra }: 95.
Extra &>: 87.
f: 28*.
false: 28*29, 72, 73, 74, 79, 81, 84, 94, 96, 109, 122, 123, 127, 218, 238, 259*.
fatal_message: 33*34, 35.
fatal_message: 9, 263.
flush: 22*.
final_limit: 28*.
finish_line: 123, 124, 132, 134, 135, 212, 218, 226, 236, 238, 239*254, 256.
first_text_char: 12*18.
first_xref: 234, 235.
five_cases: 149, 150.
flag: 236, 237, 265*266*267*269*.
flush_buffer: 122, 123, 127, 128, 218, 238.
footnote: 233, 236, 256.
for_like: 42, 64, 191, 192.
force_line: 86, 87, 186.
form_feed: 15, 28*.
format: 86, 87, 186.
forward: 30, 207.
found: 5, 58, 60, 61, 66, 95, 96, 122, 148*149, 150, 151*175, 179, 208, 216.
freeze_text: 171, 172*180, 184, 195, 198, 208.
get_line: 71, 82, 89, 90, 91, 95, 103, 123, 132, 134, 136.
get_output: 206, 207, 208, 211, 212.
getc: 28*.
getopt: 264*.
getopt_long_only: 264*.
getopt_return_val: 264*.
getopt_struct: 264*.
goto_like: 42, 64, 191, 192.
greater: 66, 68, 69.
greater_or_equal: 15, 97, 188.
gubed:  3
h:  56, 58, 242.
harmless_message:  0, 33* 263.
has_arg:  265*, 266*, 267*, 269*
hash:  38, 55, 57, 60, 242, 243.
hash_size:  3*, 55, 56, 57, 58, 59, 242, 243.
head:  244, 249, 250, 251, 252.
hex:  86, 87, 100, 186, 222.
hi_ptr:  144, 176, 178, 179.
history:  0, 10, 33* 263.
Hmm... n of the preceding...:  80.
i:  16, 58, 172* 179.
id_first:  55, 58, 59, 61, 62, 63, 93, 98, 99, 106, 107, 189, 190.
id_flag:  146, 192, 206, 227, 228.
id_loc:  55, 58, 59, 61, 62, 64, 93, 98, 99, 106, 107, 189, 190.
id_lookup:  55, 58, 63, 93, 111, 113, 116, 191, 227, 228.
identifier:  93, 98, 111, 113, 116, 185, 206, 208, 209, 227, 228.
id2:  63, 64.
id3:  63, 64.
id4:  63, 64.
id5:  63, 64.
id6:  63, 64.
id7:  63, 64.
id8:  63, 64.
id9:  63, 64.
if_like:  42, 64, 193.
if_module_start_then_make_change_pending:  79, 84.
ignore:  86, 87, 88, 186.
ii:  71, 85.
ilk:  36, 37* 42, 43, 55, 58, 60, 62, 111, 116, 191, 192, 253.
Illegal control code...:  215.
Illegal use of @...:  137.
Improper format definition:  228.
Improper macro definition:  227.
Incompatible section names:  66.
infinity:  249, 250.
init_stack:  202, 222, 225, 230, 256.
initialize:  2* 261*
inner:  200, 201, 206, 212.
inner_tok_flag:  146, 198, 206, 207.
Input ended in mid-comment:  136.
Input ended in section name:  103.
Input line too long:  28*
integer:  14, 71, 79, 121, 130, 172*, 174*, 219, 258*, 260, 264*
interval_like:  42.
join:  86, 87, 186, 222.
jump_out:  2* 33* 261*
k_limit:  208, 214, 216.
k_module:  240, 241.
kpse_open_file:  24*
kpse_set_program_name:  2*
kpse_web_format:  24*
l:  31, 58, 66, 69.
last_text_char:  12* 18.
lbrace:  146, 147.
left_angle:  15, 97, 188.
length:  38, 60, 209, 253.
less:  66, 67, 68, 69.
less_or_equal:  15, 97, 188.
lhs:  114, 116.
