The TWILL processor

(Version 4.5)

1. Introduction. [Apology: This modification of WEAVE was put together hastily at the end of September, 1985, in order to prepare the listing of TEX, on the WAITS system. No attempt has been made to polish anything or to make this code usable by anyone else but its author.]

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␣TWILL,␣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;
```

3. Some of this code is optional for use when debugging only; such material is enclosed between the delimiters **debug** and **gubed**. Other parts, delimited by **stat** and **tats**, are optionally included if statistics about WEAVE's memory usage are desired.

```
define debug ≡ @{   { change this to 'debug ≡ ' when debugging }
define gubed ≡ @}   { change this to 'gubed ≡ ' when debugging }
format debug ≡ begin
format gubed ≡ end
define stat ≡ @{   { change this to 'stat ≡ ' when gathering usage statistics }
define tats ≡ @}   { change this to 'tats ≡ ' when gathering usage statistics }
format stat ≡ begin
format tats ≡ end
```

4. The Pascal compiler used to develop this system has "compiler directives" that can appear in comments whose first character is a dollar sign. In production versions of `WEAVE` these directives tell the compiler that it is safe to avoid range checks and to leave out the extra code it inserts for the Pascal debugger's benefit, although interrupts will occur if there is arithmetic overflow.

⟨ Compiler directives 4 ⟩ ≡
  `@{@&$`$C-, A+, D-$`@}` { no range check, catch arithmetic overflow, no debug overhead }
  debug `@{@&$`$C+, D+$`@}` **gubed** { but turn everything on when debugging }
This code is used in section 2.

5. Labels are given symbolic names by the following definitions. We insert the label '*exit*:' just before the '**end**' of a procedure in which we have used the '**return**' statement defined below; the label '*restart*' is occasionally used at the very beginning of a procedure; and the label '*reswitch*' is occasionally used just prior to a **case** statement in which some cases change the conditions and we wish to branch to the newly applicable case. Loops that are set up with the **loop** construction defined below are commonly exited by going to '*done*' or to '*found*' or to '*not_found*', and they are sometimes repeated by going to '*continue*'.

define *exit* = 10 { go here to leave a procedure }
define *restart* = 20 { go here to start a procedure again }
define *reswitch* = 21 { go here to start a case statement again }
define *continue* = 22 { go here to resume a loop }
define *done* = 30 { go here to exit a loop }
define *found* = 31 { go here when you've found it }
define *not_found* = 32 { go here when you've found something else }

6. Here are some macros for common programming idioms.

define *incr*(#) ≡ # ← # + 1 { increase a variable by unity }
define *decr*(#) ≡ # ← # − 1 { decrease a variable by unity }
define *loop* ≡ **while** *true* **do** { repeat over and over until a **goto** happens }
define *do_nothing* ≡ { empty statement }
define *return* ≡ **goto** *exit* { terminate a procedure call }
format *return* ≡ *nil*
format *loop* ≡ *xclause*

7. We assume that **case** statements may include a default case that applies if no matching label is found. Thus, we shall use constructions like

case x **of**
1: ⟨ code for $x = 1$ ⟩;
3: ⟨ code for $x = 3$ ⟩;
othercases ⟨ code for $x \neq 1$ and $x \neq 3$ ⟩
endcases

since most Pascal compilers have plugged this hole in the language by incorporating some sort of default mechanism. For example, the compiler used to develop `WEB` and TEX allows '*others*:' as a default label, and other Pascals allow syntaxes like '**else**' or '**otherwise**' or '*otherwise*:', etc. The definitions of **othercases** and **endcases** should be changed to agree with local conventions. (Of course, if no default mechanism is available, the **case** statements of this program must be extended by listing all remaining cases.)

define *othercases* ≡ *others*: { default for cases not listed explicitly }
define *endcases* ≡ **end** { follows the default case in an extended **case** statement }
format *othercases* ≡ *else*
format *endcases* ≡ *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 }
max_new_refs = 200;  { number of different references to other modules }
```
This code is used in section 2.

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

```
define spotless = 0  { history value for normal jobs }
define harmless_message = 1  { history value when non-serious info was printed }
define error_message = 2  { history value when an error was noted }
define fatal_message = 3  { history value when we had to stop prematurely }

define mark_harmless ≡ if history = spotless then history ← harmless_message
define mark_error ≡ history ← error_message
define mark_fatal ≡ history ← fatal_message
```

⟨ Globals in the outer block 9 ⟩ ≡
```
history: spotless .. fatal_message;  { how bad was this run? }
```
See also sections 13, 23, 25, 27, 29, 37, 39, 45, 48, 53, 55, 63, 65, 71, 73, 93, 108, 114, 118, 121, 129, 144, 177, 202, 219, 229, 234, 240, 242, 244, 246, 258, 268, 270, and 279.

This code is used in section 2.

10. ⟨ Set initial values 10 ⟩ ≡
```
history ← spotless;
```
See also sections 14, 17, 18, 21, 26, 41, 43, 49, 54, 57, 94, 102, 124, 126, 145, 203, 245, 248, and 259.

This code is used in section 2.

11. The character set. One of the main goals in the design of WEB has been to make it readily portable between a wide variety of computers. Yet WEB by its very nature must use a greater variety of characters than most computer programs deal with, and character encoding is one of the areas in which existing machines differ most widely from each other.

To resolve this problem, all input to WEAVE and TANGLE is converted to an internal eight-bit code that is essentially standard ASCII, the "American Standard Code for Information Interchange." The conversion is done immediately when each character is read in. Conversely, characters are converted from ASCII to the user's external representation just before they are output. (The original ASCII code was seven bits only; WEB now allows eight bits in an attempt to keep up with modern times.)

Such an internal code is relevant to users of WEB only because it is the code used for preprocessed constants like "A". If you are writing a program in WEB that makes use of such one-character constants, you should convert your input to ASCII form, like WEAVE and TANGLE do. Otherwise WEB's internal coding scheme does not affect you.

Here is a table of the standard visible ASCII codes:

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
′040	␣	!	"	#	$	%	&	’
′050	(	)	*	+	,	-	.	/
′060	0	1	2	3	4	5	6	7
′070	8	9	:	;	<	=	>	?
′100	@	A	B	C	D	E	F	G
′110	H	I	J	K	L	M	N	O
′120	P	Q	R	S	T	U	V	W
′130	X	Y	Z	[	\	]	^	_
′140	‘	a	b	c	d	e	f	g
′150	h	i	j	k	l	m	n	o
′160	p	q	r	s	t	u	v	w
′170	x	y	z	{	|	}	~	

(Actually, of course, code *′040* is an invisible blank space.) Code *′136* was once an upward arrow (↑), and code *′137* was once a left arrow (←), in olden times when the first draft of ASCII code was prepared; but WEB works with today's standard ASCII in which those codes represent circumflex and underline as shown.

⟨ Types in the outer block 11 ⟩ ≡

ASCII_code = 0 . . 255; { eight-bit numbers, a subrange of the integers }

See also sections 12, 36, 38, 47, 52, and 201.

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*;

13. The `WEAVE` and `TANGLE` processors convert between ASCII code and the user's external character set by means of arrays *xord* and *xchr* that are analogous to Pascal's *ord* and *chr* functions.

⟨ Globals in the outer block 9 ⟩ +≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [*ASCII_code*] **of** *text_char*; { specifies conversion of output characters }

14. If we assume that every system using WEB is able to read and write the visible characters of standard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize most of the *xchr* array properly, without needing any system-dependent changes. For example, the statement `xchr[@'101]:='A'` that appears in the present WEB file might be encoded in, say, EBCDIC code on the external medium on which it resides, but TANGLE will convert from this external code to ASCII and back again. Therefore the assignment statement `XCHR[65]:='A'` will appear in the corresponding Pascal file, and Pascal will compile this statement so that *xchr*[65] receives the character `A` in the external (*char*) code. Note that it would be quite incorrect to say `xchr[@'101]:="A"`, because `"A"` is a constant of type *integer*, not *char*, and because we have `"A"` = 65 regardless of the external character set.

⟨ Set initial values 10 ⟩ +≡

```
xchr['40] ← ' '; xchr['41] ← '!'; xchr['42] ← '"'; xchr['43] ← '#'; xchr['44] ← '$';
xchr['45] ← '%'; xchr['46] ← '&'; xchr['47] ← '''';
xchr['50] ← '('; xchr['51] ← ')'; xchr['52] ← '*'; xchr['53] ← '+'; xchr['54] ← ',';
xchr['55] ← '-'; xchr['56] ← '.'; xchr['57] ← '/';
xchr['60] ← '0'; xchr['61] ← '1'; xchr['62] ← '2'; xchr['63] ← '3'; xchr['64] ← '4';
xchr['65] ← '5'; xchr['66] ← '6'; xchr['67] ← '7';
xchr['70] ← '8'; xchr['71] ← '9'; xchr['72] ← ':'; xchr['73] ← ';'; xchr['74] ← '<';
xchr['75] ← '='; xchr['76] ← '>'; xchr['77] ← '?';
xchr['100] ← '@'; xchr['101] ← 'A'; xchr['102] ← 'B'; xchr['103] ← 'C'; xchr['104] ← 'D';
xchr['105] ← 'E'; xchr['106] ← 'F'; xchr['107] ← 'G';
xchr['110] ← 'H'; xchr['111] ← 'I'; xchr['112] ← 'J'; xchr['113] ← 'K'; xchr['114] ← 'L';
xchr['115] ← 'M'; xchr['116] ← 'N'; xchr['117] ← 'O';
xchr['120] ← 'P'; xchr['121] ← 'Q'; xchr['122] ← 'R'; xchr['123] ← 'S'; xchr['124] ← 'T';
xchr['125] ← 'U'; xchr['126] ← 'V'; xchr['127] ← 'W';
xchr['130] ← 'X'; xchr['131] ← 'Y'; xchr['132] ← 'Z'; xchr['133] ← '['; xchr['134] ← '\';
xchr['135] ← ']'; xchr['136] ← '^'; xchr['137] ← '_';
xchr['140] ← '`'; xchr['141] ← 'a'; xchr['142] ← 'b'; xchr['143] ← 'c'; xchr['144] ← 'd';
xchr['145] ← 'e'; xchr['146] ← 'f'; xchr['147] ← 'g';
xchr['150] ← 'h'; xchr['151] ← 'i'; xchr['152] ← 'j'; xchr['153] ← 'k'; xchr['154] ← 'l';
xchr['155] ← 'm'; xchr['156] ← 'n'; xchr['157] ← 'o';
xchr['160] ← 'p'; xchr['161] ← 'q'; xchr['162] ← 'r'; xchr['163] ← 's'; xchr['164] ← 't';
xchr['165] ← 'u'; xchr['166] ← 'v'; xchr['167] ← 'w';
xchr['170] ← 'x'; xchr['171] ← 'y'; xchr['172] ← 'z'; xchr['173] ← '{'; xchr['174] ← '|';
xchr['175] ← '}'; xchr['176] ← '~';
xchr[0] ← ' '; xchr['177] ← ' ';   { these ASCII codes are not used }
```

15. Some of the ASCII codes below *'40* have been given symbolic names in WEAVE and TANGLE because they are used with a special meaning.

```
define and_sign = '4   { equivalent to 'and' }
define not_sign = '5   { equivalent to 'not' }
define set_element_sign = '6   { equivalent to 'in' }
define tab_mark = '11   { ASCII code used as tab-skip }
define line_feed = '12   { ASCII code thrown away at end of line }
define form_feed = '14   { ASCII code used at end of page }
define carriage_return = '15   { ASCII code used at end of line }
define left_arrow = '30   { equivalent to ':=' }
define not_equal = '32   { equivalent to '<>' }
define less_or_equal = '34   { equivalent to '<=' }
define greater_or_equal = '35   { equivalent to '>=' }
define equivalence_sign = '36   { equivalent to '==' }
define or_sign = '37   { equivalent to 'or' }
```

16. When we initialize the *xord* array and the remaining parts of *xchr*, it will be convenient to make use of an index variable, *i*.

⟨ Local variables for initialization 16 ⟩ ≡
i: 0 . . 255;
See also sections 40, 56, and 247.
This code is used in section 2.

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

for $i \leftarrow 1$ **to** *´37* **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** $xchr[i] \leftarrow chr(i)$;

18. The following system-independent code makes the *xord* array contain a suitable inverse to the information in *xchr*.

⟨ Set initial values 10 ⟩ +≡
for $i \leftarrow$ *first_text_char* **to** *last_text_char* **do** $xord[chr(i)] \leftarrow$ "␣";
for $i \leftarrow 1$ **to** *´377* **do** $xord[xchr[i]] \leftarrow i$;
$xord[$*´␣´*$] \leftarrow$ "␣";

19. Input and output. The input conventions of this program are intended to be very much like those of TEX (except, of course, that they are much simpler, because much less needs to be done). Furthermore they are identical to those of `TANGLE`. Therefore people who need to make modifications to all three systems should be able to do so without too many headaches.

We use the standard Pascal input/output procedures in several places that TEX cannot, since `WEAVE` does not have to deal with files that are named dynamically by the user, and since there is no input from the terminal.

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 }
```

23. The main input comes from *web_file*; this input may be overridden by changes in *change_file*. (If *change_file* is empty, there are no changes.)

```
⟨ Globals in the outer block 9 ⟩ +≡
web_file: text_file;   { primary input }
change_file: text_file;   { updates }
```

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;
```

25. The main output goes to *tex_file*.

```
⟨ Globals in the outer block 9 ⟩ +≡
tex_file: text_file;
```

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);
```

27. Input goes into an array called *buffer*.

```
⟨ Globals in the outer block 9 ⟩ +≡
buffer: array [0 .. long_buf_size] of ASCII_code;
```

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;
```

29. Reporting errors to the user. The `WEAVE` processor 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.

The global variables *phase_one* and *phase_three* tell which Phase we are in.

```
⟨ Globals in the outer block 9 ⟩ +≡
phase_one: boolean;   { true in Phase I, false in Phases II and III }
phase_three: boolean;   { true in Phase III, false in Phases I and II }
```

30. If an error is detected while we are debugging, we usually want to look at the contents of memory. A special procedure will be declared later for this purpose.

```
⟨ Error handling procedures 30 ⟩ ≡
  debug procedure debug_help; forward; gubed
```

See also sections 31 and 33.

This code is used in section 2.

31. 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.

The actual error indications are provided by a procedure called *error*. However, error messages are not actually reported during phase one, since errors detected on the first pass will be detected again during the second.

```
  define err_print(#) ≡
          begin if ¬phase_one then
            begin new_line; print(#); error;
            end;
          end
⟨ Error handling procedures 30 ⟩ +≡
procedure error;   { prints '.' and location of error message }
  var k,l: 0 .. long_buf_size;   { indices into buffer }
  begin ⟨ Print error location based on input buffer 32 ⟩;
  update_terminal; mark_error;
  debug debug_skipped ← debug_cycle; debug_help; gubed
  end;
```

32. The error locations can be indicated by using the global variables *loc*, *line*, and *changing*, which tell respectively the first unlooked-at position in *buffer*, the current line number, and whether or not the current line is from *change_file* or *web_file*. This routine should be modified on systems whose standard text editor has special line-numbering conventions.

```
⟨ Print error location based on input buffer 32 ⟩ ≡
  begin if changing then print('.␣(change␣file␣') else print('.␣(');
  print_ln('l.', line : 1, ')');
  if loc ≥ limit then l ← limit
  else l ← loc;
  for k ← 1 to l do
    if buffer[k − 1] = tab_mark then print('␣')
    else print(xchr[buffer[k − 1]]);  { print the characters already read }
  new_line;
  for k ← 1 to l do print('␣');  { space out the next line }
  for k ← l + 1 to limit do print(xchr[buffer[k − 1]]);  { print the part not yet read }
  if buffer[limit] = "|" then print(xchr["|"]);  { end of Pascal text in module names }
  print('␣');  { this space separates the message from future asterisks }
  end
```

This code is used in section 31.

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 ≠ spotless) ∧ (history ≠ harmless_message) then uexit(1)
  else uexit(0);
  end;
```

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

```
  define confusion(#) ≡ fatal_error('!␣This␣can't␣happen␣(', #, ')')
```

35. An overflow stop occurs if WEAVE's tables aren't large enough.

```
  define overflow(#) ≡ fatal_error('!␣Sorry,␣', #, '␣capacity␣exceeded')
```

36. Data structures. During the first phase of its processing, `WEAVE` puts identifier names, index entries, and module names into the large *byte_mem* array, which is packed with eight-bit integers. Allocation is sequential, since names are never deleted.

An auxiliary array *byte_start* is used as a directory for *byte_mem*, and the *link*, *ilk*, and *xref* arrays give further information about names. These auxiliary arrays consist of sixteen-bit items.

⟨ Types in the outer block 11 ⟩ +≡
eight_bits = 0 .. 255; { unsigned one-byte quantity }
sixteen_bits = 0 .. 65535; { unsigned two-byte quantity }

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 .. *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 }
def_val: **array** [0 .. *max_names*] **of** *integer*; { values of numeric macros }

38. The names of identifiers are found by computing a hash address *h* and then looking at strings of bytes signified by *hash*[*h*], *link*[*hash*[*h*]], *link*[*link*[*hash*[*h*]]], ..., until either finding the desired name or encountering a zero.

A '*name_pointer*' variable, which signifies a name, is an index into *byte_start*. The actual sequence of characters in the name pointed to by *p* appears in positions *byte_start*[*p*] to *byte_start*[*p* + *ww*] − 1, inclusive, in the segment of *byte_mem* whose first index is *p* **mod** *ww*. Thus, when *ww* = 2 the even-numbered name bytes appear in *byte_mem*[0, ∗] and the odd-numbered ones appear in *byte_mem*[1, ∗]. The pointer 0 is used for undefined module names; we don't want to use it for the names of identifiers, since 0 stands for a null pointer in a linked list.

We usually have *byte_start*[*name_ptr* + *w*] = *byte_ptr*[(*name_ptr* + *w*) **mod** *ww*] for 0 ≤ *w* < *ww*, since these are the starting positions for the next *ww* names to be stored in *byte_mem*.

define *length*(#) ≡ *byte_start*[# + *ww*] − *byte_start*[#] { the length of a name }

⟨ Types in the outer block 11 ⟩ +≡
name_pointer = 0 .. *max_names*; { identifies a name }

39. ⟨ Globals in the outer block 9 ⟩ +≡
name_ptr: *name_pointer*; { first unused position in *byte_start* }
byte_ptr: **array** [0 .. *ww* − 1] **of** 0 .. *max_bytes*; { first unused position in *byte_mem* }

40. ⟨ Local variables for initialization 16 ⟩ +≡
wi: 0 .. *ww* − 1; { to initialize the *byte_mem* indices }

41. ⟨ Set initial values 10 ⟩ +≡
```
for wi ← 0 to ww − 1 do
  begin byte_start[wi] ← 0; byte_ptr[wi] ← 0;
  end;
byte_start[ww] ← 0;  { this makes name 0 of length zero }
name_ptr ← 1;
```

42. Several types of identifiers are distinguished by their *ilk*:

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

roman identifiers are index entries that appear after `@^` in the `WEB` file.

wildcard identifiers are index entries that appear after `@:` in the `WEB` file.

typewriter identifiers are index entries that appear after `@.` in the `WEB` file.

array_like, *begin_like*, ..., *var_like* identifiers are Pascal reserved words whose *ilk* explains how they are to be treated when Pascal code is being formatted.

Finally, if c is an ASCII code, an *ilk* equal to *char_like* + c denotes a reserved word that will be converted to character c.