Line had to be broken:  128.
line_break:  86, 87, 186, 222.
line_feed:  15, 28*.
line_length:  8* 121, 122, 125, 127.
lines_dont_match:  74, 79.
link:  36, 37* 38, 43, 60, 243.
lo_ptr:  144, 172*, 175, 176, 178, 179, 180, 181.
long_buf_size:  8*, 27, 28*, 31, 55, 58, 71, 179, 183, 208, 216, 217.
long_options:  264*, 265*, 266*, 267*, 269*.
longest_name:  8*, 65, 66, 69, 95, 103, 105.
loop:  6.
loop_like:  42, 64, 193.
m:  50*, 130.
make_output: 207, 208, 213, 226, 256.
mark_error: 9, 31, 215, 216.
mark_fatal: 9, 33*.
meth: 139, 140, 142, 143, 148*, 150, 151*, 158,
  160, 162*, 163, 167*, 170* 179, 180, 186, 188,
  192, 227, 228, 231.
meth_bin: 141, 142, 147, 192, 208, 210.
meth_break: 86, 87, 186, 222.
meth_op: 141, 143, 147, 162*, 163, 208.
meth_rel: 141, 142, 146, 147, 192, 208, 210, 231.
max_bytes: 8*, 37*, 39, 44, 58, 62, 66, 67, 69,
  131, 208, 244.
max_modules: 8*, 45, 46, 110, 240.
max_refs: 8*, 47, 50*.
max_scr_ptr: 144, 145, 187, 197, 199, 226, 262.
max_scraps: 8*, 144, 172*, 178, 179, 187, 197,
  199, 244.
max_sort_ptr: 244, 245, 249, 262.
max_sorts: 244, 249.
max_stack_ptr: 202, 203, 204, 262.
max_texts: 8*, 52, 175, 179, 187, 199.
max_toks: 8*, 53, 136, 146, 175, 179, 180, 187,
  189, 198, 199.
mid_xref: 234, 235.
Missing "|"...: 197.
mod_check: 119, 120.
mod_flag: 146, 206, 231, 232, 256.
mod_name: 206, 208.
mod_print: 256, 257.
mod_scrap: 140, 142, 143, 149, 167*, 231, 232.
mod_text: 65, 66, 67, 68, 69, 95, 101, 102, 103,
  104, 105, 260.
mod_xref_switch: 46, 48, 49, 51, 117.
mode: 201, 208.
mode_field: 201, 202.
module_count: 45, 50*, 51, 71, 79, 84, 109, 110,
  181, 218, 220, 221, 231, 241.
module_name: 86, 87, 93, 100, 113, 117, 222,
  230, 232.
my_name: 1*, 2*, 264*.
n: 50*, 79, 178.
n_options: 264*.
name: 264*, 265*, 266*, 267*, 269*.
Name does not match: 69.
name_ptr: 38, 39, 41, 44, 58, 60, 62, 67, 262.
Never defined: <section name>: 119.
Never used: <section name>: 119.
new_line: 20*, 31, 32, 33*, 128.
new_mod_xref: 51, 117.
new_module: 86, 87, 90, 95, 134.
next_control: 108, 111, 112, 113, 115, 116, 117,
  183, 185, 186, 189, 191, 197, 198, 207, 222,
  225, 227, 228, 229, 230, 231, 232.
next_name: 242, 243, 251.
next_xref: 234, 235, 255.
nil: 6.
nil_like: 42, 64, 191, 192.
no_line_break: 86, 87, 186, 222.
no_underline: 86, 87, 100, 113.
no_xref: 50*, 239*, 267*, 268*.
normal: 42, 58, 60, 111, 116, 191, 192, 227,
  228, 253.
not_equal: 15, 97, 188.
not_found: 5.
not_sign: 15, 64, 188.
um: 46, 49, 50*, 51, 119, 213, 231, 235, 236,
  237, 254.
um_field: 46, 48.
octal: 86, 87, 186, 222.