```
define normal = 0  { ordinary identifiers have normal ilk }
define roman = 1  { normal index entries have roman ilk }
define wildcard = 2  { user-formatted index entries have wildcard ilk }
define typewriter = 3  { 'typewriter type' entries have typewriter ilk }
define reserved(#) ≡ (ilk[#] > typewriter)  { tells if a name is a reserved word }
define array_like = 4  { array, file, set }
define begin_like = 5  { begin }
define case_like = 6  { case }
define const_like = 7  { const, label, type }
define div_like = 8  { div, mod }
define do_like = 9  { do, of, then }
define else_like = 10  { else }
define end_like = 11  { end }
define for_like = 12  { for, while, with }
define goto_like = 13  { goto, packed }
define if_like = 14  { if }
define intercal_like = 15  { not used }
define nil_like = 16  { nil }
define proc_like = 17  { function, procedure, program }
define record_like = 18  { record }
define repeat_like = 19  { repeat }
define to_like = 20  { downto, to }
define until_like = 21  { until }
define var_like = 22  { var }
define loop_like = 23  { loop, xclause }
define char_like = 24  { and, or, not, in }
```

43. The names of modules are stored in *byte_mem* together with the identifier names, but a hash table is not used for them because `WEAVE` needs to be able to recognize a module name when given a prefix of that name. A conventional binary search tree is used to retrieve module names, with fields called *llink* and *rlink* in place of *link* and *ilk*. The root of this tree is *rlink*[0].

define *llink* ≡ *link* { left link in binary search tree for module names }
define *rlink* ≡ *ilk* { right link in binary search tree for module names }
define *root* ≡ *rlink*[0] { the root of the binary search tree for module names }

⟨ Set initial values 10 ⟩ +≡
root ← 0; { the binary search tree starts out with nothing in it }

44. Here is a little procedure that prints the text of a given name on the user's terminal.

```
procedure print_id(p : name_pointer); { print identifier or module name }
  var k: 0 .. max_bytes; { index into byte_mem }
    w: 0 .. ww − 1; { row of byte_mem }
  begin if p ≥ name_ptr then print('IMPOSSIBLE')
  else begin w ← p mod ww;
    for k ← byte_start[p] to byte_start[p + ww] − 1 do print(xchr[byte_mem[w, k]]);
    end;
  end;
```

45. We keep track of the current module number in *module_count*, which is the total number of modules that have started. Modules which have been altered by a change file entry have their *changed_module* flag turned on during the first phase.

⟨ Globals in the outer block 9 ⟩ +≡
module_count: 0 .. *max_modules*; { the current module number }
changed_module: **packed array** [0 .. *max_modules*] **of** *boolean*; { is it changed? }
change_exists: *boolean*; { has any module changed? }

46. The other large memory area in `WEAVE` keeps the cross-reference data. All uses of the name p are recorded in a linked list beginning at *xref*[p], which points into the *xmem* array. Entries in *xmem* consist of two sixteen-bit items per word, called the *num* and *xlink* fields. If x is an index into *xmem*, reached from name p, the value of *num*(x) is either a module number where p is used, or it is *def_flag* plus a module number where p is defined; and *xlink*(x) points to the next such cross reference for p, if any. This list of cross references is in decreasing order by module number. The current number of cross references is *xref_ptr*.

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

define *num*(#) ≡ *xmem*[#].*num_field*
define *xlink*(#) ≡ *xmem*[#].*xlink_field*
define *dtype*(#) ≡ *xmem*[#].*dtype_field*
define *dname*(#) ≡ *xmem*[#].*dname_field*
define *dback*(#) ≡ *xmem*[#].*dback_field*
define *dlink*(#) ≡ *xmem*[#].*dlink_field*
define *def_flag* = 10240 { must be strictly larger than *max_modules* }

47. ⟨ Types in the outer block 11 ⟩ +≡
xref_number = 0 .. *max_refs*;

48. Definitions are classified into 15 types.

```
define dtype_none = 0
define dtype_macro = 1
define dtype_const = 2
define dtype_string = 3
define dtype_colon_bold = 4
define dtype_equal_bold = 5
define dtype_colon_packed = 6
define dtype_equal_packed = 7
define dtype_colon_ital = 8
define dtype_equal_ital = 9
define dtype_comma = 10
define dtype_colon_const_dots = 11
define dtype_equal_const_dots = 12
define dtype_colon_ital_dots = 13
define dtype_equal_ital_dots = 14
```

```
⟨ Globals in the outer block 9 ⟩ +≡
xmem: array [xref_number] of packed record
    num_field: sixteen_bits;  { module number plus zero or def_flag }
    xlink_field: sixteen_bits;  { pointer to the previous cross reference }
    dtype_field: sixteen_bits;  { type of definition }
    dname_field: sixteen_bits;  { identifier or constant }
    dback_field: sixteen_bits;  { if nonzero, this is a reference to name }
    dlink_field: sixteen_bits;  { link to definitions only }
    end;
xref_ptr: xref_number;  { the largest occupied position in xmem }
xref_switch, mod_xref_switch: 0 .. def_flag;  { either zero or def_flag }
def_type, def_name, def_subtype, def_subname, const_name, packed_name: integer;
danger_zone: boolean;
```

49.

```
⟨ Set initial values 10 ⟩ +≡
  xref_ptr ← 0; xref_switch ← 0; mod_xref_switch ← 0; num(0) ← 0; danger_zone ← false; xref[0] ← 0;
      { cross references to undefined modules }
```

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.

The *new_blank_xref* is similar but it ignores the *xref_switch*.

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) ← #; dback(xref_ptr) ← 0;
          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
        if (def_type ≥ dtype_comma) ∨ danger_zone then q ← xlink(q) { delete entry }
        else begin num(q) ← m; dtype(q) ← def_type; dname(q) ← def_name; dback(q) ← p; return;
          end;
    end;
  append_xref(m); xlink(xref_ptr) ← q; xref[p] ← xref_ptr;
  if m > def_flag then
    begin dtype(xref_ptr) ← def_type; dname(xref_ptr) ← def_name; dback(xref_ptr) ← p;
    if def_type > dtype_comma then
      begin append_xref(0); dtype(xref_ptr) ← def_subtype; dname(xref_ptr) ← def_subname;
      end;
    end;
exit: end;
procedure new_blank_xref(p : integer);
  var xs: integer;
  begin xs ← xref_switch; xref_switch ← 0; new_xref(p); xref_switch ← xs;
  end;
```

51. The cross reference lists for module names are slightly different. Suppose that a module name is defined in modules $m_1, \ldots, m_k$ and used in modules $n_1, \ldots, n_l$. Then its list will contain $m_1 + def\_flag$, $m_k + def\_flag, \ldots, m_2 + def\_flag, n_l, \ldots, n_1$, in this order. After Phase II, however, the order will be $m_1 + def\_flag, \ldots, m_k + def\_flag, n_1, \ldots, n_l$.

```
procedure new_mod_xref(p : name_pointer);
  var q, r: xref_number;  { pointers to previous cross references }
  begin q ← xref[p]; r ← 0;
  if q > 0 then
    begin if mod_xref_switch = 0 then
      while num(q) ≥ def_flag do
        begin r ← q; q ← xlink(q);
        end
    else if num(q) ≥ def_flag then
        begin r ← q; q ← xlink(q);
        end;
    end;
  append_xref(module_count + mod_xref_switch); xlink(xref_ptr) ← q; mod_xref_switch ← 0;
  if r = 0 then xref[p] ← xref_ptr
  else xlink(r) ← xref_ptr;
  end;
```

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

```
⟨ Types in the outer block 11 ⟩ +≡
  text_pointer = 0 .. max_texts;  { identifies a token list }
```

53. The first position of *tok_mem* that is unoccupied by replacement text is called *tok_ptr*, and the first unused location of *tok_start* is called *text_ptr*. Thus, we usually have *tok_start*[*text_ptr*] = *tok_ptr*.

```
⟨ Globals in the outer block 9 ⟩ +≡
  tok_mem: packed array [0 .. max_toks] of sixteen_bits;  { tokens }
  tok_start: array [text_pointer] of sixteen_bits;  { directory into tok_mem }
  text_ptr: text_pointer;  { first unused position in tok_start }
  tok_ptr: 0 .. max_toks;  { first unused position in tok_mem }
  stat max_tok_ptr, max_txt_ptr: 0 .. max_toks;  { largest values occurring }
  tats
```

54.
```
⟨ Set initial values 10 ⟩ +≡
  tok_ptr ← 1; text_ptr ← 1; tok_start[0] ← 1; tok_start[1] ← 1;
  stat max_tok_ptr ← 1; max_txt_ptr ← 1; tats
```

55. Searching for identifiers. The hash table described above is updated by the *id_lookup* procedure, which finds a given identifier and returns a pointer to its index in *byte_start*. The identifier is supposed to match character by character and it is also supposed to have a given *ilk* code; the same name may be present more than once if it is supposed to appear in the index with different typesetting conventions. If the identifier was not already present, it is inserted into the table.

Because of the way WEAVE's scanning mechanism works, it is most convenient to let *id_lookup* search for an identifier that is present in the *buffer* array. Two other global variables specify its position in the buffer: the first character is *buffer*[*id_first*], and the last is *buffer*[*id_loc* − 1].

```
⟨ Globals in the outer block 9 ⟩ +≡
id_first: 0 .. long_buf_size;  { where the current identifier begins in the buffer }
id_loc: 0 .. long_buf_size;  { just after the current identifier in the buffer }
hash: array [0 .. hash_size] of sixteen_bits;  { heads of hash lists }
```

56. Initially all the hash lists are empty.

```
⟨ Local variables for initialization 16 ⟩ +≡
h: 0 .. hash_size;  { index into hash-head array }
```

57.
```
⟨ Set initial values 10 ⟩ +≡
  for h ← 0 to hash_size − 1 do hash[h] ← 0;
```

58. Here now is the main procedure for finding identifiers (and index entries). The parameter t is set to the desired *ilk* code. The identifier must either have *ilk* = t, or we must have t = *normal* and the identifier must be a reserved word.

```
function id_lookup(t : eight_bits): name_pointer;  { finds current identifier }
  label found;
  var i: 0 .. long_buf_size;  { index into buffer }
    h: 0 .. hash_size;  { hash code }
    k: 0 .. max_bytes;  { index into byte_mem }
    w: 0 .. ww − 1;  { row of byte_mem }
    l: 0 .. long_buf_size;  { length of the given identifier }
    p: name_pointer;  { where the identifier is being sought }
  begin l ← id_loc − id_first;  { compute the length }
  ⟨ Compute the hash code h 59 ⟩;
  ⟨ Compute the name location p 60 ⟩;
  if p = name_ptr then ⟨ Enter a new name into the table at position p 62 ⟩;
  id_lookup ← p;
  end;
```

59. A simple hash code is used: If the sequence of ASCII 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 59 ⟩ ≡
  h ← buffer[id_first]; i ← id_first + 1;
  while i < id_loc do
    begin h ← (h + h + buffer[i]) mod hash_size; incr(i);
    end
```

This code is used in section 58.

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

```
⟨ Compute the name location p 60 ⟩ ≡
  p ← hash[h];
  while p ≠ 0 do
    begin if (length(p) = l) ∧ ((ilk[p] = t) ∨ ((t = normal) ∧ reserved(p))) then
      ⟨ Compare name p with current identifier, goto found if equal 61 ⟩;
    p ← link[p];
    end;
  p ← name_ptr;  { the current identifier is new }
  link[p] ← hash[h]; hash[h] ← p;  { insert p at beginning of hash list }
found:
```

This code is used in section 58.

61.

```
⟨ Compare name p with current identifier, goto found if equal 61 ⟩ ≡
  begin i ← id_first; k ← byte_start[p]; w ← p mod ww;
  while (i < id_loc) ∧ (buffer[i] = byte_mem[w, k]) do
    begin incr(i); incr(k);
    end;
  if i = id_loc then goto found;  { all characters agree }
  end
```

This code is used in section 60.

62. When we begin the following segment of the program, $p = name\_ptr$.

```
  define undef_val ≡ 10000000
⟨ Enter a new name into the table at position p 62 ⟩ ≡
  begin w ← name_ptr mod ww;
  if byte_ptr[w] + l > max_bytes then overflow('byte␣memory');
  if name_ptr + ww > max_names then overflow('name');
  i ← id_first; k ← byte_ptr[w];  { get ready to move the identifier into byte_mem }
  while i < id_loc do
    begin byte_mem[w, k] ← buffer[i]; incr(k); incr(i);
    end;
  byte_ptr[w] ← k; byte_start[name_ptr + ww] ← k; incr(name_ptr); ilk[p] ← t; xref[p] ← 0;
  def_val[p] ← undef_val;
  end
```

This code is used in section 58.

63. Initializing the table of reserved words. We have to get Pascal's reserved words into the hash table, and the simplest way to do this is to insert them every time `WEAVE` is run. A few macros permit us to do the initialization with a compact program.

```
define sid9(#) ≡ buffer[9] ← #; cur_name ← id_lookup
define sid8(#) ≡ buffer[8] ← #; sid9
define sid7(#) ≡ buffer[7] ← #; sid8
define sid6(#) ≡ buffer[6] ← #; sid7
define sid5(#) ≡ buffer[5] ← #; sid6
define sid4(#) ≡ buffer[4] ← #; sid5
define sid3(#) ≡ buffer[3] ← #; sid4
define sid2(#) ≡ buffer[2] ← #; sid3
define sid1(#) ≡ buffer[1] ← #; sid2
define id2 ≡ id_first ← 8; sid8
define id3 ≡ id_first ← 7; sid7
define id4 ≡ id_first ← 6; sid6
define id5 ≡ id_first ← 5; sid5
define id6 ≡ id_first ← 4; sid4
define id7 ≡ id_first ← 3; sid3
define id8 ≡ id_first ← 2; sid2
define id9 ≡ id_first ← 1; sid1
```

⟨ Globals in the outer block 9 ⟩ +≡

```
cur_name: name_pointer; { points to the identifier just inserted }
```

64. The intended use of the macros above might not be immediately obvious, but the riddle is answered by the following:

```
⟨ Store all the reserved words 64 ⟩ ≡
  id_loc ← 10;
  id3("a")("n")("d")(char_like + and_sign);
  id5("a")("r")("r")("a")("y")(array_like);
  id5("b")("e")("g")("i")("n")(begin_like);
  id4("c")("a")("s")("e")(case_like);
  id5("c")("o")("n")("s")("t")(const_like); const_name ← cur_name;
  id3("d")("i")("v")(div_like);
  id2("d")("o")(do_like);
  id6("d")("o")("w")("n")("t")("o")(to_like);
  id4("e")("l")("s")("e")(else_like);
  id3("e")("n")("d")(end_like);
  id4("f")("i")("l")("e")(array_like);
  id3("f")("o")("r")(for_like);
  id8("f")("u")("n")("c")("t")("i")("o")("n")(proc_like);
  id4("g")("o")("t")("o")(goto_like);
  id2("i")("f")(if_like);
  id2("i")("n")(char_like + set_element_sign);
  id5("l")("a")("b")("e")("l")(const_like);
  id3("m")("o")("d")(div_like);
  id3("n")("i")("l")(nil_like);
  id3("n")("o")("t")(char_like + not_sign);
  id2("o")("f")(do_like);
  id2("o")("r")(char_like + or_sign);
  id6("p")("a")("c")("k")("e")("d")(goto_like); packed_name ← cur_name;
  id9("p")("r")("o")("c")("e")("d")("u")("r")("e")(proc_like);
  id7("p")("r")("o")("g")("r")("a")("m")(proc_like);
  id6("r")("e")("c")("o")("r")("d")(record_like);
  id6("r")("e")("p")("e")("a")("t")(repeat_like);
  id3("s")("e")("t")(array_like);
  id4("t")("h")("e")("n")(do_like);
  id2("t")("o")(to_like);
  id4("t")("y")("p")("e")(const_like);
  id5("u")("n")("t")("i")("l")(until_like);
  id3("v")("a")("r")(var_like);
  id5("w")("h")("i")("l")("e")(for_like);
  id4("w")("i")("t")("h")(for_like);
  id7("x")("c")("l")("a")("u")("s")("e")(loop_like);
```

This code is used in section 261.

65. Searching for module names. The *mod_lookup* procedure finds the module name *mod_text*[1 .. *l*] in the search tree, after inserting it if necessary, and returns a pointer to where it was found.

```
⟨ Globals in the outer block 9 ⟩ +≡
mod_text: array [0 .. longest_name] of ASCII_code;  { name being sought for }
```

66. According to the rules of WEB, no module name should be a proper prefix of another, so a "clean" comparison should occur between any two names. The result of *mod_lookup* is 0 if this prefix condition is violated. An error message is printed when such violations are detected during phase two of WEAVE.

```
  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 }
function mod_lookup(l : sixteen_bits): name_pointer;  { finds module name }
  label found;
  var c: less .. extension;  { comparison between two names }
    j: 0 .. longest_name;  { index into mod_text }
    k: 0 .. max_bytes;  { index into byte_mem }
    w: 0 .. ww − 1;  { row of byte_mem }
    p: name_pointer;  { current node of the search tree }
    q: name_pointer;  { father of node p }
  begin c ← greater; q ← 0; p ← root;
  while p ≠ 0 do
    begin ⟨ Set variable c to the result of comparing the given name to name p 68 ⟩;
    q ← p;
    if c = less then p ← llink[q]
    else if c = greater then p ← rlink[q]
      else goto found;
    end;
  ⟨ Enter a new module name into the tree 67 ⟩;
found: if c ≠ equal then
    begin err_print('! Incompatible section names'); p ← 0;
    end;
  mod_lookup ← p;
  end;
```

67. ⟨ Enter a new module name into the tree 67 ⟩ ≡

```
  w ← name_ptr mod ww; k ← byte_ptr[w];
  if k + l > max_bytes then overflow('byte memory');
  if name_ptr > max_names − ww then overflow('name');
  p ← name_ptr;
  if c = less then llink[q] ← p
  else rlink[q] ← p;
  llink[p] ← 0; rlink[p] ← 0; xref[p] ← 0; c ← equal;
  for j ← 1 to l do byte_mem[w, k + j − 1] ← mod_text[j];
  byte_ptr[w] ← k + l; byte_start[name_ptr + ww] ← k + l; incr(name_ptr);
```

This code is used in section 66.

68. ⟨ Set variable c to the result of comparing the given name to name p 68 ⟩ ≡

```
begin k ← byte_start[p]; w ← p mod ww; c ← equal; j ← 1;
while (k < byte_start[p + ww]) ∧ (j ≤ l) ∧ (mod_text[j] = byte_mem[w, k]) do
  begin incr(k); incr(j);
  end;
if k = byte_start[p + ww] then
  if j > l then c ← equal
  else c ← extension
else if j > l then c ← prefix
  else if mod_text[j] < byte_mem[w, k] then c ← less
    else c ← greater;
end
```

This code is used in sections 66 and 69.

69. The *prefix_lookup* procedure is supposed to find exactly one module name that has *mod_text*[1 . . l] as a prefix. Actually the algorithm silently accepts also the situation that some module name is a prefix of *mod_text*[1 . . l], because the user who painstakingly typed in more than necessary probably doesn't want to be told about the wasted effort.

Recall that error messages are not printed during phase one. It is possible that the *prefix_lookup* procedure will fail on the first pass, because there is no match, yet the second pass might detect no error if a matching module name has occurred after the offending prefix. In such a case the cross-reference information will be incorrect and `WEAVE` will report no error. However, such a mistake will be detected by the `TANGLE` processor.