omega: 140, 142, 143, 151*, 192, 193, 195.
oot: 125.
oot1: 125.
oot2: 125.
oot3: 125.
oot4: 125.
oot5: 125.
open: 139, 140, 142, 143, 150, 186.
open_input: 24*. 81.
opt: 139, 141, 142, 143, 147, 159, 163, 186,
  208, 211.
optind: 264*.
option_index: 264*.
or_sign: 15, 64, 188.
ord: 13.
other_line: 71, 72, 81, 85.
othercases: 7.
others: 7.
out: 125, 130, 131, 133, 135, 208, 209, 210,
  211, 212, 213, 214, 221, 222, 223, 224, 236,
out_buf: 121, 122, 124, 125, 126, 127, 128, 212,
  226, 231, 260.
out_line: 121, 122, 124, 128, 219.
out_mod: 130, 213, 221, 237, 241, 254.
out_name: 131, 209, 253.
Pascal
pseudo
print
prefix
Phase
param:

outdent:  141, 143, 147, 153, 155, 156, 160, 162,* 163, 164, 208.
outer:  200, 201, 202, 211, 212.
outer_parens:  198, 225, 230.
output_Pascal:  207, 214, 222.
out2:  125, 130, 210, 211, 212, 213, 219, 221, 226, 237, 241, 252, 253, 254, 256.
out3:  125, 210, 223, 224, 237, 238.
out4:  125, 226, 239* 241.
out5:  125, 210, 239*.

param:  86.
pars_arguments:  2* 264*.
Pascal text...didn't end:  216.
Pascal_parens:  183, 186, 192, 196, 197, 198.
Pascal_translate:  197, 198, 207.
percent:  122.
Phase_I:  261*.
Phase_II:  261*.
phase_three:  29, 109, 213, 239*.
pop_level:  205, 206.
prefix:  66, 68.
prime_the_change_buffer:  75, 81, 84.
print_cat:  140, 178, 181, 260.
print_IN:  20* 32, 128, 181, 261*.
print_text:  146, 260.
print_version_and_exit:  264*.
proc:  140, 142, 143, 149, 162*, 163, 164, 193.
proc_like:  42, 64, 111, 193.
prod:  148*, 178, 183.
production:  148*.
productions, table of:  143.
pseudo_semi:  86, 87, 186, 222.
push_level:  204, 206, 208.
q:  50* 51, 66, 69, 198, 236.
r:  51, 69, 146.
rbbrace:  146.
read:  260.
read_in:  28*.
record_head:  140, 142, 143, 149, 193.
record_like:  42, 64, 193.
recursion:  119, 207, 256.
red:  148*, 172*, 174*.
repeat_like:  42, 64, 193.
res_flag:  146, 192, 193, 206.
res_word:  206, 208, 209.
reserved:  42, 50* 60.
reset_input:  81, 109, 218.
restart:  5, 82, 95, 100, 206.
reswitch:  5, 183, 185, 191, 208, 212.
return:  5, 6*.
rewrite:  26*.
rhs:  114, 116.
roman:  42, 111, 253.
root:  43, 66, 69, 120, 257.
save_base:  197.
save_limit:  208, 214.
save_line:  219.
save_loc:  208, 214.
save_mode:  208, 212.
save_next_control:  207.
save_place:  219.
save_text_ptr:  207.
save_tok_ptr:  207.
scan_digit:  93, 94, 95, 96, 100.
scrap_prn:  144, 145, 176, 178, 179, 183, 184, 187, 194, 195, 197, 199, 226, 228, 244, 256.
sco:  184, 186, 193, 194, 195, 228.
scl:  184, 186, 189, 190, 192, 193, 196, 227, 228, 231, 232.
sco2:  184, 186, 188, 192, 193, 227, 228, 231.
sco3:  184, 186, 192, 193, 231.
sco4:  184, 186, 193.
Section name didn't end:  104.