```
function prefix_lookup(l : sixteen_bits): name_pointer;  { finds name extension }
  var c: less . . extension;  { comparison between two names }
    count: 0 . . max_names;  { the number of hits }
    j: 0 . . longest_name;  { index into mod_text }
    k: 0 . . max_bytes;  { index into byte_mem }
    w: 0 . . ww − 1;  { row of byte_mem }
    p: name_pointer;  { current node of the search tree }
    q: name_pointer;  { another place to resume the search after one branch is done }
    r: name_pointer;  { extension found }
  begin q ← 0; p ← root; count ← 0; r ← 0;  { begin search at root of tree }
  while p ≠ 0 do
    begin ⟨ Set variable c to the result of comparing the given name to name p 68 ⟩;
    if c = less then p ← llink[p]
    else if c = greater then p ← rlink[p]
      else begin r ← p; incr(count); q ← rlink[p]; p ← llink[p];
        end;
    if p = 0 then
      begin p ← q; q ← 0;
      end;
    end;
  if count ≠ 1 then
    if count = 0 then err_print('! Name does not match')
    else err_print('! Ambiguous prefix');
  prefix_lookup ← r;  { the result will be 0 if there was no match }
  end;
```

70. Lexical scanning. Let us now consider the subroutines that read the WEB source file and break it into meaningful units. There are four such procedures: One simply skips to the next '@␣' or '@*' that begins a module; another passes over the TEX text at the beginning of a module; the third passes over the TEX text in a Pascal comment; and the last, which is the most interesting, gets the next token of a Pascal text.

71. But first we need to consider the low-level routine *get_line* that takes care of merging *change_file* into *web_file*. The *get_line* procedure also updates the line numbers for error messages.

```
⟨ Globals in the outer block 9 ⟩ +≡
ii: integer; { general purpose for loop variable in the outer block }
line: integer; { the number of the current line in the current file }
other_line: integer; { the number of the current line in the input file that is not currently being read }
temp_line: integer; { used when interchanging line with other_line }
limit: 0 .. long_buf_size; { the last character position occupied in the buffer }
loc: 0 .. long_buf_size; { the next character position to be read from the buffer }
input_has_ended: boolean; { if true, there is no more input }
changing: boolean; { if true, the current line is from change_file }
```

72. As we change *changing* from *true* to *false* and back again, we must remember to swap the values of *line* and *other_line* so that the *err_print* routine will be sure to report the correct line number.

```
define change_changing ≡ changing ← ¬changing; temp_line ← other_line; other_line ← line;
    line ← temp_line { line ↔ other_line }
```

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

```
⟨ Globals in the outer block 9 ⟩ +≡
change_buffer: array [0 .. buf_size] of ASCII_code;
change_limit: 0 .. buf_size; { the last position occupied in change_buffer }
```

74. Here's a simple function that checks if the two buffers are different.

```
function lines_dont_match: boolean;
  label exit;
  var k: 0 .. buf_size; { index into the buffers }
  begin lines_dont_match ← true;
  if change_limit ≠ limit then return;
  if limit > 0 then
    for k ← 0 to limit − 1 do
      if change_buffer[k] ≠ buffer[k] then return;
  lines_dont_match ← false;
exit: end;
```

75. 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 $(change\_limit = 0) \wedge \neg changing$ 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.

```
procedure prime_the_change_buffer;
  label continue, done, exit;
  var k: 0 .. buf_size;  { index into the buffers }
  begin change_limit ← 0;  { this value will be used if the change file ends }
  ⟨ Skip over comment lines in the change file; return if end of file 76 ⟩;
  ⟨ Skip to the next nonblank line; return if end of file 77 ⟩;
  ⟨ Move buffer and limit to change_buffer and change_limit 78 ⟩;
exit: end;
```

76. 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` or `@z` (which would probably indicate that the change file is fouled up).

```
⟨ Skip over comment lines in the change file; return if end of file 76 ⟩ ≡
  loop begin incr(line);
    if ¬input_ln(change_file) then return;
    if limit < 2 then goto continue;
    if buffer[0] ≠ "@" then goto continue;
    if (buffer[1] ≥ "X") ∧ (buffer[1] ≤ "Z") then buffer[1] ← buffer[1] + "z" − "Z";  { lowercasify }
    if buffer[1] = "x" then goto done;
    if (buffer[1] = "y") ∨ (buffer[1] = "z") then
      begin loc ← 2; err_print('!␣Where␣is␣the␣matching␣@x?');
      end;
  continue: end;
done:
```

This code is used in section 75.

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

```
⟨ Skip to the next nonblank line; return if end of file 77 ⟩ ≡
  repeat incr(line);
    if ¬input_ln(change_file) then
      begin err_print('!␣Change␣file␣ended␣after␣@x'); return;
      end;
  until limit > 0;
```

This code is used in section 75.

78. ⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 78 ⟩ ≡

```
  begin change_limit ← limit;
  if limit > 0 then
    for k ← 0 to limit − 1 do change_buffer[k] ← buffer[k];
  end
```

This code is used in sections 75 and 79.

79. 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*.

```
procedure check_change; { switches to change_file if the buffers match }
  label exit;
  var n: integer; { the number of discrepancies found }
    k: 0 .. buf_size; { index into the buffers }
  begin if lines_dont_match then return;
  n ← 0;
  loop begin change_changing; { now it's true }
    incr(line);
    if ¬input_ln(change_file) then
      begin err_print('!␣Change␣file␣ended␣before␣@y'); change_limit ← 0; change_changing;
          { false again }
      return;
      end;
    ⟨ If the current line starts with @y, report any discrepancies and return 80 ⟩;
    ⟨ Move buffer and limit to change_buffer and change_limit 78 ⟩;
    change_changing; { now it's false }
    incr(line);
    if ¬input_ln(web_file) then
      begin err_print('!␣WEB␣file␣ended␣during␣a␣change'); input_has_ended ← true; return;
      end;
    if lines_dont_match then incr(n);
    end;
exit: end;
```

80. ⟨ If the current line starts with `@y`, report any discrepancies and **return** 80 ⟩ ≡

```
if limit > 1 then
  if buffer[0] = "@" then
    begin if (buffer[1] ≥ "X") ∧ (buffer[1] ≤ "Z") then buffer[1] ← buffer[1] + "z" − "Z";
        { lowercasify }
    if (buffer[1] = "x") ∨ (buffer[1] = "z") then
      begin loc ← 2; err_print('!␣Where␣is␣the␣matching␣@y?');
      end
    else if buffer[1] = "y" then
        begin if n > 0 then
          begin loc ← 2;
          err_print('!␣Hmm...␣', n : 1, '␣of␣the␣preceding␣lines␣failed␣to␣match');
          end;
        return;
        end;
    end
```

This code is used in section 79.

81. The *reset_input* procedure, which gets `WEAVE` ready to read the user's `WEB` input, is used at the beginning of phases one and two.

```
procedure reset_input;
  begin open_input; line ← 0; other_line ← 0;
  changing ← true; prime_the_change_buffer; change_changing;
  limit ← 0; loc ← 1; buffer[0] ← "␣"; input_has_ended ← false;
  end;
```

82. 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.

```
procedure get_line; { inputs the next line }
  label restart;
  begin restart: if changing then changed_module[module_count] ← true
  else ⟨ Read from web_file and maybe turn on changing 83 ⟩;
  if changing then
    begin ⟨ Read from change_file and maybe turn off changing 84 ⟩;
    if ¬changing then
      begin changed_module[module_count] ← true; goto restart;
      end;
    end;
  loc ← 0; buffer[limit] ← "␣";
  end;
```

83. ⟨ Read from *web_file* and maybe turn on *changing* 83 ⟩ ≡

```
  begin incr(line);
  if ¬input_ln(web_file) then input_has_ended ← true
  else if limit = change_limit then
      if buffer[0] = change_buffer[0] then
        if change_limit > 0 then check_change;
  end
```

This code is used in section 82.

84. ⟨ Read from *change_file* and maybe turn off *changing* 84 ⟩ ≡

```
  begin incr(line);
  if ¬input_ln(change_file) then
    begin err_print('!␣Change␣file␣ended␣without␣@z'); buffer[0] ← "@"; buffer[1] ← "z"; limit ← 2;
    end;
  if limit > 1 then { check if the change has ended }
    if buffer[0] = "@" then
      begin if (buffer[1] ≥ "X") ∧ (buffer[1] ≤ "Z") then buffer[1] ← buffer[1] + "z" − "Z";
            { lowercasify }
      if (buffer[1] = "x") ∨ (buffer[1] = "y") then
        begin loc ← 2; err_print('!␣Where␣is␣the␣matching␣@z?');
        end
      else if buffer[1] = "z" then
          begin prime_the_change_buffer; change_changing;
          end;
      end;
  end
```

This code is used in section 82.

85. 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*.

```
⟨ Check that all changes have been read 85 ⟩ ≡
  if change_limit ≠ 0 then { changing is false }
    begin for ii ← 0 to change_limit − 1 do buffer[ii] ← change_buffer[ii];
    limit ← change_limit; changing ← true; line ← other_line; loc ← change_limit;
    err_print('! Change file entry did not match');
    end
```

This code is used in section 261.

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

```
define ignore = 0 { control code of no interest to WEAVE }
define verbatim = '2 { extended ASCII alpha will not appear }
define force_line = '3 { extended ASCII beta will not appear }
define begin_comment = '11 { ASCII tab mark will not appear }
define end_comment = '12 { ASCII line feed will not appear }
define octal = '14 { ASCII form feed will not appear }
define hex = '15 { ASCII carriage return will not appear }
define double_dot = '40 { ASCII space will not appear except in strings }
define no_underline = '175 { this code will be intercepted without confusion }
define underline = '176 { this code will be intercepted without confusion }
define param = '177 { ASCII delete will not appear }
define xref_roman = '203 { control code for '@^' }
define xref_wildcard = '204 { control code for '@:' }
define xref_typewriter = '205 { control code for '@.' }
define TeX_string = '206 { control code for '@t' }
define check_sum = '207 { control code for '@$' }
define join = '210 { control code for '@&' }
define thin_space = '211 { control code for '@,' }
define math_break = '212 { control code for '@|' }
define line_break = '213 { control code for '@/' }
define big_line_break = '214 { control code for '@#' }
define no_line_break = '215 { control code for '@+' }
define pseudo_semi = '216 { control code for '@;' }
define format = '217 { control code for '@f' }
define definition = '220 { control code for '@d' }
define begin_Pascal = '221 { control code for '@p' }
define module_name = '222 { control code for '@<' }
define new_module = '223 { control code for '@␣' and '@*' }
```

87. Control codes are converted from ASCII to WEAVE's internal representation by the *control_code* routine.

```
function control_code(c : ASCII_code): eight_bits;  { convert c after @ }
  begin case c of
  "@": control_code ← "@";  { 'quoted' at sign }
  "´": control_code ← octal;  { precedes octal constant }
  """": control_code ← hex;  { precedes hexadecimal constant }
  "$": control_code ← check_sum;  { precedes check sum constant }
  "␣", tab_mark, "*": control_code ← new_module;  { beginning of a new module }
  "=": control_code ← verbatim;
  "\": control_code ← force_line;
  "D", "d": control_code ← definition;  { macro definition }
  "F", "f": control_code ← format;  { format definition }
  "{": control_code ← begin_comment;  { begin-comment delimiter }
  "}": control_code ← end_comment;  { end-comment delimiter }
  "P", "p": control_code ← begin_Pascal;  { Pascal text in unnamed module }
  "&": control_code ← join;  { concatenate two tokens }
  "<": control_code ← module_name;  { beginning of a module name }
  ">": begin err_print('!␣Extra␣@>'); control_code ← ignore;
    end;  { end of module name should not be discovered in this way }
  "T", "t": control_code ← TeX_string;  { TEX box within Pascal }
  "!": control_code ← underline;  { set definition flag }
  "?": control_code ← no_underline;  { reset definition flag }
  "^": control_code ← xref_roman;  { index entry to be typeset normally }
  ":": control_code ← xref_wildcard;  { index entry to be in user format }
  ".": control_code ← xref_typewriter;  { index entry to be in typewriter type }
  ",": control_code ← thin_space;  { puts extra space in Pascal format }
  "|": control_code ← math_break;  { allows a break in a formula }
  "/": control_code ← line_break;  { forces end-of-line in Pascal format }
  "#": control_code ← big_line_break;  { forces end-of-line and some space besides }
  "+": control_code ← no_line_break;  { cancels end-of-line down to single space }
  ";": control_code ← pseudo_semi;  { acts like a semicolon, but is invisible }
  ⟨ Special control codes allowed only when debugging 88 ⟩
  othercases begin err_print('!␣Unknown␣control␣code'); control_code ← ignore;
    end
  endcases;
  end;
```

88. If WEAVE is compiled with debugging commands, one can write @2, @1, and @0 to turn tracing fully on, partly on, and off, respectively.

```
⟨ Special control codes allowed only when debugging 88 ⟩ ≡
  debug
"0", "1", "2": begin tracing ← c − "0"; control_code ← ignore;
"3": begin @{'**********************************************************'@}control_code ← ignore;
      { to set a breakpoint }
  end;
  end;
  gubed
```

This code is used in section 87.

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

```
procedure skip_limbo; { skip to next module }
  label exit;
  var c: ASCII_code; { character following @ }
  begin loop
    if loc > limit then
      begin get_line;
      if input_has_ended then return;
      end
    else begin buffer[limit + 1] ← "@";
      while buffer[loc] ≠ "@" do incr(loc);
      if loc ≤ limit then
        begin loc ← loc + 2; c ← buffer[loc − 1];
        if (c = "␣") ∨ (c = tab_mark) ∨ (c = "*") then return;
        end;
      end;
exit: end;
```

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

```
function skip_TeX: eight_bits; { skip past pure TEX code }
  label done;
  var c: eight_bits; { control code found }
  begin loop
    begin if loc > limit then
      begin get_line;
      if input_has_ended then
        begin c ← new_module; goto done;
        end;
      end;
    buffer[limit + 1] ← "@";
    repeat c ← buffer[loc]; incr(loc);
      if c = "|" then goto done;
    until c = "@";
    if loc ≤ limit then
      begin c ← control_code(buffer[loc]); incr(loc); goto done;
      end;
    end;
done: skip_TeX ← c;
  end;
```

91. The *skip_comment* routine is used on the first pass to skip through TEX code in Pascal comments. The *bal* parameter tells how many left braces are assumed to have been scanned when this routine is called, and the procedure returns a corresponding value of *bal* at the point that scanning has stopped. Scanning stops either at a '|' that introduces Pascal text, in which case the returned value is positive, or it stops at the end of the comment, in which case the returned value is zero. The scanning also stops in anomalous situations when the comment doesn't end or when it contains an illegal use of `@`. One should call *skip_comment*(1) when beginning to scan a comment.

```
function skip_comment(bal : eight_bits): eight_bits;  { skips TEX code in comments }
  label done;
  var c: ASCII_code;  { the current character }
  begin loop
    begin if loc > limit then
      begin get_line;
      if input_has_ended then
        begin bal ← 0; goto done;
        end;  { an error message will occur in phase two }
      end;
    c ← buffer[loc]; incr(loc);
    if c = "|" then goto done;
    ⟨ Do special things when c = "@", "\", "{", "}"; goto done at end 92 ⟩;
    end;
done: skip_comment ← bal;
  end;
```

92. ⟨ Do special things when $c =$ `"@"`, `"\"`, `"{"`, `"}"`; **goto** *done* at end 92 ⟩ ≡

```
  if c = "@" then
    begin c ← buffer[loc];
    if (c ≠ "␣") ∧ (c ≠ tab_mark) ∧ (c ≠ "*") then incr(loc)
    else begin decr(loc); bal ← 0; goto done;
      end  { an error message will occur in phase two }
    end
  else if (c = "\") ∧ (buffer[loc] ≠ "@") then incr(loc)
    else if c = "{" then incr(bal)
      else if c = "}" then
          begin decr(bal);
          if bal = 0 then goto done;
          end
```

This code is used in section 91.

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

The result of *get_next* is either an ASCII code for some special character, or it is a special code representing a pair of characters (e.g., '`:=`' or '`..`'), or it is the numeric value computed by the *control_code* procedure, or it is one of the following special codes:

exponent: The '`E`' in a real constant.

identifier: In this case the global variables *id_first* and *id_loc* will have been set to the appropriate values needed by the *id_lookup* routine.

string: In this case the global variables *id_first* and *id_loc* will have been set to the beginning and ending-plus-one locations in the buffer. The string ends with the first reappearance of its initial delimiter; thus, for example,

```
´This isn´´t a single string´
```

will be treated as two consecutive strings, the first being `´This isn´`.

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

xref_roman, *xref_wildcard*, *xref_typewriter*, *TeX_string*: The values of *id_first* and *id_loc* will be set so that the string in question appears in $buffer[id\_first\ ..\ (id\_loc - 1)]$.

module_name: In this case the global variable *cur_module* will point to the *byte_start* entry for the module name that has just been scanned.

If *get_next* sees '`@!`' or '`@?`', it sets *xref_switch* to *def_flag* or zero and goes on to the next token.

A global variable called *scanning_hex* is set *true* during the time that the letters `A` through `F` should be treated as if they were digits.

```
define exponent = ´200   { E or e following a digit }
define string = ´201   { Pascal string or WEB precomputed string }
define identifier = ´202   { Pascal identifier or reserved word }
⟨ Globals in the outer block 9 ⟩ +≡
cur_module: name_pointer;   { name of module just scanned }
scanning_hex: boolean;   { are we scanning a hexadecimal constant? }
```

94. ⟨ Set initial values 10 ⟩ +≡

```
scanning_hex ← false;
```

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

```
define up_to(#) ≡ # − 24, # − 23, # − 22, # − 21, # − 20, # − 19, # − 18, # − 17, # − 16, # − 15, # − 14, # − 13,
    # − 12, # − 11, # − 10, # − 9, # − 8, # − 7, # − 6, # − 5, # − 4, # − 3, # − 2, # − 1, #
function get_next: eight_bits; { produces the next input token }
  label restart, done, found;
  var c: eight_bits; { the current character }
    d: eight_bits; { the next character }
    j, k: 0 .. longest_name; { indices into mod_text }
  begin restart: if loc > limit then
    begin get_line;
    if input_has_ended then
      begin c ← new_module; goto found;
      end;
    end;
  c ← buffer[loc]; incr(loc);
  if scanning_hex then ⟨ Go to found if c is a hexadecimal digit, otherwise set scanning_hex ← false 96 ⟩;
  case c of
  "A", up_to("Z"), "a", up_to("z"): ⟨ Get an identifier 98 ⟩;
  "´", """": ⟨ Get a string 99 ⟩;
  "@": ⟨ Get control code and possible module name 100 ⟩;
  ⟨ Compress two-symbol combinations like ':=' 97 ⟩
  "␣", tab_mark: goto restart; { ignore spaces and tabs }
  "}": begin err_print(´!␣Extra␣}´); goto restart;
    end;
  othercases if c ≥ 128 then goto restart { ignore nonstandard characters }
    else do_nothing
  endcases;
found: debug if trouble_shooting then debug_help; gubed
  get_next ← c;
  end;
```

96. ⟨ Go to *found* if *c* is a hexadecimal digit, otherwise set *scanning_hex* ← *false* 96 ⟩ ≡

```
if ((c ≥ "0") ∧ (c ≤ "9")) ∨ ((c ≥ "A") ∧ (c ≤ "F")) then goto found
else scanning_hex ← false
```

This code is used in section 95.

97. Note that the following code substitutes `@{` and `@}` for the respective combinations '`(*`' and '`*)`'. Explicit braces should be used for TEX comments in Pascal text.