Section name too long:  105.
semi:  139, 140, 142, 143, 149, 161*, 163, 186, 194, 195, 228, 231.
set_element_sign:  15, 64, 186.
sid1:  63.
sid2:  63.
sid3:  63.
sid4:  63.
sid5:  63.
sid6:  63.
sid7:  63.
sid8:  63.
sid9:  63.
skip_limbo:  89, 109, 132.
skip_TeX:  90, 113, 132.
Sorry, x capacity exceeded:  35.
sort_ptr:  244, 249, 250, 251, 252.
special string characters:  189.
split procedures:  149, 183, 261*.
spoilless:  9, 10, 33*, 263.
sq:  148*, 174*.
stack:  201, 202, 204, 205.
stack_ptr:  201, 202, 204, 205.
stack_size:  8*, 202, 204.
stat:  3.
stderr:  33*, 264*
stdin:  258*
stdout:  20*.
str_cmp:  264*.
string:  93, 99, 185.
String constant didn’t end:  99.
string_delimiter:  208, 216.
sub_cases:  183, 191, 192.
s0:  184.
s1:  184.
s2:  184.
s3:  184.
s4:  184.
t:  58.
tab_mark:  15, 32, 79, 87, 89, 92, 95, 103, 104, 123, 133, 135.
tats:  3.
temp_line:  71, 72.
term_in:  258*, 260.
term_out:  20*, 22*.
TeX string should be...:  222.
tex_file:  2*, 25, 26*, 122, 124.
tex_name:  26*, 264*, 270*.
TeX_string:  86, 87, 93, 100, 185, 222.
text_char:  12*, 14, 20*.
text_pointer:  52, 53, 144, 146, 179, 197, 198, 204, 207, 226.
thin_space:  86, 87, 186, 222.
This can’t happen:  34.
this_xref:  234, 235, 255.
to_like:  42, 64, 191, 192.
tok_field:  201, 202.
tok_start:  52, 53, 54, 144, 146, 171, 204.
tracing:  88, 177, 178, 181, 182, 259*.
trans:  144, 148*, 172*, 176, 179, 183, 184, 195, 197, 244.
translate:  149, 179, 180, 197, 226.
trouble_shooting:  95, 206, 252, 258*, 259*.
true:  6, 28*, 29, 71, 72, 74, 79, 81, 83, 84, 85, 93, 100, 109, 110, 122, 127, 128, 239*, 258*, 259*.
typewriter:  42, 111, 253.
xexit:  33*.
unbucket:  249, 250, 251.
underline:  86, 87, 100, 113.
Unknown control code:  87.
until_like:  42, 64, 193.
up_to:  95.
usage:  264*.
usage_help:  264*.
val:  265*, 266*, 267*, 269*.
var_like:  42, 64, 111, 193.
verbatim:  86, 87, 100, 107, 185, 189.
Verbatim string didn’t end:  107.
version_string:  261*.
vgetc:  28*.
w:  44, 58, 66, 69, 131, 208.
WEAVE:  2*.
WEAVE_HELP:  264*.
WEB file ended:  79.
web_name: 24*264*, 270*
webmac: 124.
Where is the match...: 76, 80, 84.
wi: 40, 41.
wildcard: 42, 111, 253.
write: 20*, 33*, 122, 124.
write_in: 20*, 122, 264*
ww: 8*37*38, 39, 40, 41, 44, 50*, 58, 61, 62, 66, 67, 68, 69, 131, 208, 209, 214, 243, 244, 251, 262.
xchr: 13, 14, 16, 17*, 18, 32, 44, 105, 122, 128, 146, 147, 182, 260.
xclause: 6.
xlink_field: 46, 48.
xmem: 46, 48.
xord: 13, 16, 18, 28*
xref_number: 47, 48, 50*, 51, 118, 234, 236.
xref_ptr: 46, 48, 49, 50*, 51, 262.
xref_roman: 86, 87, 93, 100, 111, 113, 186, 222.
xref_switch: 46, 48, 49, 50*, 93, 100, 101, 111, 113, 115.
xref_typewriter: 86, 87, 93, 111, 113, 186, 222.
xref_wildcard: 86, 87, 93, 111, 113, 186, 222.