```
define compress(#) ≡
        begin if loc ≤ limit then
          begin c ← #; incr(loc);
          end;
        end
```

⟨ Compress two-symbol combinations like '`:=`' 97 ⟩ ≡

```
":": if buffer[loc] = "." then compress(double_dot)
  else if buffer[loc] = ")" then compress("]");
":": if buffer[loc] = "=" then compress(left_arrow);
"=": if buffer[loc] = "=" then compress(equivalence_sign);
">": if buffer[loc] = "=" then compress(greater_or_equal);
"<": if buffer[loc] = "=" then compress(less_or_equal)
  else if buffer[loc] = ">" then compress(not_equal);
"(": if buffer[loc] = "*" then compress(begin_comment)
  else if buffer[loc] = "." then compress("[");
"*": if buffer[loc] = ")" then compress(end_comment);
```

This code is used in section 95.

98. ⟨ Get an identifier 98 ⟩ ≡

```
begin if ((c = "E") ∨ (c = "e")) ∧ (loc > 1) then
  if (buffer[loc − 2] ≤ "9") ∧ (buffer[loc − 2] ≥ "0") then c ← exponent;
if c ≠ exponent then
  begin decr(loc); id_first ← loc;
  repeat incr(loc); d ← buffer[loc];
  until ((d < "0") ∨ ((d > "9") ∧ (d < "A")) ∨ ((d > "Z") ∧ (d < "a")) ∨ (d > "z")) ∧ (d ≠ "_");
  c ← identifier; id_loc ← loc;
  end;
end
```

This code is used in section 95.

99. A string that starts and ends with single or double quote marks is scanned by the following piece of the program.

⟨ Get a string 99 ⟩ ≡

```
begin id_first ← loc − 1;
repeat d ← buffer[loc]; incr(loc);
  if loc > limit then
    begin err_print('! String␣constant␣didn''t␣end'); loc ← limit; d ← c;
    end;
until d = c;
id_loc ← loc; c ← string;
end
```

This code is used in section 95.

100. After an `@` sign has been scanned, the next character tells us whether there is more work to do.

```
⟨ Get control code and possible module name 100 ⟩ ≡
  begin c ← control_code(buffer[loc]); incr(loc);
  if c = underline then
    begin xref_switch ← def_flag; goto restart;
    end
  else if c = no_underline then
      begin xref_switch ← 0; goto restart;
      end
    else if (c ≤ TeX_string) ∧ (c ≥ xref_roman) then ⟨ Scan to the next @> 106 ⟩
      else if c = hex then scanning_hex ← true
        else if c = module_name then ⟨ Scan the module name and make cur_module point to it 101 ⟩
          else if c = verbatim then ⟨ Scan a verbatim string 107 ⟩;
  end
```

This code is used in section 95.

101. The occurrence of a module name sets *xref_switch* to zero, because the module name might (for example) follow **var**.

```
⟨ Scan the module name and make cur_module point to it 101 ⟩ ≡
  begin ⟨ Put module name into mod_text[1 .. k] 103 ⟩;
  if k > 3 then
    begin if (mod_text[k] = ".") ∧ (mod_text[k − 1] = ".") ∧ (mod_text[k − 2] = ".") then
      cur_module ← prefix_lookup(k − 3)
    else cur_module ← mod_lookup(k);
    end
  else cur_module ← mod_lookup(k);
  xref_switch ← 0;
  end
```

This code is used in section 100.

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

```
⟨ Set initial values 10 ⟩ +≡
  mod_text[0] ← "␣";
```

103. ⟨ Put module name into *mod_text*[1 .. *k*] 103 ⟩ ≡

```
k ← 0;
loop begin if loc > limit then
    begin get_line;
    if input_has_ended then
      begin err_print('!␣Input␣ended␣in␣section␣name'); loc ← 1; goto done;
      end;
    end;
  d ← buffer[loc]; ⟨ If end of name, goto done 104 ⟩;
  incr(loc);
  if k < longest_name − 1 then incr(k);
  if (d = "␣") ∨ (d = tab_mark) then
    begin d ← "␣";
    if mod_text[k − 1] = "␣" then decr(k);
    end;
  mod_text[k] ← d;
  end;
done: ⟨ Check for overlong name 105 ⟩;
  if (mod_text[k] = "␣") ∧ (k > 0) then decr(k)
```

This code is used in section 101.

104. ⟨ If end of name, **goto** *done* 104 ⟩ ≡

```
if d = "@" then
  begin d ← buffer[loc + 1];
  if d = ">" then
    begin loc ← loc + 2; goto done;
    end;
  if (d = "␣") ∨ (d = tab_mark) ∨ (d = "*") then
    begin err_print('!␣Section␣name␣didn''t␣end'); goto done;
    end;
  incr(k); mod_text[k] ← "@"; incr(loc);   { now d = buffer[loc] again }
  end
```

This code is used in section 103.

105. ⟨ Check for overlong name 105 ⟩ ≡

```
if k ≥ longest_name − 2 then
  begin print_nl('!␣Section␣name␣too␣long:␣');
  for j ← 1 to 25 do print(xchr[mod_text[j]]);
  print('...'); mark_harmless;
  end
```

This code is used in section 103.

106. ⟨ Scan to the next `@>` 106 ⟩ ≡

```
begin id_first ← loc; buffer[limit + 1] ← "@";
while buffer[loc] ≠ "@" do incr(loc);
id_loc ← loc;
if loc > limit then
  begin err_print('!␣Control␣text␣didn''t␣end'); loc ← limit;
  end
else begin loc ← loc + 2;
  if buffer[loc − 1] ≠ ">" then err_print('!␣Control␣codes␣are␣forbidden␣in␣control␣text');
  end;
end
```

This code is used in section 100.

107. A verbatim Pascal string will be treated like ordinary strings, but with no surrounding delimiters. At the present point in the program we have $buffer[loc - 1] = verbatim$; we must set *id_first* to the beginning of the string itself, and *id_loc* to its ending-plus-one location in the buffer. We also set *loc* to the position just after the ending delimiter.

⟨ Scan a verbatim string 107 ⟩ ≡

```
begin id_first ← loc; incr(loc); buffer[limit + 1] ← "@"; buffer[limit + 2] ← ">";
while (buffer[loc] ≠ "@") ∨ (buffer[loc + 1] ≠ ">") do incr(loc);
if loc ≥ limit then err_print('!␣Verbatim␣string␣didn''t␣end');
id_loc ← loc; loc ← loc + 2;
end
```

This code is used in section 100.

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

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

```
⟨ Globals in the outer block 9 ⟩ +≡
next_control: eight_bits;   { control code waiting to be acting upon }
```

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

```
⟨ Phase I: Read all the user's text and store the cross references 109 ⟩ ≡
  phase_one ← true; phase_three ← false; reset_input; module_count ← 0; changed_module[0] ← false;
  skip_limbo; change_exists ← false;
  while ¬input_has_ended do ⟨ Store cross reference data for the current module 110 ⟩;
  changed_module[module_count] ← change_exists;   { the index changes if anything does }
  phase_one ← false;   { prepare for second phase }
  ⟨ Print error messages about unused or undefined module names 120 ⟩;
```

This code is used in section 261.

110.

```
⟨ Store cross reference data for the current module 110 ⟩ ≡
  begin incr(module_count);
  if module_count = max_modules then overflow('section␣number');
  changed_module[module_count] ← false;   { it will become true if any line changes }
  if buffer[loc − 1] = "*" then
    begin print('*', module_count : 1); update_terminal;   { print a progress report }
    end;
  ⟨ Store cross references in the TEX part of a module 113 ⟩;
  ⟨ Store cross references in the definition part of a module 115 ⟩;
  ⟨ Store cross references in the Pascal part of a module 117 ⟩;
  if changed_module[module_count] then change_exists ← true;
  end
```

This code is used in section 109.

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

The program uses the fact that our internal code numbers satisfy the relations *xref_roman* = *identifier* + *roman* and *xref_wildcard* = *identifier* + *wildcard* and *xref_typewriter* = *identifier* + *typewriter* and *normal* = 0. An implied '`@!`' is inserted after **function**, **procedure**, **program**, and **var**.

```
⟨ Functions scan_const and scan_exp 271 ⟩
procedure Pascal_xref; { makes cross references for Pascal identifiers }
  label exit, done, found, not_found;
  var p: name_pointer; { a referenced name }
    eq: 0 .. 3; { addition to dtype code }
  begin while next_control < format do
    begin if (next_control ≥ identifier) ∧ (next_control ≤ xref_typewriter) then
      begin p ← id_lookup(next_control − identifier);
      if next_control = identifier then
        if xref_switch ≠ 0 then
          if ilk[p] = normal then ⟨ Figure out the def_type and def_name, etc. 273 ⟩;
      new_xref(p); danger_zone ← (def_type = dtype_comma);
      if (ilk[p] = proc_like) ∨ (ilk[p] = var_like) then xref_switch ← def_flag; { implied '@!' }
      if ilk[p] = proc_like then
        begin def_name ← p; next_control ← get_next;
        if next_control ≠ identifier then goto done;
        if xref_switch = 0 then goto done;
        p ← id_lookup(normal); def_type ← dtype_colon_bold; new_xref(p);
        end;
      end;
    next_control ← get_next;
  done: if (next_control = "|") ∨ (next_control = "{") then return;
    end;
exit: end;
```

112. The *outer_xref* subroutine is like *Pascal_xref* but it begins with *next_control* $\neq$ `"|"` and ends with *next_control* $\geq$ *format*. Thus, it handles Pascal text with embedded comments.

```
procedure outer_xref; { extension of Pascal_xref }
  var bal: eight_bits; { brace level in comment }
  begin while next_control < format do
    if next_control ≠ "{" then Pascal_xref
    else begin bal ← skip_comment(1); next_control ← "|";
      while bal > 0 do
        begin Pascal_xref;
        if next_control = "|" then bal ← skip_comment(bal)
        else bal ← 0; { an error will be reported in phase two }
        end;
      end;
  end;
```

113. In the TEX part of a module, cross reference entries are made only for the identifiers in Pascal texts enclosed in `|...|`, or for control texts enclosed in `@^...@>` or `@.....@>` or `@:...@>`.

```
⟨ Store cross references in the TEX part of a module 113 ⟩ ≡
  repeat next_control ← skip_TeX;
    case next_control of
    underline: xref_switch ← def_flag;
    no_underline: xref_switch ← 0;
    "|": Pascal_xref;
    xref_roman, xref_wildcard, xref_typewriter, module_name: begin loc ← loc − 2;
      next_control ← get_next;  { scan to @> }
      if next_control ≠ module_name then new_xref(id_lookup(next_control − identifier));
      end;
    othercases do_nothing
    endcases;
  until next_control ≥ format
```

This code is used in section 110.

114. During the definition and Pascal parts of a module, cross references are made for all identifiers except reserved words; however, the identifiers in a format definition are referenced even if they are reserved. The TEX code in comments is, of course, ignored, except for Pascal portions enclosed in `|...|`; the text of a module name is skipped entirely, even if it contains `|...|` constructions.

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

```
⟨ Globals in the outer block 9 ⟩ +≡
lhs, rhs: name_pointer;  { indices into byte_start for format identifiers }
```

115. When we get to the following code we have *next_control* ≥ *format*.

```
⟨ Store cross references in the definition part of a module 115 ⟩ ≡
  while next_control ≤ definition do  { format or definition }
    begin xref_switch ← def_flag;  { implied @! }
    if next_control = definition then
      begin next_control ← get_next;
      if next_control = identifier then
        begin lhs ← id_lookup(normal); next_control ← get_next;
        if (next_control = equivalence_sign) ∨ (next_control = "(") then
          begin def_type ← dtype_macro; new_xref(lhs);
          end
        else if next_control = "=" then
            begin xref_switch ← 0; next_control ← get_next; def_val[lhs] ← scan_exp;
            def_name ← def_val[lhs]; def_type ← dtype_const; xref_switch ← def_flag;
            if abs(def_name) ≥ 32768 then def_type ← dtype_macro
            else if def_name < 0 then def_name ← def_name + 65536;
            new_xref(lhs);
            end;
        end;
      end
    else ⟨ Process a format definition 116 ⟩;
    outer_xref;
    end
```

This code is used in section 110.

116. Error messages for improper format definitions will be issued in phase two. Our job in phase one is to define the *ilk* of a properly formatted identifier, and to fool the *new_xref* routine into thinking that the identifier on the right-hand side of the format definition is not a reserved word.

```
⟨ Process a format definition 116 ⟩ ≡
  begin next_control ← get_next;
  if next_control = identifier then
    begin lhs ← id_lookup(normal); ilk[lhs] ← normal; new_xref(lhs); next_control ← get_next;
    if next_control = equivalence_sign then
      begin next_control ← get_next;
      if next_control = identifier then
        begin rhs ← id_lookup(normal); ilk[lhs] ← ilk[rhs]; ilk[rhs] ← normal; new_xref(rhs);
        ilk[rhs] ← ilk[lhs]; next_control ← get_next;
        end;
      end;
    end;
  end
```

This code is used in section 115.

117. Finally, when the TEX and definition parts have been treated, we have $next\_control \geq begin\_Pascal$.

```
⟨ Store cross references in the Pascal part of a module 117 ⟩ ≡
  if next_control ≤ module_name then { begin_Pascal or module_name }
    begin if next_control = begin_Pascal then mod_xref_switch ← 0
    else mod_xref_switch ← def_flag;
    repeat if next_control = module_name then new_mod_xref(cur_module);
      next_control ← get_next; outer_xref;
    until next_control > module_name;
    end
```

This code is used in section 110.

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

```
⟨ Globals in the outer block 9 ⟩ +≡
cur_xref: xref_number; { temporary cross reference pointer }
```

119. The following recursive procedure walks through the tree of module names and prints out anomalies.

```
procedure mod_check(p : name_pointer); { print anomalies in subtree p }
  begin if p > 0 then
    begin mod_check(llink[p]);
    cur_xref ← xref[p];
    if num(cur_xref) < def_flag then
      begin print_nl('!␣Never␣defined:␣<'); print_id(p); print('>'); mark_harmless;
      end;
    while num(cur_xref) ≥ def_flag do cur_xref ← xlink(cur_xref);
    if cur_xref = 0 then
      begin print_nl('!␣Never␣used:␣<'); print_id(p); print('>'); mark_harmless;
      end;
    mod_check(rlink[p]);
    end;
  end;
```

120. ⟨ Print error messages about unused or undefined module names 120 ⟩ ≡ *mod_check*(*root*)

This code is used in section 109.

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

```
⟨ Globals in the outer block 9 ⟩ +≡
out_buf: array [0 .. line_length] of ASCII_code;  { assembled characters }
out_ptr: 0 .. line_length;  { number of characters in out_buf }
out_line: integer;  { coordinates of next line to be output }
```

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

```
procedure flush_buffer(b : eight_bits; per_cent, carryover : boolean);
          { outputs out_buf[1 .. b], where b ≤ out_ptr }
  label done, found;
  var j, k: 0 .. line_length;
  begin j ← b;
  if ¬per_cent then  { remove trailing blanks }
    loop begin if j = 0 then goto done;
      if out_buf[j] ≠ "␣" then goto done;
      decr(j);
      end;
done: for k ← 1 to j do write(tex_file, xchr[out_buf[k]]);
  if per_cent then write(tex_file, xchr["%"]);
  write_ln(tex_file); incr(out_line);
  if carryover then
    for k ← 1 to j do
      if out_buf[k] = "%" then
        if (k = 1) ∨ (out_buf[k − 1] ≠ "\") then  { comment mode should be preserved }
          begin out_buf[b] ← "%"; decr(b); goto found;
          end;
found: if (b < out_ptr) then
    for k ← b + 1 to out_ptr do out_buf[k − b] ← out_buf[k];
  out_ptr ← out_ptr − b;
  end;
```

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

```
procedure finish_line; { do this at the end of a line }
  label exit;
  var k: 0 .. buf_size; { index into buffer }
  begin if out_ptr > 0 then flush_buffer(out_ptr, false, false)
  else begin for k ← 0 to limit do
      if (buffer[k] ≠ "␣") ∧ (buffer[k] ≠ tab_mark) then return;
    flush_buffer(0, false, false);
    end;
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 twimac-web`'.

```
⟨ Set initial values 10 ⟩ +≡
  out_ptr ← 1; out_line ← 1; out_buf[1] ← "b"; write(tex_file, ´\input␣twimac-we´);
```

125. When we wish to append the character c to the output buffer, we write '$out(c)$'; this will cause the buffer to be emptied if it was already full. Similarly, '$out2(c_1)(c_2)$' appends a pair of characters. A line break will occur at a space or after a single-nonletter TEX control sequence.

```
  define oot(#) ≡
      if out_ptr = line_length then break_out;
      incr(out_ptr); out_buf[out_ptr] ← #;
  define oot1(#) ≡ oot(#) end
  define oot2(#) ≡ oot(#) oot1
  define oot3(#) ≡ oot(#) oot2
  define oot4(#) ≡ oot(#) oot3
  define oot5(#) ≡ oot(#) oot4
  define oot6(#) ≡ oot(#) oot5
  define oot7(#) ≡ oot(#) oot6
  define oot8(#) ≡ oot(#) oot7
  define oot9(#) ≡ oot(#) oot8
  define oot10(#) ≡ oot(#) oot9
  define oot11(#) ≡ oot(#) oot10
  define oot12(#) ≡ oot(#) oot11
  define out ≡ begin oot1
  define out2 ≡ begin oot2
  define out3 ≡ begin oot3
  define out4 ≡ begin oot4
  define out5 ≡ begin oot5
  define out6 ≡ begin oot6
  define out7 ≡ begin oot7
  define out8 ≡ begin oot8
  define out9 ≡ begin oot9
  define out10 ≡ begin oot10
  define out11 ≡ begin oot11
  define out12 ≡ begin oot12
```

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

```
⟨ Set initial values 10 ⟩ +≡
  out_buf[0] ← "\";
```

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

```
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] ≠ "\") then { in this case k > 1 }
      begin flush_buffer(k − 1, true, true); return;
      end;
    decr(k);
    end;
exit: end;
```

128. We get to this module only in unusual cases that the entire output line consists of a string of backslashes followed by a string of nonblank non-backslashes. In such cases it is almost always safe to break the line by putting a "%" just before the last character.

```
⟨ Print warning message, break the line, return 128 ⟩ ≡
  begin print_nl('!␣Line␣had␣to␣be␣broken␣(output␣l.', out_line : 1); print_ln('):');
  for k ← 1 to out_ptr − 1 do print(xchr[out_buf[k]]);
  new_line; mark_harmless; flush_buffer(out_ptr − 1, true, true); return;
  end
```

This code is used in section 127.

129. Here is a procedure that outputs a module number in decimal notation.

```
⟨ Globals in the outer block 9 ⟩ +≡
dig: array [0 .. 4] of 0 .. 9; { digits to output }
```

130. The number to be converted by *out_mod* is known to be less than *def_flag*, so it cannot have more than five decimal digits.

```
procedure out_mod(m : integer); { output a module number }
  var k: 0 .. 5; { index into dig }
    a: integer; { accumulator }
  begin k ← 0; a ← m;
  repeat dig[k] ← a mod 10; a ← a div 10; incr(k);
  until a = 0;
  repeat decr(k); out(dig[k] + "0");
  until k = 0;
  end;
```

131. The *out_name* subroutine is used to output an identifier or index entry, enclosing it in braces.

```
procedure out_name(p : name_pointer); { outputs a name }
  var k: 0 .. max_bytes; { index into byte_mem }
    w: 0 .. ww − 1; { row of byte_mem }
  begin out("{"); w ← p mod ww;
  for k ← byte_start[p] to byte_start[p + ww] − 1 do
    begin if byte_mem[w, k] = "_" then out("\");
    out(byte_mem[w, k]);
    end;
  out("}");
  end;
```

132. Routines that copy TEX material. During phase two, we use the subroutines *copy_limbo*, *copy_TeX*, and *copy_comment* in place of the analogous *skip_limbo*, *skip_TeX*, and *skip_comment* that were used in phase one.