You can’t do that...: 222, 232.
You need an = sign...: 231.
Append a string scrap 189 Used in section 185.
Append a \TeX{} string scrap 190 Used in section 185.
Append an identifier scrap 191 Used in section 185.
Append the scrap appropriate to next_control 185 Used in section 183.
Append terminator if not already present 194 Used in sections 193, 193, and 193.
Cases for alpha 151 Used in section 150.
Cases for beginning 152 Used in section 150.
Cases for case_head 153 Used in section 149.
Cases for casey 154 Used in section 149.
Cases for cond 156 Used in section 149.
Cases for else 157 Used in section 149.
Cases for exp 158 Used in section 149.
Cases for intro 159 Used in section 150.
Cases for math 160 Used in section 150.
Cases for mod_scrap 161 Used in section 149.
Cases for open math 163 Used in section 162.
Cases for open 162 Used in section 150.
Cases for proc 164 Used in section 149.
Cases for record_head 165 Used in section 149.
Cases for semi 166 Used in section 149.
Cases for simp 167 Used in section 150.
Cases for stmt 168 Used in section 149.
Cases for terminator 169 Used in section 149.
Cases for var_head 170 Used in section 149.
Cases involving nonstandard ASCII characters 188 Used in section 186.
Cases that generate more than one scrap 193 Used in section 191.
Change pp to max(scrap_base, pp+d) 173 Used in section 172.
Check for overlong name 105 Used in section 103.
Check that all changes have been read 85 Used in section 261.
Check that = or \equiv follows this module name, and emit the scraps to start the module definition 231
Used in section 230.
Clear bal and goto done 138 Used in sections 136 and 137.
Combine the irreducible scraps that remain 180 Used in section 179.
Compare name p with current identifier, goto found if equal 61 Used in section 60.
Compiler directives 4 Used in section 2.
Compress two-symbol combinations like `:=\;' 97 Used in section 95.
Compute the hash code h 59 Used in section 58.
Compute the name location p 60 Used in section 58.
Constants in the outer block 8 Used in section 2.
Copy a control code into the buffer 217 Used in section 216.
Copy special things when c = "@", "\n", "\{", "\}"; goto done at end 137 Used in section 136.
Copy the Pascal text into buffer[(limit + 1) \ldots j] 216 Used in section 214.
Copy up to \`{1}' or control code, goto done if finished 135 Used in section 134.
Copy up to control code, return if finished 133 Used in section 132.
Declaration of subprocedures for translate 150 Used in section 179.
Declaration of the app_comment procedure 195 Used in section 183.
Declaration of the app_octal and app_hex procedures 196 Used in section 183.
Declaration of the easy_cases procedure 186 Used in section 183.
Declaration of the sub_cases procedure 192 Used in section 183.
Define the option table 265, 266, 267, 269 Used in section 264.
Define parse_arguments 264 Used in section 2.
Phase I: Read all the user’s text and store the cross references 109  Used in section 261*. 
Phase II: Read all the text again and translate it to \TeX form 218  Used in section 261*. 
Phase III: Output the cross-reference index 239*  Used in section 261*. 
Print error location based on input buffer 32  Used in section 31. 
Print error messages about unused or undefined module names 120  Used in section 109. 
Print statistics about memory usage 262  Used in section 33*. 
Print the job history 263  Used in section 33*. 
Print token r in symbolic form 147  Used in section 146. 
Print warning message, break the line, return 128  Used in section 127. 
Process a format definition 116  Used in section 115. 
Put module name into mod_text[1 . . k] 103  Used in section 101. 
Read from change_file and maybe turn off changing 84  Used in section 82. 
Read from web_file and maybe turn on changing 83  Used in section 82. 
Rearrange the list pointed to by cur_xref 235  Used in section 233.
(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 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*.
(Skip over comment lines in the change file; return if end of file 76) Used in section 75.
(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*.