The *copy_limbo* routine, for example, takes TEX material that is not part of any module and transcribes it almost verbatim to the output file. No '@' signs should occur in such material except in '@@' pairs; such pairs are replaced by singletons.

```
procedure copy_limbo; { copy TEX code until the next module begins }
  label exit;
  var c: ASCII_code; { character following @ sign }
  begin loop
    if loc > limit then
      begin finish_line; get_line;
      if input_has_ended then return;
      end
    else begin buffer[limit + 1] ← "@"; ⟨ Copy up to control code, return if finished 133 ⟩;
      end;
exit: end;
```

133. ⟨ Copy up to control code, **return** if finished 133 ⟩ ≡

```
  while buffer[loc] ≠ "@" do
    begin out(buffer[loc]); incr(loc);
    end;
  if loc ≤ limit then
    begin loc ← loc + 2; c ← buffer[loc − 1];
    if (c = "␣") ∨ (c = tab_mark) ∨ (c = "*") then return;
    if (c ≠ "z") ∧ (c ≠ "Z") then
      begin out("@");
      if c ≠ "@" then err_print('!␣Double␣@␣required␣outside␣of␣sections');
      end;
    end
```

This code is used in section 132.

134. The *copy_TeX* routine processes the TEX code at the beginning of a module; for example, the words you are now reading were copied in this way. It returns the next control code or '|' found in the input.

```
function copy_TeX: eight_bits; { copy pure TEX material }
  label done;
  var c: eight_bits; { control code found }
  begin loop
    begin if loc > limit then
      begin finish_line; get_line;
      if input_has_ended then
        begin c ← new_module; goto done;
        end;
      end;
    buffer[limit + 1] ← "@"; ⟨ Copy up to '|' or control code, goto done if finished 135 ⟩;
    end;
done: copy_TeX ← c;
  end;
```

135. We don't copy spaces or tab marks into the beginning of a line. This makes the test for empty lines in *finish_line* work.

```
⟨ Copy up to '|' or control code, goto done if finished 135 ⟩ ≡
  repeat c ← buffer[loc]; incr(loc);
    if c = "|" then goto done;
    if c ≠ "@" then
      begin out(c);
      if (out_ptr = 1) ∧ ((c = "␣") ∨ (c = tab_mark)) then decr(out_ptr);
      end;
  until c = "@";
  if loc ≤ limit then
    begin c ← control_code(buffer[loc]); incr(loc); goto done;
    end
```

This code is used in section 134.

136. The *copy_comment* uses and returns a brace-balance value, following the conventions of *skip_comment* above. Instead of copying the TEX material into the output buffer, this procedure copies it into the token memory. The abbreviation *app_tok*(*t*) is used to append token *t* to the current token list, and it also makes sure that it is possible to append at least one further token without overflow.

```
  define app_tok(#) ≡
           begin if tok_ptr + 2 > max_toks then overflow('token');
           tok_mem[tok_ptr] ← #; incr(tok_ptr);
           end

function copy_comment(bal : eight_bits): eight_bits;  { copies TEX code in comments }
  label done;
  var c: ASCII_code;  { current character being copied }
  begin loop
    begin if loc > limit then
      begin get_line;
      if input_has_ended then
        begin err_print('! Input ended in mid-comment'); loc ← 1; ⟨ Clear bal and goto done 138 ⟩;
        end;
      end;
    c ← buffer[loc]; incr(loc);
    if c = "|" then goto done;
    app_tok(c); ⟨ Copy special things when c = "@", "\", "{", "}"; goto done at end 137 ⟩;
    end;
done: copy_comment ← bal;
  end;
```

137. ⟨ Copy special things when $c =$ `"@"`, `"\"`, `"{"`, `"}"`; **goto** *done* at end 137 ⟩ ≡

```
if c = "@" then
  begin incr(loc);
  if buffer[loc − 1] ≠ "@" then
    begin err_print('!␣Illegal␣use␣of␣@␣in␣comment'); loc ← loc − 2; decr(tok_ptr);
    ⟨ Clear bal and goto done 138 ⟩;
    end;
  end
else if (c = "\") ∧ (buffer[loc] ≠ "@") then
    begin app_tok(buffer[loc]); incr(loc);
    end
  else if c = "{" then incr(bal)
    else if c = "}" then
        begin decr(bal);
        if bal = 0 then goto done;
        end
```

This code is used in section 136.

138. When the comment has terminated abruptly due to an error, we output enough right braces to keep TEX happy.

⟨ Clear *bal* and **goto** *done* 138 ⟩ ≡

```
app_tok("␣");  { this is done in case the previous character was '\' }
repeat app_tok("}"); decr(bal);
until bal = 0;
goto done;
```

This code is used in sections 136 and 137.

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

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

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

A production applies if the category codes have a given pattern. For example, one of the productions is

$$\textit{open math semi} \;\rightarrow\; \textit{open math}$$

and it means that three consecutive scraps whose respective categories are *open*, *math*, and *semi* are converted to two scraps whose categories are *open* and *math*. This production also has an associated rule that tells how to combine the translation parts:

$$O_2 = O_1$$
$$M_2 = M_1\, S \texttt{\textbackslash,}\ \textit{opt}\ \texttt{5}$$

This means that the *open* scrap has not changed, while the new *math* scrap has a translation M_2 composed of the translation M_1 of the original *math* scrap followed by the translation S of the *semi* scrap followed by '`\,`' followed by '*opt*' followed by '`5`'. (In the TEX file, this will specify an additional thin space after the semicolon, followed by an optional line break with penalty 50.) Translation rules use subscripts to distinguish between translations of scraps whose categories have the same initial letter; these subscripts are assigned from left to right.

WEAVE also has the production rule

$$\textit{semi} \;\rightarrow\; \textit{terminator}$$

(meaning that a semicolon can terminate a Pascal statement). Since productions are applied from left to right, this rule will be activated only if the *semi* is not preceded by scraps that match other productions; in particular, a *semi* that is preceded by '*open math*' will have disappeared because of the production above, and such semicolons do not act as statement terminators. This incidentally is how WEAVE is able to treat semicolons in two distinctly different ways, the first of which is intended for semicolons in the parameter list of a procedure declaration.

The translation rule corresponding to *semi* $\rightarrow$ *terminator* is

$$T = S$$

but we shall not mention translation rules in the common case that the translation of the new scrap on the right-hand side is simply the concatenation of the disappearing scraps on the left-hand side.

140. Here is a list of the category codes that scraps can have.

```
define simp = 1 { the translation can be used both in horizontal mode and in math mode of TEX }
define math = 2 { the translation should be used only in TEX math mode }
define intro = 3 { a statement is expected to follow this, after a space and an optional break }
define open = 4 { denotes an incomplete parenthesized quantity to be used in math mode }
define beginning = 5 { denotes an incomplete compound statement to be used in horizontal mode }
define close = 6 { ends a parenthesis or compound statement }
define alpha = 7 { denotes the beginning of a clause }
define omega = 8 { denotes the ending of a clause and possible comment following }
define semi = 9 { denotes a semicolon and possible comment following it }
define terminator = 10 { something that ends a statement or declaration }
define stmt = 11 { denotes a statement or declaration including its terminator }
define cond = 12 { precedes an if clause that might have a matching else }
define clause = 13 { precedes a statement after which indentation ends }
define colon = 14 { denotes a colon }
define exp = 15 { stands for the E in a floating point constant }
define proc = 16 { denotes a procedure or program or function heading }
define case_head = 17 { denotes a case statement or record heading }
define record_head = 18 { denotes a record heading without indentation }
define var_head = 19 { denotes a variable declaration heading }
define elsie = 20 { else }
define casey = 21 { case }
define mod_scrap = 22 { denotes a module name }
debug procedure print_cat(c : eight_bits); { symbolic printout of a category }
begin case c of
simp: print('simp');
math: print('math');
intro: print('intro');
open: print('open');
beginning: print('beginning');
close: print('close');
alpha: print('alpha');
omega: print('omega');
semi: print('semi');
terminator: print('terminator');
stmt: print('stmt');
cond: print('cond');
clause: print('clause');
colon: print('colon');
exp: print('exp');
proc: print('proc');
case_head: print('casehead');
record_head: print('recordhead');
var_head: print('varhead');
elsie: print('elsie');
casey: print('casey');
mod_scrap: print('module');
othercases print('UNKNOWN')
endcases;
end;
gubed
```

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

break_space denotes an optional line break or an en space;

force denotes a line break;

big_force denotes a line break with additional vertical space;

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

backup denotes a backspace of one em;

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

indent causes future lines to be indented one more em;

outdent causes future lines to be indented one less em.

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

The tokens *math_rel*, *math_bin*, *math_op* will be translated into `\mathrel{`, `\mathbin{`, and `\mathop{`, respectively. Other control sequences in the TEX output will be '`\\{...}`' surrounding identifiers, '`\&{...}`' surrounding reserved words, '`\.{...}`' surrounding strings, '`\C{...}`*force*' surrounding comments, and '`\X`n`:...\X`' surrounding module names, where n is the module number.

define *math_bin* = ′203
define *math_rel* = ′204
define *math_op* = ′205
define *big_cancel* = ′206 { like *cancel*, also overrides spaces }
define *cancel* = ′207 { overrides *backup*, *break_space*, *force*, *big_force* }
define *indent* = *cancel* + 1 { one more tab (`\1`) }
define *outdent* = *cancel* + 2 { one less tab (`\2`) }
define *opt* = *cancel* + 3 { optional break in mid-statement (`\3`) }
define *backup* = *cancel* + 4 { stick out one unit to the left (`\4`) }
define *break_space* = *cancel* + 5 { optional break between statements (`\5`) }
define *force* = *cancel* + 6 { forced break between statements (`\6`) }
define *big_force* = *cancel* + 7 { forced break with additional space (`\7`) }
define *end_translation* = *big_force* + 1 { special sentinel token at end of list }

142. The raw input is converted into scraps according to the following table, which gives category codes followed by the translations. Sometimes a single item of input produces more than one scrap. (The symbol '**' stands for '\&{identifier}', i.e., the identifier itself treated as a reserved word. In a few cases the category is given as '*comment*'; this is not an actual category code, it means that the translation will be treated as a comment, as explained below.)

<>	*math*: \I
<=	*math*: \L
>=	*math*: \G
:=	*math*: \K
==	*math*: \S
(*	*math*: \B
*)	*math*: \T
(.	*open*: [
.)	*close*:]
" string "	*simp*: \.{" modified string "}
´ string ´	*simp*: \.{\´ modified string \´}
@= string @>	*simp*: \={ modified string }
#	*math*: \#
$	*math*: \$
_	*math*: _
%	*math*: \%
^	*math*: \^
(	*open*: (
)	*close*:)
[	*open*: [
]	*close*:]
*	*math*: \ast
,	*math*: , *opt* 9
..	*math*: \to
.	*simp*: .
:	*colon*: :
;	*semi*: ;
identifier	*simp*: \\{ identifier }
E in constant	*exp*: \E{
digit *d*	*simp*: *d*
other character *c*	*math*: *c*
and	*math*: \W
array	*alpha*: **
begin	*beginning*: *force* ** *cancel* *intro*:
case	*casey*: *alpha*: *force* **
const	*intro*: *force backup* **
div	*math*: *math_bin* ** }
do	*omega*: **
downto	*math*: *math_rel* ** }
else	*terminator*: *elsie*: *force backup* **
end	*terminator*: *close*: *force* **
file	*alpha*: **
for	*alpha*: *force* **
function	*proc*: *force backup* ** *cancel* *intro*: *indent* \␣
goto	*intro*: **
if	*cond*: *alpha*: *force* **
in	*math*: \in

`label` *intro*: *force backup* `**`
`mod` *math*: *math_bin* `**}`
`nil` *simp*: `**`
`not` *math*: `\R`
`of` *omega*: `**`
`or` *math*: `\V`
`packed` *intro*: `**`
`procedure` *proc*: *force backup* `**` *cancel* *intro*: *indent* `\␣`
`program` *proc*: *force backup* `**` *cancel* *intro*: *indent* `\␣`
`record` *record_head*: `**` *intro*:
`repeat` *beginning*: *force indent* `**` *cancel* *intro*:
`set` *alpha*: `**`
`then` *omega*: `**`
`to` *math*: *math_rel* `**}`
`type` *intro*: *force backup* `**`
`until` *terminator*: *close*: *force backup* `**` *clause*:
`var` *var_head*: *force backup* `**` *cancel* *intro*:
`while` *alpha*: *force* `**`
`with` *alpha*: *force* `**`
`xclause` *alpha*: *force* `\~` *omega*: `**`
`@´` const *simp*: `\O{`const`}`
`@"` const *simp*: `\H{`const`}`
`@$` *simp*: `\)`
`@\` *simp*: `\]`
`@,` *math*: `\,`
`@t` stuff `@>` *simp*: `\hbox{` stuff `}`
`@<` module `@>` *mod_scrap*: `\X`*n*`:` module `\X`
`@#` *comment*: *big_force*
`@/` *comment*: *force*
`@|` *simp*: *opt* `0`
`@+` *comment*: *big_cancel* `\␣` *big_cancel*
`@;` *semi*:
`@&` *math*: `\J`
`@{` *math*: `\B`
`@}` *math*: `\T`

When a string is output, certain characters are preceded by '\' signs so that they will print properly.

A comment in the input will be combined with the preceding *omega* or *semi* scrap, or with the following *terminator* scrap, if possible; otherwise it will be inserted as a separate *terminator* scrap. An additional "comment" is effectively appended at the end of the Pascal text, just before translation begins; this consists of a *cancel* token in the case of Pascal text in `|` ... `|`, otherwise it consists of a *force* token.

From this table it is evident that `WEAVE` will parse a lot of non-Pascal programs. For example, the reserved words '`for`' and '`array`' are treated in an identical way by `WEAVE` from a syntactic standpoint, and semantically they are equivalent except that a forced line break occurs just before '**for**'; Pascal programmers may well be surprised at this similarity. The idea is to keep `WEAVE`'s rules as simple as possible, consistent with doing a reasonable job on syntactically correct Pascal programs. The production rules below have been formulated in the same spirit of "almost anything goes."

143. Here is a table of all the productions. The reader can best get a feel for how they work by trying them out by hand on small examples; no amount of explanation will be as effective as watching the rules in action. Parsing can also be watched by debugging with '`@2`'.

Production categories ⟦translations⟧	Remarks
1 alpha math colon → alpha math	e.g., **case** v : *boolean* **of**
2 alpha math omega → clause ⟦$C = A\,␣\,\$\,M\,\$\,␣\,indent\ O$⟧	e.g., **while** $x > 0$ **do**
3 alpha omega → clause ⟦$C = A\,␣\,indent\ O$⟧	e.g., **file of**
4 alpha simp → alpha math	convert to math mode
5 beginning close (*terminator* or *stmt*) *→ stmt*	compound statement ends
6 beginning stmt → beginning ⟦$B_2 = B_1\ break\_space\ S$⟧	compound statement grows
7 case_head casey clause → case_head ⟦$C_4 = C_1\ outdent\ C_2\ C_3$⟧	variant records
8 case_head close terminator → stmt ⟦$S = C_1\ cancel\ outdent\ C_2\ T$⟧	end of case statement
9 case_head stmt → case_head ⟦$C_2 = C_1\ force\ S$⟧	case statement grows
10 casey clause → case_head	beginning of case statement
11 clause stmt → stmt ⟦$S_2 = C\ break\_space\ S_1\ cancel\ outdent\ force$⟧	end of controlled statement
12 cond clause stmt elsie → clause ⟦$C_3 = C_1\,C_2\ break\_space\ S\,E\,␣\,cancel$⟧	complete conditional
13 cond clause stmt → stmt ⟦$S_2 = C_1\,C_2\ break\_space\ S_1\ cancel\ outdent\ force$⟧	incomplete conditional
14 elsie → intro	unmatched else
15 exp math simp → math* ⟦$M_2 = E\,M_1\,S\,\}$⟧	signed exponent
16 exp simp → math* ⟦$M = E\,S\,\}$⟧	unsigned exponent
17 intro stmt → stmt ⟦$S_2 = I\,␣\,opt\ 7\ cancel\ S_1$⟧	labeled statement, etc.
18 math close → stmt close ⟦$S = \$\,M\,\$$⟧	end of field list
19 math colon → intro ⟦$I = force\ backup\ \$\,M\,\$\,C$⟧	compound label
20 math math → math	simple concatenation
21 math simp → math	simple concatenation
22 math stmt → stmt ⟦$S_2 = \$\,M\,\$\ indent\ break\_space\ S_1\ cancel\ outdent\ force$⟧	macro or type definition
23 math terminator → stmt ⟦$S = \$\,M\,\$\,T$⟧	statement involving math
24 mod_scrap (*terminator* or *semi*) *→ stmt* ⟦$S = M\,T\ force$⟧	module like a statement
25 mod_scrap → simp	module unlike a statement
26 open case_head close → math ⟦$M = O\,\$\ cancel\ C_1\ cancel\ outdent\ \$\,C_2$⟧	case in field list
27 open close → math ⟦$M = O\,\backslash\,,\,C$⟧	empty set [ ]
28 open math case_head close → math ⟦$M_2 = O\,M_1\,\$\ cancel\ C_1\ cancel\ outdent\ \$\,C_2$⟧	case in field list
29 open math close → math	parenthesized group
30 open math colon → open math	colon in parentheses
31 open math proc intro → open math ⟦$M_2 = M_1\ math\_op\ cancel\ P\,\}$⟧	**procedure** in parentheses
32 open math semi → open math ⟦$M_2 = M_1\,S\,\backslash\,,\,opt\ 5$⟧	semicolon in parentheses
33 open math var_head intro → open math ⟦$M_2 = M_1\ math\_op\ cancel\ V\,\}$⟧	**var** in parentheses
34 open proc intro → open math ⟦$M = math\_op\ cancel\ P\,\}$⟧	**procedure** in parentheses
35 open simp → open math	convert to math mode
36 open stmt close → math ⟦$M = O\,\$\ cancel\ S\ cancel\ \$\,C$⟧	field list
37 open var_head intro → open math ⟦$M = math\_op\ cancel\ V\,\}$⟧	**var** in parentheses
38 proc beginning close terminator → stmt ⟦$S = P\ cancel\ outdent\ B\,C\,T$⟧	end of procedure declaration
39 proc stmt → proc ⟦$P_2 = P_1\ break\_space\ S$⟧	procedure declaration grows
40 record_head intro casey → casey ⟦$C_2 = R\,I\,␣\,cancel\ C_1$⟧	**record case** ...
41 record_head → case_head ⟦$C = indent\ R\ cancel$⟧	other **record** structures
42 semi → terminator	semicolon after statement
43 simp close → stmt close	end of field list
44 simp colon → intro ⟦$I = force\ backup\ S\,C$⟧	simple label
45 simp math → math	simple concatenation

46 *simp mod_scrap* $\rightarrow$ *mod_scrap*	in emergencies
47 *simp simp* $\rightarrow$ *simp*	simple concatenation
48 *simp terminator* $\rightarrow$ *stmt*	simple statement
49 *stmt stmt* $\rightarrow$ *stmt* $\quad [\![S_3 = S_1 \; break\_space \; S_2]\!]$	adjacent statements
50 *terminator* $\rightarrow$ *stmt*	empty statement
51 *var_head beginning* $\rightarrow$ *stmt beginning*	end of variable declarations
52 *var_head math colon* $\rightarrow$ *var_head intro* $\quad [\![I = \$\, M \, \$\, C]\!]$	variable declaration
53 *var_head simp colon* $\rightarrow$ *var_head intro*	variable declaration
54 *var_head stmt* $\rightarrow$ *var_head* $\quad [\![V_2 = V_1 \; break\_space \; S]\!]$	variable declarations grow

Translations are not specified here when they are simple concatenations of the scraps that change. For example, the full translation of '*open math colon* $\rightarrow$ *open math*' is $O_2 = O_1$, $M_2 = M_1 C$.

The notation '*simp**', in the *exp*-related productions above, stands for a *simp* scrap that isn't followed by another *simp*.

144. Implementing the productions. When Pascal text is to be processed with the grammar above, we put its initial scraps $s_1 \ldots s_n$ into two arrays $cat[1 \,..\, n]$ and $trans[1 \,..\, n]$. The value of $cat[k]$ is simply a category code from the list above; the value of $trans[k]$ is a text pointer, i.e., an index into *tok_start*. Our production rules have the nice property that the right-hand side is never longer than the left-hand side. Therefore it is convenient to use sequential allocation for the current sequence of scraps. Five pointers are used to manage the parsing:

pp (the parsing pointer) is such that we are trying to match the category codes $cat[pp]\ cat[pp+1] \ldots$ to the left-hand sides of productions.

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

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

The *trans* array elements are declared to be of type 0 .. 10239 instead of type *text_pointer*, because the final sorting phase of `WEAVE` uses this array to contain elements of type *name_pointer*. Both of these types are subranges of 0 .. 10239.

```
⟨ Globals in the outer block 9 ⟩ +≡
cat: array [0 .. max_scraps] of eight_bits;   { category codes of scraps }
trans: array [0 .. max_scraps] of 0 .. 10239;   { translation texts of scraps }
pp: 0 .. max_scraps;   { current position for reducing productions }
scrap_base: 0 .. max_scraps;   { beginning of the current scrap sequence }
scrap_ptr: 0 .. max_scraps;   { ending of the current scrap sequence }
lo_ptr: 0 .. max_scraps;   { last scrap that has been examined }
hi_ptr: 0 .. max_scraps;   { first scrap that has not been examined }
  stat max_scr_ptr: 0 .. max_scraps;   { largest value assumed by scrap_ptr }
  tats
```

145.

```
⟨ Set initial values 10 ⟩ +≡
  scrap_base ← 1; scrap_ptr ← 0;
  stat max_scr_ptr ← 0; tats
```

146. Token lists in *tok_mem* are composed of the following kinds of items for TEX output.

- ASCII codes and special codes like *force* and *math_rel* represent themselves;
- *id_flag* + *p* represents \\{identifier *p*};
- *res_flag* + *p* represents \&{identifier *p*};
- *mod_flag* + *p* represents module name *p*;
- *tok_flag* + *p* represents token list number *p*;
- *inner_tok_flag* + *p* represents token list number *p*, to be translated without line-break controls.

```
define id_flag = 10240 { signifies an identifier }
define res_flag = id_flag + id_flag { signifies a reserved word }
define mod_flag = res_flag + id_flag { signifies a module name }
define tok_flag ≡ mod_flag + id_flag { signifies a token list }
define inner_tok_flag ≡ tok_flag + id_flag { signifies a token list in '| ... |' }
define lbrace ≡ xchr["{"] { this avoids possible Pascal compiler confusion }
define rbrace ≡ xchr["}"] { because these braces might occur within comments }
debug procedure print_text(p : text_pointer); { prints a token list }
var j: 0 .. max_toks; { index into tok_mem }
  r: 0 .. id_flag − 1; { remainder of token after the flag has been stripped off }
begin if p ≥ text_ptr then print('BAD')
else for j ← tok_start[p] to tok_start[p + 1] − 1 do
    begin r ← tok_mem[j] mod id_flag;
    case tok_mem[j] div id_flag of
    1: begin print('\\', lbrace); print_id(r); print(rbrace);
      end; { id_flag }
    2: begin print('\&', lbrace); print_id(r); print(rbrace);
      end; { res_flag }
    3: begin print('<'); print_id(r); print('>');
      end; { mod_flag }
    4: print('[[', r : 1, ']]'); { tok_flag }
    5: print('|[[', r : 1, ']]|'); { inner_tok_flag }
    othercases ⟨ Print token r in symbolic form 147 ⟩
    endcases;
    end;
end;
gubed
```

147. ⟨ Print token r in symbolic form 147 ⟩ ≡

```
case r of
math_bin: print('\mathbin', lbrace);
math_rel: print('\mathrel', lbrace);
math_op: print('\mathop', lbrace);
big_cancel: print('[ccancel]');
cancel: print('[cancel]');
indent: print('[indent]');
outdent: print('[outdent]');
backup: print('[backup]');
opt: print('[opt]');
break_space: print('[break]');
force: print('[force]');
big_force: print('[fforce]');
end_translation: print('[quit]');
othercases print(xchr[r])
endcases
```

This code is used in section 146.

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
define reduce(#) ≡ red(#); production
define production_end(#) ≡
        debug prod(#)
        gubed;
    goto found;
    end
define squash(#) ≡
    begin sq(#); production_end
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)
```

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

The code here is more complicated than it need be, since some popular Pascal compilers are unable to deal with procedures that contain a lot of program text. The *translate* procedure, which incorporates the **case** statement here, would become too long for those compilers if we did not do something to split the cases into parts. Therefore a separate procedure called *five_cases* has been introduced. This auxiliary procedure contains approximately half of the program text that *translate* would otherwise have had. There's also a procedure called *alpha_cases*, which turned out to be necessary because the best two-way split wasn't good enough. The procedure could be split further in an analogous manner, but the present scheme works on all compilers known to the author.

```
⟨ Match a production at pp, or increase pp if there is no match 149 ⟩ ≡
  if cat[pp] ≤ alpha then
    if cat[pp] < alpha then five_cases else alpha_cases
  else begin case cat[pp] of
    case_head: ⟨ Cases for case_head 153 ⟩;
    casey: ⟨ Cases for casey 154 ⟩;
    clause: ⟨ Cases for clause 155 ⟩;
    cond: ⟨ Cases for cond 156 ⟩;
    elsie: ⟨ Cases for elsie 157 ⟩;
    exp: ⟨ Cases for exp 158 ⟩;
    mod_scrap: ⟨ Cases for mod_scrap 161 ⟩;
    proc: ⟨ Cases for proc 164 ⟩;
    record_head: ⟨ Cases for record_head 165 ⟩;
    semi: ⟨ Cases for semi 166 ⟩;
    stmt: ⟨ Cases for stmt 168 ⟩;
    terminator: ⟨ Cases for terminator 169 ⟩;
    var_head: ⟨ Cases for var_head 170 ⟩;
    othercases do_nothing
    endcases;
    incr(pp);   { if no match was found, we move to the right }
  found: end
```

This code is used in section 175.

150. Here are the procedures that need to be present for the reason just explained.

```
⟨ Declaration of subprocedures for translate 150 ⟩ ≡
procedure five_cases; { handles almost half of the syntax }
  label found;
  begin case cat[pp] of
  beginning: ⟨ Cases for beginning 152 ⟩;
  intro: ⟨ Cases for intro 159 ⟩;
  math: ⟨ Cases for math 160 ⟩;
  open: ⟨ Cases for open 162 ⟩;
  simp: ⟨ Cases for simp 167 ⟩;
  othercases do_nothing
  endcases;
  incr(pp); { if no match was found, we move to the right }
found: end;
procedure alpha_cases;
  label found;
  begin ⟨ Cases for alpha 151 ⟩;
  incr(pp); { if no match was found, we move to the right }
found: end;
```

This code is used in section 179.

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 squash(pp + 1, 1, math, 0)(4)
```

This code is used in section 150.

152. ⟨ Cases for *beginning* 152 ⟩ ≡

```
  if cat[pp + 1] = close then
    begin if (cat[pp + 2] = terminator) ∨ (cat[pp + 2] = stmt) then squash(pp, 3, stmt, −2)(5);
    end
  else if cat[pp + 1] = stmt then
      begin app1(pp); app(break_space); app1(pp + 1); reduce(pp, 2, beginning, −1)(6);
      end
```

This code is used in section 150.

153. ⟨ Cases for *case_head* 153 ⟩ ≡

```
if cat[pp + 1] = casey then
  begin if cat[pp + 2] = clause then
    begin app1(pp); app(outdent); app2(pp + 1); reduce(pp, 3, case_head, 0)(7);
    end;
  end
else if cat[pp + 1] = close then
    begin if cat[pp + 2] = terminator then
      begin app1(pp); app(cancel); app(outdent); app2(pp + 1); reduce(pp, 3, stmt, −2)(8);
      end;
    end
  else if cat[pp + 1] = stmt then
      begin app1(pp); app(force); app1(pp + 1); reduce(pp, 2, case_head, 0)(9);
      end
```

This code is used in section 149.

154. ⟨ Cases for *casey* 154 ⟩ ≡

```
if cat[pp + 1] = clause then squash(pp, 2, case_head, 0)(10)
```

This code is used in section 149.

155. ⟨ Cases for *clause* 155 ⟩ ≡

```
if cat[pp + 1] = stmt then
  begin app1(pp); app(break_space); app1(pp + 1); app(cancel); app(outdent); app(force);
  reduce(pp, 2, stmt, −2)(11);
  end
```

This code is used in section 149.

156. ⟨ Cases for *cond* 156 ⟩ ≡

```
if (cat[pp + 1] = clause) ∧ (cat[pp + 2] = stmt) then
  if cat[pp + 3] = elsie then
    begin app2(pp); app(break_space); app2(pp + 2); app("␣"); app(cancel);
    reduce(pp, 4, clause, −2)(12);
    end
  else begin app2(pp); app(break_space); app1(pp + 2); app(cancel); app(outdent); app(force);
    reduce(pp, 3, stmt, −2)(13);
    end
```

This code is used in section 149.

157. ⟨ Cases for *elsie* 157 ⟩ ≡

```
squash(pp, 1, intro, −3)(14)
```

This code is used in section 149.

158. ⟨ Cases for *exp* 158 ⟩ ≡

```
if cat[pp + 1] = math then
  begin if cat[pp + 2] = simp then
    if cat[pp + 3] ≠ simp then
      begin app3(pp); app("}"); reduce(pp, 3, math, −1)(15);
      end;
  end
else if cat[pp + 1] = simp then
    if cat[pp + 2] ≠ simp then
      begin app2(pp); app("}"); reduce(pp, 2, math, −1)(16);
      end
```

This code is used in section 149.

159. ⟨ Cases for *intro* 159 ⟩ ≡

```
if cat[pp + 1] = stmt then
  begin app1(pp); app("␣"); app(opt); app("7"); app(cancel); app1(pp + 1);
  reduce(pp, 2, stmt, −2)(17);
  end
```

This code is used in section 150.

160. ⟨ Cases for *math* 160 ⟩ ≡

```
if cat[pp + 1] = close then
  begin app("$"); app1(pp); app("$"); reduce(pp, 1, stmt, −2)(18);
  end
else if cat[pp + 1] = colon then
    begin app(force); app(backup); app("$"); app1(pp); app("$"); app1(pp + 1);
    reduce(pp, 2, intro, −3)(19);
    end
  else if cat[pp + 1] = math then squash(pp, 2, math, −1)(20)
    else if cat[pp + 1] = simp then squash(pp, 2, math, −1)(21)
      else if cat[pp + 1] = stmt then
          begin app("$"); app1(pp); app("$"); app(indent); app(break_space); app1(pp + 1);
          app(cancel); app(outdent); app(force); reduce(pp, 2, stmt, −2)(22);
          end
        else if cat[pp + 1] = terminator then
            begin app("$"); app1(pp); app("$"); app1(pp + 1); reduce(pp, 2, stmt, −2)(23);
            end
```

This code is used in section 150.

161. ⟨ Cases for *mod_scrap* 161 ⟩ ≡

```
if (cat[pp + 1] = terminator) ∨ (cat[pp + 1] = semi) then
  begin app2(pp); app(force); reduce(pp, 2, stmt, −2)(24);
  end
else squash(pp, 1, simp, −2)(25)
```

This code is used in section 149.

162. ⟨ Cases for *open* 162 ⟩ ≡

```
if (cat[pp + 1] = case_head) ∧ (cat[pp + 2] = close) then
  begin app1(pp); app("$"); app(cancel); app1(pp + 1); app(cancel); app(outdent); app("$");
  app1(pp + 2); reduce(pp, 3, math, −1)(26);
  end
else if cat[pp + 1] = close then
    begin app1(pp); app("\"); app(","); 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("}"); reduce(pp + 1, 2, math, 0)(34);
          end;
        end
      else if cat[pp + 1] = simp then squash(pp + 1, 1, math, 0)(35)
        else if (cat[pp + 1] = stmt) ∧ (cat[pp + 2] = close) then
            begin app1(pp); app("$"); app(cancel); app1(pp + 1); app(cancel); app("$");
            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("}");
                reduce(pp + 1, 2, math, 0)(37);
                end;
              end
```

This code is used in section 150.

163. ⟨ Cases for *open math* 163 ⟩ ≡

```
begin if (cat[pp + 2] = case_head) ∧ (cat[pp + 3] = close) then
  begin app2(pp); app("$"); app(cancel); app1(pp + 2); app(cancel); app(outdent); app("$");
  app1(pp + 3); reduce(pp, 4, math, −1)(28);
  end
else if cat[pp + 2] = close then squash(pp, 3, math, −1)(29)
  else if cat[pp + 2] = colon then squash(pp + 1, 2, math, 0)(30)
    else if cat[pp + 2] = proc then
        begin if cat[pp + 3] = intro then
          begin app1(pp + 1); app(math_op); app(cancel); app1(pp + 2); app("}");
          reduce(pp + 1, 3, math, 0)(31);
          end;
        end
      else if cat[pp + 2] = semi then
          begin app2(pp + 1); app("\"); app(","); app(opt); app("5");
          reduce(pp + 1, 2, math, 0)(32);
          end
        else if cat[pp + 2] = var_head then
            begin if cat[pp + 3] = intro then
              begin app1(pp + 1); app(math_op); app(cancel); app1(pp + 2); app("}");
              reduce(pp + 1, 3, math, 0)(33);
              end;
            end;
end
```

This code is used in section 162.

164. ⟨ Cases for *proc* 164 ⟩ ≡

```
if cat[pp + 1] = beginning then
  begin if (cat[pp + 2] = close) ∧ (cat[pp + 3] = terminator) then
    begin app1(pp); app(cancel); app(outdent); app3(pp + 1); reduce(pp, 4, stmt, −2)(38);
    end;
  end
else if cat[pp + 1] = stmt then
    begin app1(pp); app(break_space); app1(pp + 1); reduce(pp, 2, proc, −2)(39);
    end
```

This code is used in section 149.

165. ⟨ Cases for *record_head* 165 ⟩ ≡

```
if (cat[pp + 1] = intro) ∧ (cat[pp + 2] = casey) then
  begin app2(pp); app("␣"); app(cancel); app1(pp + 2); reduce(pp, 3, casey, −2)(40);
  end
else begin app(indent); app1(pp); app(cancel); reduce(pp, 1, case_head, 0)(41);
  end
```

This code is used in section 149.

166. ⟨ Cases for *semi* 166 ⟩ ≡

```
squash(pp, 1, terminator, −3)(42)
```

This code is used in section 149.

167. ⟨ Cases for *simp* 167 ⟩ ≡

```
if cat[pp + 1] = close then squash(pp, 1, stmt, −2)(43)
else if cat[pp + 1] = colon then
    begin app(force); app(backup); app2(pp); reduce(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.

168. ⟨ Cases for *stmt* 168 ⟩ ≡

```
if cat[pp + 1] = stmt then
  begin app1(pp); app(break_space); app1(pp + 1); reduce(pp, 2, stmt, −2)(49);
  end
```

This code is used in section 149.

169. ⟨ Cases for *terminator* 169 ⟩ ≡

```
squash(pp, 1, stmt, −2)(50)
```

This code is used in section 149.

170. ⟨ Cases for *var_head* 170 ⟩ ≡

```
if cat[pp + 1] = beginning then squash(pp, 1, 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.

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

```
define freeze_text ≡ incr(text_ptr); tok_start[text_ptr] ← tok_ptr
```

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.

```
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; trans[j] ← text_ptr; freeze_text;
  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 sections 172 and 174.

174. Similarly, the '*squash*' macro invokes a procedure called '*sq*'. This procedure takes advantage of the simplification that occurs when $k = 1$.

```
procedure sq(j : sixteen_bits; k : eight_bits; c : eight_bits; d : integer);
  var i: 0 .. max_scraps;  { index into scrap memory }
  begin if k = 1 then
    begin cat[j] ← c; ⟨ Change pp to max(scrap_base, pp+d) 173 ⟩;
    end
  else begin for i ← j to j + k − 1 do
      begin app1(i);
      end;
    red(j, k, c, d);
    end;
  end;
```

175. Here now is the code that applies productions as long as possible. It requires two local labels (*found* and *done*), as well as a local variable (i).

```
⟨ Reduce the scraps using the productions until no more rules apply 175 ⟩ ≡
  loop begin ⟨ Make sure the entries cat[pp .. (pp + 3)] are defined 176 ⟩;
    if (tok_ptr + 8 > max_toks) ∨ (text_ptr + 4 > max_texts) then
      begin stat if tok_ptr > max_tok_ptr then max_tok_ptr ← tok_ptr;
      if text_ptr > max_txt_ptr then max_txt_ptr ← text_ptr;
      tats
      overflow('token/text');
      end;
    if pp > lo_ptr then goto done;
    ⟨ Match a production at pp, or increase pp if there is no match 149 ⟩;
    end;
done:
```

This code is used in section 179.

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

```
⟨ Make sure the entries cat[pp .. (pp + 3)] are defined 176 ⟩ ≡
  if lo_ptr < pp + 3 then
    begin repeat if hi_ptr ≤ scrap_ptr then
        begin incr(lo_ptr);
        cat[lo_ptr] ← cat[hi_ptr]; trans[lo_ptr] ← trans[hi_ptr];
        incr(hi_ptr);
        end;
    until (hi_ptr > scrap_ptr) ∨ (lo_ptr = pp + 3);
    for i ← lo_ptr + 1 to pp + 3 do cat[i] ← 0;
    end
```

This code is used in section 175.

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

```
⟨ Globals in the outer block 9 ⟩ +≡
  debug tracing: 0 .. 2;  { can be used to show parsing details }
  gubed
```

178. The *prod* procedure is called in debugging mode just after *reduce* or *squash*; its parameter is the number of the production that has just been applied.

```
debug procedure prod(n : eight_bits); { shows current categories }
var k: 1 .. max_scraps; { index into cat }
begin if tracing = 2 then
  begin print_nl(n : 1, ´:´);
  for k ← scrap_base to lo_ptr do
    begin if k = pp then print(´*´) else print(´␣´);
    print_cat(cat[k]);
    end;
  if hi_ptr ≤ scrap_ptr then print(´...´); { indicate that more is coming }
  end;
end;
gubed
```

179. The *translate* function assumes that scraps have been stored in positions *scrap_base* through *scrap_ptr* of *cat* and *trans*. It appends a *terminator* scrap and begins to apply productions as much as possible. The result is a token list containing the translation of the given sequence of scraps.

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

```
⟨ Declaration of subprocedures for translate 150 ⟩
function translate: text_pointer; { converts a sequence of scraps }
  label done, found;
  var i: 1 .. max_scraps; { index into cat }
    j: 0 .. max_scraps; { runs through final scraps }
    debug k: 0 .. long_buf_size; { index into buffer }
    gubed
    begin pp ← scrap_base; lo_ptr ← pp − 1; hi_ptr ← pp;
    ⟨ If tracing, print an indication of where we are 182 ⟩;
    ⟨ Reduce the scraps using the productions until no more rules apply 175 ⟩;
    if (lo_ptr = scrap_base) ∧ (cat[lo_ptr] ≠ math) then translate ← trans[lo_ptr]
    else ⟨ Combine the irreducible scraps that remain 180 ⟩;
    end;
```

180. If the initial sequence of scraps does not reduce to a single scrap, we concatenate the translations of all remaining scraps, separated by blank spaces, with dollar signs surrounding the translations of *math* scraps.

```
⟨ Combine the irreducible scraps that remain 180 ⟩ ≡
  begin ⟨ If semi-tracing, show the irreducible scraps 181 ⟩;
  for j ← scrap_base to lo_ptr do
    begin if j ≠ scrap_base then
      begin app("␣");
      end;
    if cat[j] = math then
      begin app("$");
      end;
    app1(j);
    if cat[j] = math then
      begin app("$");
      end;
    if tok_ptr + 6 > max_toks then overflow('token');
    end;
  freeze_text; translate ← text_ptr − 1;
  end
```

This code is used in section 179.

181.

```
⟨ If semi-tracing, show the irreducible scraps 181 ⟩ ≡
  debug if (lo_ptr > scrap_base) ∧ (tracing = 1) then
    begin print_nl('Irreducible␣scrap␣sequence␣in␣section␣', module_count : 1); print_ln(':');
    mark_harmless;
    for j ← scrap_base to lo_ptr do
      begin print('␣'); print_cat(cat[j]);
      end;
    end;
  gubed
```

This code is used in section 180.

182.

```
⟨ If tracing, print an indication of where we are 182 ⟩ ≡
  debug if tracing = 2 then
    begin print_nl('Tracing␣after␣l.', line : 1, ':'); mark_harmless;
    if loc > 50 then
      begin print('...');
      for k ← loc − 50 to loc do print(xchr[buffer[k − 1]]);
      end
    else for k ← 1 to loc do print(xchr[buffer[k − 1]]);
    end
  gubed
```

This code is used in section 179.

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

Like *Pascal_xref*, the *Pascal_parse* procedure starts with the current value of *next_control* and it uses the operation *next_control* ← *get_next* repeatedly to read Pascal text until encountering the next '|' or '{', or until *next_control* ≥ *format*. The scraps corresponding to what it reads are appended into the *cat* and *trans* arrays, and *scrap_ptr* is advanced.

Like *prod*, this procedure has to split into pieces so that each part is short enough to be handled by Pascal compilers that discriminate against long subroutines. This time there are two split-off routines, called *easy_cases* and *sub_cases*.

After studying *Pascal_parse*, we will look at the sub-procedures *app_comment*, *app_octal*, and *app_hex* that are used in some of its branches.

```
⟨ Declaration of the app_comment procedure 195 ⟩
⟨ Declaration of the app_octal and app_hex procedures 196 ⟩
⟨ Declaration of the easy_cases procedure 186 ⟩
⟨ Declaration of the sub_cases procedure 192 ⟩
procedure Pascal_parse; { creates scraps from Pascal tokens }
  label reswitch, exit;
  var j: 0 .. long_buf_size; { index into buffer }
    p: name_pointer; { identifier designator }
    q, qq, r: integer; { registers for new reference insertion loop }
  begin while next_control < format do
    begin ⟨ Append the scrap appropriate to next_control 185 ⟩;
    next_control ← get_next;
    if (next_control = "|") ∨ (next_control = "{") then return;
    end;
exit: end;
```

184. The macros defined here are helpful abbreviations for the operations needed when generating the scraps. A scrap of category c whose translation has three tokens t_1, t_2, t_3 is generated by $sc3(t_1)(t_2)(t_3)(c)$, etc.

```
define s0(#) ≡ incr(scrap_ptr); cat[scrap_ptr] ← #; trans[scrap_ptr] ← text_ptr; freeze_text;
       end
define s1(#) ≡ app(#); s0
define s2(#) ≡ app(#); s1
define s3(#) ≡ app(#); s2
define s4(#) ≡ app(#); s3
define sc4 ≡ begin s4
define sc3 ≡ begin s3
define sc2 ≡ begin s2
define sc1 ≡ begin s1
define sc0(#) ≡
         begin incr(scrap_ptr); cat[scrap_ptr] ← #; trans[scrap_ptr] ← 0;
         end
define comment_scrap(#) ≡
         begin app(#); app_comment;
         end
```

185. ⟨ Append the scrap appropriate to *next_control* 185 ⟩ ≡

```
⟨ Make sure that there is room for at least four more scraps, six more tokens, and four more texts 187 ⟩;
reswitch: case next_control of
  string, verbatim: ⟨ Append a string scrap 189 ⟩;
  identifier: ⟨ Append an identifier scrap 191 ⟩;
  TeX_string: ⟨ Append a TEX string scrap 190 ⟩;
  othercases easy_cases
  endcases
```

This code is used in section 183.

186. The *easy_cases* each result in straightforward scraps.

```
⟨ Declaration of the easy_cases procedure 186 ⟩ ≡
procedure easy_cases; { a subprocedure of Pascal_parse }
  begin case next_control of
  set_element_sign: sc3("\")("i")("n")(math);
  double_dot: sc3("\")("t")("o")(math);
  "#","$","%","^","_": sc2("\")(next_control)(math);
  ignore,"|",xref_roman,xref_wildcard,xref_typewriter: do_nothing;
  "(","[": sc1(next_control)(open);
  ")","]": sc1(next_control)(close);
  "*": sc4("\")("a")("s")("t")(math);
  ",": sc3(",")(opt)("9")(math);
  ".","0","1","2","3","4","5","6","7","8","9": sc1(next_control)(simp);
  ";": sc1(";")(semi);
  ":": sc1(":")(colon);
  ⟨ Cases involving nonstandard ASCII characters 188 ⟩
  exponent: sc3("\")("E")("{")(exp);
  begin_comment: sc2("\")("B")(math);
  end_comment: sc2("\")("T")(math);
  octal: app_octal;
  hex: app_hex;
  check_sum: sc2("\")(")")(simp);
  force_line: sc2("\")("]")(simp);
  thin_space: sc2("\")(",")(math);
  math_break: sc2(opt)("0")(simp);
  line_break: comment_scrap(force);
  big_line_break: comment_scrap(big_force);
  no_line_break: begin app(big_cancel); app("\"); app("␣"); comment_scrap(big_cancel);
    end;
  pseudo_semi: sc0(semi);
  join: sc2("\")("J")(math);
  othercases sc1(next_control)(math)
  endcases;
  end;
```

This code is used in section 183.

187. ⟨ Make sure that there is room for at least four more scraps, six more tokens, and four more texts 187 ⟩ ≡

```
if (scrap_ptr + 4 > max_scraps) ∨ (tok_ptr + 6 > max_toks) ∨ (text_ptr + 4 > max_texts) then
  begin stat if scrap_ptr > max_scr_ptr then max_scr_ptr ← scrap_ptr;
  if tok_ptr > max_tok_ptr then max_tok_ptr ← tok_ptr;
  if text_ptr > max_txt_ptr then max_txt_ptr ← text_ptr;
  tats
  overflow('scrap/token/text');
  end
```

This code is used in section 185.

188. Some nonstandard ASCII characters may have entered `WEAVE` by means of standard ones. They are converted to TEX control sequences so that it is possible to keep `WEAVE` from stepping beyond standard ASCII.

⟨ Cases involving nonstandard ASCII characters 188 ⟩ ≡

```
not_equal: sc2("\")("I")(math);
less_or_equal: sc2("\")("L")(math);
greater_or_equal: sc2("\")("G")(math);
equivalence_sign: sc2("\")("S")(math);
and_sign: sc2("\")("W")(math);
or_sign: sc2("\")("V")(math);
not_sign: sc2("\")("R")(math);
left_arrow: sc2("\")("K")(math);
```

This code is used in section 186.

189. The following code must use *app_tok* instead of *app* in order to protect against overflow. Note that $tok\_ptr + 1 \le max\_toks$ after *app_tok* has been used, so another *app* is legitimate before testing again.

Many of the special characters in a string must be prefixed by '\' so that TEX will print them properly.

⟨ Append a string scrap 189 ⟩ ≡

```
begin app("\");
if next_control = verbatim then
  begin app("=");
  end
else begin app(".");
  end;
app("{"); j ← id_first;
while j < id_loc do
  begin case buffer[j] of
  "␣","\","#","%","$","^","´","`","{","}","~","&","_": begin app("\");
    end;
  "@": if buffer[j + 1] = "@" then incr(j)
    else err_print('!␣Double␣@␣should␣be␣used␣in␣strings');
  othercases do_nothing
  endcases;
  app_tok(buffer[j]); incr(j);
  end;
sc1("}")(simp);
end
```

This code is used in section 185.

190. ⟨ Append a TEX string scrap 190 ⟩ ≡

```
begin app("\"); app("h"); app("b"); app("o"); app("x"); app("{");
for j ← id_first to id_loc − 1 do app_tok(buffer[j]);
sc1("}")(simp);
end
```

This code is used in section 185.

191. ⟨ Append an identifier scrap 191 ⟩ ≡

```
begin p ← id_lookup(normal);
if ilk[p] = normal then ⟨ Insert a new reference, if this is new 277 ⟩;
case ilk[p] of
normal, array_like, const_like, div_like, do_like, for_like, goto_like, nil_like, to_like: sub_cases(p);
⟨ Cases that generate more than one scrap 193 ⟩
othercases begin next_control ← ilk[p] − char_like; goto reswitch;
  end { and, in, not, or }
endcases;
end
```

This code is used in section 185.

192. The *sub_cases* also result in straightforward scraps.

⟨ Declaration of the *sub_cases* procedure 192 ⟩ ≡

```
procedure sub_cases(p : name_pointer); { a subprocedure of Pascal_parse }
  begin case ilk[p] of
  normal: sc1(id_flag + p)(simp); { not a reserved word }
  array_like: sc1(res_flag + p)(alpha); { array, file, set }
  const_like: sc3(force)(backup)(res_flag + p)(intro); { const, label, type }
  div_like: sc3(math_bin)(res_flag + p)("}")(math); { div, mod }
  do_like: sc1(res_flag + p)(omega); { do, of, then }
  for_like: sc2(force)(res_flag + p)(alpha); { for, while, with }
  goto_like: sc1(res_flag + p)(intro); { goto, packed }
  nil_like: sc1(res_flag + p)(simp); { nil }
  to_like: sc3(math_rel)(res_flag + p)("}")(math); { downto, to }
  end;
  end;
```

This code is used in section 183.

193. ⟨ Cases that generate more than one scrap 193 ⟩ ≡
```
begin_like: begin sc3(force)(res_flag + p)(cancel)(beginning); sc0(intro);
  end; { begin }
case_like: begin sc0(casey); sc2(force)(res_flag + p)(alpha);
  end; { case }
else_like: begin ⟨ Append terminator if not already present 194 ⟩;
  sc3(force)(backup)(res_flag + p)(elsie);
  end; { else }
end_like: begin ⟨ Append terminator if not already present 194 ⟩;
  sc2(force)(res_flag + p)(close);
  end; { end }
if_like: begin sc0(cond); sc2(force)(res_flag + p)(alpha);
  end; { if }
loop_like: begin sc3(force)("\")("~")(alpha); sc1(res_flag + p)(omega);
  end; { xclause }
proc_like: begin sc4(force)(backup)(res_flag + p)(cancel)(proc); sc3(indent)("\")("␣")(intro);
  end; { function, procedure, program }
record_like: begin sc1(res_flag + p)(record_head); sc0(intro);
  end; { record }
repeat_like: begin sc4(force)(indent)(res_flag + p)(cancel)(beginning); sc0(intro);
  end; { repeat }
until_like: begin ⟨ Append terminator if not already present 194 ⟩;
  sc3(force)(backup)(res_flag + p)(close); sc0(clause);
  end; { until }
var_like: begin sc4(force)(backup)(res_flag + p)(cancel)(var_head); sc0(intro);
  end; { var }
```
This code is used in section 191.

194. If a comment or semicolon appears before the reserved words **end**, **else**, or **until**, the *semi* or *terminator* scrap that is already present overrides the *terminator* scrap belonging to this reserved word.

⟨ Append *terminator* if not already present 194 ⟩ ≡
```
if (scrap_ptr < scrap_base) ∨ ((cat[scrap_ptr] ≠ terminator) ∧ (cat[scrap_ptr] ≠ semi)) then
  sc0(terminator)
```
This code is used in sections 193, 193, and 193.

195. A comment is incorporated into the previous scrap if that scrap is of type *omega* or *semi* or *terminator*. (These three categories have consecutive category codes.) Otherwise the comment is entered as a separate scrap of type *terminator*, and it will combine with a *terminator* scrap that immediately follows it.

The *app_comment* procedure takes care of placing a comment at the end of the current scrap list. When *app_comment* is called, we assume that the current token list is the translation of the comment involved.

⟨ Declaration of the *app_comment* procedure 195 ⟩ ≡
```
procedure app_comment; { append a comment to the scrap list }
  begin freeze_text;
  if (scrap_ptr < scrap_base) ∨ (cat[scrap_ptr] < omega) ∨ (cat[scrap_ptr] > terminator) then
    sc0(terminator)
  else begin app1(scrap_ptr); { cat[scrap_ptr] is omega or semi or terminator }
    end;
  app(text_ptr − 1 + tok_flag); trans[scrap_ptr] ← text_ptr; freeze_text;
  end;
```
This code is used in section 183.

196. We are now finished with *Pascal_parse*, except for two relatively trivial subprocedures that convert constants into tokens.

```
⟨ Declaration of the app_octal and app_hex procedures 196 ⟩ ≡
procedure app_octal;
  begin app("\"); app("0"); app("{");
  while (buffer[loc] ≥ "0") ∧ (buffer[loc] ≤ "7") do
    begin app_tok(buffer[loc]); incr(loc);
    end;
  sc1("}")(simp);
  end;
procedure app_hex;
  begin app("\"); app("H"); app("{");
  while ((buffer[loc] ≥ "0") ∧ (buffer[loc] ≤ "9")) ∨ ((buffer[loc] ≥ "A") ∧ (buffer[loc] ≤ "F")) do
    begin app_tok(buffer[loc]); incr(loc);
    end;
  sc1("}")(simp);
  end;
```

This code is used in section 183.

197. When the '|' that introduces Pascal text is sensed, a call on *Pascal_translate* will return a pointer to the TEX translation of that text. If scraps exist in the *cat* and *trans* arrays, they are unaffected by this translation process.

```
  define flaky = 1
  define guaranteed = 0
function Pascal_translate: text_pointer;
  var p: text_pointer; { points to the translation }
    save_base: 0 .. max_scraps; { holds original value of scrap_base }
  begin save_base ← scrap_base; scrap_base ← scrap_ptr + 1; safety ← flaky; Pascal_parse;
      { get the scraps together }
  if next_control ≠ "|" then err_print('!␣Missing␣"|"␣after␣Pascal␣text');
  app_tok(cancel); app_comment; { place a cancel token as a final "comment" }
  p ← translate; { make the translation }
  stat if scrap_ptr > max_scr_ptr then max_scr_ptr ← scrap_ptr; tats
  scrap_ptr ← scrap_base − 1; scrap_base ← save_base; { scrap the scraps }
  safety ← guaranteed; Pascal_translate ← p;
  end;
```

198. The *outer_parse* routine is to *Pascal_parse* as *outer_xref* is to *Pascal_xref*: It constructs a sequence of scraps for Pascal text until *next_control* ≥ *format*. Thus, it takes care of embedded comments.

```
procedure outer_parse; { makes scraps from Pascal tokens and comments }
  var bal: eight_bits; { brace level in comment }
    p, q: text_pointer; { partial comments }
  begin while next_control < format do
    if next_control ≠ "{" then Pascal_parse
    else begin ⟨ Make sure that there is room for at least seven more tokens, three more texts, and one
          more scrap 199 ⟩;
      app("\"); app("C"); app("{"); bal ← copy_comment(1); next_control ← "|";
      while bal > 0 do
        begin p ← text_ptr; freeze_text; q ← Pascal_translate;
          { at this point we have tok_ptr + 6 ≤ max_toks }
        app(tok_flag + p); app(inner_tok_flag + q);
        if next_control = "|" then bal ← copy_comment(bal)
        else bal ← 0; { an error has been reported }
        end;
      app(force); app_comment; { the full comment becomes a scrap }
      end;
  end;
```

199. ⟨ Make sure that there is room for at least seven more tokens, three more texts, and one more scrap 199 ⟩ ≡

```
if (tok_ptr + 7 > max_toks) ∨ (text_ptr + 3 > max_texts) ∨ (scrap_ptr ≥ max_scraps) then
  begin stat if scrap_ptr > max_scr_ptr then max_scr_ptr ← scrap_ptr;
  if tok_ptr > max_tok_ptr then max_tok_ptr ← tok_ptr;
  if text_ptr > max_txt_ptr then max_txt_ptr ← text_ptr;
  tats
  overflow('token/text/scrap');
  end
```

This code is used in section 198.

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

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

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

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

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

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

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

```
define inner = 0  { value of mode for Pascal texts within TEX texts }
define outer = 1  { value of mode for Pascal texts in modules }
⟨ Types in the outer block 11 ⟩ +≡
  mode = inner .. outer;
  output_state = record end_field: sixteen_bits;  { ending location of token list }
    tok_field: sixteen_bits;  { present location within token list }
    mode_field: mode;  { interpretation of control tokens }
    end;
```

202.
```
define cur_end ≡ cur_state.end_field  { current ending location in tok_mem }
define cur_tok ≡ cur_state.tok_field  { location of next output token in tok_mem }
define cur_mode ≡ cur_state.mode_field  { current mode of interpretation }
define init_stack ≡ stack_ptr ← 0; cur_mode ← outer  { do this to initialize the stack }
⟨ Globals in the outer block 9 ⟩ +≡
cur_state: output_state;  { cur_end, cur_tok, cur_mode }
stack: array [1 .. stack_size] of output_state;  { info for non-current levels }
stack_ptr: 0 .. stack_size;  { first unused location in the output state stack }
  stat max_stack_ptr: 0 .. stack_size;  { largest value assumed by stack_ptr }
  tats
```

203.
```
⟨ Set initial values 10 ⟩ +≡
  stat max_stack_ptr ← 0; tats
```

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

```
procedure push_level(p : text_pointer); { suspends the current level }
  begin if stack_ptr = stack_size then overflow('stack')
  else begin if stack_ptr > 0 then stack[stack_ptr] ← cur_state; { save cur_end ... cur_mode }
    incr(stack_ptr);
    stat if stack_ptr > max_stack_ptr then max_stack_ptr ← stack_ptr; tats
    cur_tok ← tok_start[p]; cur_end ← tok_start[p + 1];
    end;
  end;
```

205. Conversely, the *pop_level* routine restores the conditions that were in force when the current level was begun. This subroutine will never be called when $stack\_ptr = 1$. It is so simple, we declare it as a macro:

```
define pop_level ≡
         begin decr(stack_ptr); cur_state ← stack[stack_ptr];
         end { do this when cur_tok reaches cur_end }
```

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

```
define res_word = '201  { returned by get_output for reserved words }
define mod_name = '200  { returned by get_output for module names }
function get_output: eight_bits; { returns the next token of output }
  label restart;
  var a: sixteen_bits; { current item read from tok_mem }
  begin restart: while cur_tok = cur_end do pop_level;
  a ← tok_mem[cur_tok]; incr(cur_tok);
  if a ≥ '400 then
    begin cur_name ← a mod id_flag;
    case a div id_flag of
    2: a ← res_word; { a = res_flag + cur_name }
    3: a ← mod_name; { a = mod_flag + cur_name }
    4: begin push_level(cur_name); goto restart;
      end; { a = tok_flag + cur_name }
    5: begin push_level(cur_name); cur_mode ← inner; goto restart;
      end; { a = inner_tok_flag + cur_name }
    othercases a ← identifier { a = id_flag + cur_name }
    endcases;
    end;
  debug if trouble_shooting then debug_help;
  gubed
  get_output ← a;
  end;
```

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

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

```
procedure make_output; forward;
procedure output_Pascal; { outputs the current token list }
  var save_tok_ptr, save_text_ptr, save_next_control: sixteen_bits; { values to be restored }
    p: text_pointer; { translation of the Pascal text }
  begin save_tok_ptr ← tok_ptr; save_text_ptr ← text_ptr; save_next_control ← next_control;
  next_control ← "|"; p ← Pascal_translate; app(p + inner_tok_flag); make_output; { output the list }
  stat if text_ptr > max_txt_ptr then max_txt_ptr ← text_ptr;
  if tok_ptr > max_tok_ptr then max_tok_ptr ← tok_ptr; tats
  text_ptr ← save_text_ptr; tok_ptr ← save_tok_ptr; { forget the tokens }
  next_control ← save_next_control; { restore next_control to original state }
  end;
```

208. Here is WEAVE's major output handler.

```
procedure make_output; { outputs the equivalents of tokens }
  label reswitch, exit, found;
  var a: eight_bits; { current output byte }
    b: eight_bits; { next output byte }
    k, k_limit: 0 .. max_bytes; { indices into byte_mem }
    w: 0 .. ww − 1; { row of byte_mem }
    j: 0 .. long_buf_size; { index into buffer }
    string_delimiter: ASCII_code; { first and last character of string being copied }
    save_loc, save_limit: 0 .. long_buf_size; { loc and limit to be restored }
    cur_mod_name: name_pointer; { name of module being output }
    save_mode: mode; { value of cur_mode before a sequence of breaks }
  begin app(end_translation); { append a sentinel }
  freeze_text; push_level(text_ptr − 1);
  loop begin a ← get_output;
  reswitch: case a of
    end_translation: return;
    identifier, res_word: ⟨ Output an identifier 209 ⟩;
    mod_name: ⟨ Output a module name 213 ⟩;
    math_bin, math_op, math_rel: ⟨ Output a \math operator 210 ⟩;
    cancel: begin repeat a ← get_output;
      until (a < backup) ∨ (a > big_force);
      goto reswitch;
      end;
    big_cancel: begin repeat a ← get_output;
      until ((a < backup) ∧ (a ≠ "␣")) ∨ (a > big_force);
      goto reswitch;
      end;
    indent, outdent, opt, backup, break_space, force, big_force: ⟨ Output a control, look ahead in case of line
        breaks, possibly goto reswitch 211 ⟩;
    othercases out(a) { otherwise a is an ASCII character }
    endcases;
    end;
exit: end;
```

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

```
⟨ Output an identifier 209 ⟩ ≡
  begin out("\");
  if a = identifier then
    if length(cur_name) = 1 then out("|")
    else out("\")
  else out("&"); { a = res_word }
  if length(cur_name) = 1 then out(byte_mem[cur_name mod ww, byte_start[cur_name]])
  else out_name(cur_name);
  end
```

This code is used in section 208.

210. ⟨ Output a `\math` operator 210 ⟩ ≡

```
begin out5("\")("m")("a")("t")("h");
if a = math_bin then out3("b")("i")("n")
else if a = math_rel then out3("r")("e")("l")
  else out2("o")("p");
out("{");
end
```

This code is used in section 208.

211. The current mode does not affect the behavior of `WEAVE`'s output routine except when we are outputting control tokens.

⟨ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 211 ⟩ ≡

```
if a < break_space then
  begin if cur_mode = outer then
    begin out2("\")(a − cancel + "0");
    if a = opt then out(get_output)   { opt is followed by a digit }
    end
  else if a = opt then b ← get_output   { ignore digit following opt }
  end
else ⟨ Look ahead for strongest line break, goto reswitch 212 ⟩
```

This code is used in section 208.

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

⟨ Look ahead for strongest line break, **goto** *reswitch* 212 ⟩ ≡

```
begin b ← a; save_mode ← cur_mode;
loop begin a ← get_output;
  if (a = cancel) ∨ (a = big_cancel) then goto reswitch;   { cancel overrides everything }
  if ((a ≠ "␣") ∧ (a < break_space)) ∨ (a > big_force) then
    begin if save_mode = outer then
      begin if out_ptr > 3 then
        if (out_buf[out_ptr] = "P") ∧ (out_buf[out_ptr − 1] = "\") ∧ (out_buf[out_ptr − 2] =
            "Y") ∧ (out_buf[out_ptr − 3] = "\") then goto reswitch;
      out2("\")(b − cancel + "0");
      if a ≠ end_translation then finish_line;
      end
    else if (a ≠ end_translation) ∧ (cur_mode = inner) then out("␣");
    goto reswitch;
    end;
  if a > b then b ← a;   { if a = "␣" we have a < b }
  end;
end
```

This code is used in section 211.

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

```
⟨ Output a module name 213 ⟩ ≡
  begin out2("\")("X"); cur_xref ← xref[cur_name];
  if num(cur_xref) ≥ def_flag then
    begin out_mod(num(cur_xref) − def_flag);
    if phase_three then
      begin cur_xref ← xlink(cur_xref);
      while num(cur_xref) ≥ def_flag do
        begin out2(",")("␣"); out_mod(num(cur_xref) − def_flag); cur_xref ← xlink(cur_xref);
        end;
      end;
    end
  else out("0"); { output the module number, or zero if it was undefined }
  out(":"); ⟨ Output the text of the module name 214 ⟩;
  out2("\")("X");
  end
```

This code is used in section 208.

214.

```
⟨ Output the text of the module name 214 ⟩ ≡
  k ← byte_start[cur_name]; w ← cur_name mod ww; k_limit ← byte_start[cur_name + ww];
  cur_mod_name ← cur_name;
  while k < k_limit do
    begin b ← byte_mem[w, k]; incr(k);
    if b = "@" then ⟨ Skip next character, give error if not '@' 215 ⟩;
    if b ≠ "|" then out(b)
    else begin ⟨ Copy the Pascal text into buffer[(limit + 1) .. j] 216 ⟩;
      save_loc ← loc; save_limit ← limit; loc ← limit + 2; limit ← j + 1; buffer[limit] ← "|";
      output_Pascal; loc ← save_loc; limit ← save_limit;
      end;
    end
```

This code is used in section 213.

215.

```
⟨ Skip next character, give error if not '@' 215 ⟩ ≡
  begin if byte_mem[w, k] ≠ "@" then
    begin print_nl('!␣Illegal␣control␣code␣in␣section␣name:␣'); print_nl('<␣');
    print_id(cur_mod_name); print('␣>␣'); mark_error;
    end;
  incr(k);
  end
```

This code is used in section 214.

216. The Pascal text enclosed in | ... | should not contain '|' characters, except within strings. We put a '|' at the front of the buffer, so that an error message that displays the whole buffer will look a little bit sensible. The variable *string_delimiter* is zero outside of strings, otherwise it equals the delimiter that began the string being copied.

```
⟨ Copy the Pascal text into buffer[(limit + 1) .. j] 216 ⟩ ≡
  j ← limit + 1; buffer[j] ← "|"; string_delimiter ← 0;
  loop begin if k ≥ k_limit then
      begin print_nl('! Pascal text in section name didn''t end:'); print_nl('<');
      print_id(cur_mod_name); print('> '); mark_error; goto found;
      end;
    b ← byte_mem[w, k]; incr(k);
    if b = "@" then ⟨ Copy a control code into the buffer 217 ⟩
    else begin if (b = """") ∨ (b = "'") then
        if string_delimiter = 0 then string_delimiter ← b
        else if string_delimiter = b then string_delimiter ← 0;
      if (b ≠ "|") ∨ (string_delimiter ≠ 0) then
        begin if j > long_buf_size − 3 then overflow('buffer');
        incr(j); buffer[j] ← b;
        end
      else goto found;
      end;
    end;
found:
```

This code is used in section 214.

217. ⟨ Copy a control code into the buffer 217 ⟩ ≡

```
  begin if j > long_buf_size − 4 then overflow('buffer');
  buffer[j + 1] ← "@"; buffer[j + 2] ← byte_mem[w, k]; j ← j + 2; incr(k);
  end
```

This code is used in section 216.

218. Phase two processing. We have assembled enough pieces of the puzzle in order to be ready to specify the processing in WEAVE's main pass over the source file. Phase two is analogous to phase one, except that more work is involved because we must actually output the TEX material instead of merely looking at the WEB specifications.

```
⟨ Phase II: Read all the text again and translate it to TEX form 218 ⟩ ≡
  reset_input; print_nl('Writing␣the␣output␣file...'); module_count ← 0; xx ← 0; copy_limbo;
  finish_line; flush_buffer(0, false, false); { insert a blank line, it looks nice }
  ⟨ Prepare high-speed access to definitions via dlink and def_val 278 ⟩;
  while ¬input_has_ended do ⟨ Translate the current module 220 ⟩
```

This code is used in section 261.

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

```
define save_position ≡ save_line ← out_line; save_place ← out_ptr
define emit_space_if_needed ≡
        if (save_line ≠ out_line) ∨ (save_place ≠ out_ptr) then out2("\")("Y")
⟨ Globals in the outer block 9 ⟩ +≡
save_line: integer; { former value of out_line }
save_place: sixteen_bits; { former value of out_ptr }
```

220.

```
⟨ Translate the current module 220 ⟩ ≡
  begin incr(module_count);
  mm ← module_count + def_flag; ref_link[0] ← 0; ref_loc[0] ← 0; new_ref_ptr ← 0; safety ← guaranteed;
  ⟨ Output the code for the beginning of a new module 221 ⟩;
  save_position;
  ⟨ Translate the TEX part of the current module 222 ⟩;
  ⟨ Translate the definition part of the current module 225 ⟩;
  ⟨ Translate the Pascal part of the current module 230 ⟩;
  ⟨ Show cross references to this module 233 ⟩;
  ⟨ Output the code for the end of a module 238 ⟩;
  end
```

This code is used in section 218.

221. Modules beginning with the WEB control sequence '@␣' start in the output with the TEX control sequence '\M', followed by the module number. Similarly, '@*' modules lead to the control sequence '\N'. If this is a changed module, we put * just before the module number.

```
⟨ Output the code for the beginning of a new module 221 ⟩ ≡
  out("\");
  if buffer[loc − 1] ≠ "*" then out("M")
  else begin out("N"); print('*', module_count : 1); update_terminal; { print a progress report }
    end;
  out_mod(module_count); out2(".")("␣")
```

This code is used in section 220.

222. In the TEX part of a module, we simply copy the source text, except that index entries are not copied and Pascal text within | ... | is translated.

```
⟨ Translate the TEX part of the current module 222 ⟩ ≡
  repeat next_control ← copy_TeX;
    case next_control of
    "|": begin init_stack; output_Pascal;
      end;
    "@": out("@");
    octal: ⟨ Translate an octal constant appearing in TEX text 223 ⟩;
    hex: ⟨ Translate a hexadecimal constant appearing in TEX text 224 ⟩;
    TeX_string, xref_roman, xref_wildcard, xref_typewriter, module_name: begin loc ← loc − 2;
      next_control ← get_next; { skip to @> }
      if next_control = TeX_string then err_print('!␣TeX␣string␣should␣be␣in␣Pascal␣text␣only');
      end;
    begin_comment, end_comment, check_sum, thin_space, math_break, line_break, big_line_break,
        no_line_break, join, pseudo_semi: err_print('!␣You␣can''t␣do␣that␣in␣TeX␣text');
    othercases do_nothing
    endcases;
  until next_control ≥ format
```

This code is used in section 220.

223.

```
⟨ Translate an octal constant appearing in TEX text 223 ⟩ ≡
  begin out3("\")("0")("{");
  while (buffer[loc] ≥ "0") ∧ (buffer[loc] ≤ "7") do
    begin out(buffer[loc]); incr(loc);
    end; { since buffer[limit] = "␣", this loop will end }
  out("}");
  end
```

This code is used in section 222.

224.

```
⟨ Translate a hexadecimal constant appearing in TEX text 224 ⟩ ≡
  begin out3("\")("H")("{");
  while ((buffer[loc] ≥ "0") ∧ (buffer[loc] ≤ "9")) ∨ ((buffer[loc] ≥ "A") ∧ (buffer[loc] ≤ "F")) do
    begin out(buffer[loc]); incr(loc);
    end;
  out("}");
  end
```

This code is used in section 222.

225. When we get to the following code we have *next_control* ≥ *format*, and the token memory is in its initial empty state.

```
⟨ Translate the definition part of the current module 225 ⟩ ≡
  if next_control ≤ definition then { definition part non-empty }
    begin emit_space_if_needed; save_position;
    end;
  while next_control ≤ definition do { format or definition }
    begin init_stack;
    if next_control = definition then ⟨ Start a macro definition 227 ⟩
    else ⟨ Start a format definition 228 ⟩;
    outer_parse; finish_Pascal;
    end
```

This code is used in section 220.

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

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

```
procedure finish_Pascal; { finishes a definition or a Pascal part }
  var p: text_pointer; { translation of the scraps }
  begin out2("\")("P"); app_tok(force); app_comment; p ← translate; app(p + tok_flag); make_output;
      { output the list }
  if out_ptr > 1 then
    if out_buf[out_ptr − 1] = "\" then
      if out_buf[out_ptr] = "6" then out_ptr ← out_ptr − 2
      else if out_buf[out_ptr] = "7" then out_buf[out_ptr] ← "Y";
  out4("\")("p")("a")("r"); finish_line;
  stat if text_ptr > max_txt_ptr then max_txt_ptr ← text_ptr;
  if tok_ptr > max_tok_ptr then max_tok_ptr ← tok_ptr;
  if scrap_ptr > max_scr_ptr then max_scr_ptr ← scrap_ptr;
  tats
  tok_ptr ← 1; text_ptr ← 1; scrap_ptr ← 0; { forget the tokens and the scraps }
  end;
```

227.

```
⟨ Start a macro definition 227 ⟩ ≡
  begin sc2("\")("D")(intro); { this will produce 'define ' }
  next_control ← get_next;
  if next_control ≠ identifier then err_print('! Improper macro definition')
  else sc1(id_flag + id_lookup(normal))(math);
  next_control ← get_next;
  end
```

This code is used in section 225.

228. ⟨ Start a format definition 228 ⟩ ≡

```
begin sc2("\")("F")(intro);  { this will produce 'format ' }
next_control ← get_next;
if next_control = identifier then
  begin sc1(id_flag + id_lookup(normal))(math); next_control ← get_next;
  if next_control = equivalence_sign then
    begin sc2("\")("S")(math);  { output an equivalence sign }
    next_control ← get_next;
    if next_control = identifier then
      begin sc1(id_flag + id_lookup(normal))(math); sc0(semi);  { insert an invisible semicolon }
      next_control ← get_next;
      end;
    end;
  end;
if scrap_ptr ≠ 5 then err_print('!␣Improper␣format␣definition');
end
```

This code is used in section 225.

229. Finally, when the TEX and definition parts have been treated, we have *next_control* ≥ *begin_Pascal*. We will make the global variable *this_module* point to the current module name, if it has a name.

⟨ Globals in the outer block 9 ⟩ +≡

```
this_module: name_pointer;  { the current module name, or zero }
```

230. ⟨ Translate the Pascal part of the current module 230 ⟩ ≡

```
this_module ← 0;
if next_control ≤ module_name then
  begin emit_space_if_needed; init_stack;
  if next_control = begin_Pascal then next_control ← get_next
  else begin this_module ← cur_module; ⟨ Check that = or ≡ follows this module name, and emit the
      scraps to start the module definition 231 ⟩;
    end;
  while next_control ≤ module_name do
    begin outer_parse; ⟨ Emit the scrap for a module name if present 232 ⟩;
    end;
  finish_Pascal;
  end
```

This code is used in section 220.

231. ⟨ Check that = or ≡ follows this module name, and emit the scraps to start the module definition 231 ⟩ ≡

```
repeat next_control ← get_next;
until next_control ≠ "+";  { allow optional '+=' }
if (next_control ≠ "=") ∧ (next_control ≠ equivalence_sign) then
  err_print('!␣You␣need␣an␣=␣sign␣after␣the␣section␣name')
else next_control ← get_next;
if out_ptr > 1 then
  if (out_buf[out_ptr] = "Y") ∧ (out_buf[out_ptr − 1] = "\") then
    begin app(backup);  { the module name will be flush left }
    end;
sc1(mod_flag + this_module)(mod_scrap); cur_xref ← xref[this_module];
if num(cur_xref) ≠ module_count + def_flag then
  begin sc3(math_rel)("+")("}")(math);  { module name is multiply defined }
  this_module ← 0;  { so we won't give cross-reference info here }
  end;
sc2("\")("S")(math);  { output an equivalence sign }
sc1(force)(semi);  { this forces a line break unless '@+' follows }
```

This code is used in section 230.

232. ⟨ Emit the scrap for a module name if present 232 ⟩ ≡

```
if next_control < module_name then
  begin err_print('!␣You␣can''t␣do␣that␣in␣Pascal␣text'); next_control ← get_next;
  end
else if next_control = module_name then
    begin sc1(mod_flag + cur_module)(mod_scrap); next_control ← get_next;
    end
```

This code is used in section 230.

233. Cross references relating to a named module are given after the module ends.

⟨ Show cross references to this module 233 ⟩ ≡

```
if this_module > 0 then
  begin ⟨ Rearrange the list pointed to by cur_xref 235 ⟩;
  footnote(def_flag); footnote(0);
  end
```

This code is used in section 220.

234. To rearrange the order of the linked list of cross references, we need four more variables that point to cross reference entries. We'll end up with a list pointed to by *cur_xref*.

⟨ Globals in the outer block 9 ⟩ +≡

```
next_xref, this_xref, first_xref, mid_xref: xref_number;  { pointer variables for rearranging a list }
```

235. We want to rearrange the cross reference list so that all the entries with *def_flag* come first, in ascending order; then come all the other entries, in ascending order. There may be no entries in either one or both of these categories.

```
⟨ Rearrange the list pointed to by cur_xref 235 ⟩ ≡
  first_xref ← xref[this_module]; this_xref ← xlink(first_xref); { bypass current module number }
  if num(this_xref) > def_flag then
    begin mid_xref ← this_xref; cur_xref ← 0; { this value doesn't matter }
    repeat next_xref ← xlink(this_xref); xlink(this_xref) ← cur_xref; cur_xref ← this_xref;
      this_xref ← next_xref;
    until num(this_xref) ≤ def_flag;
    xlink(first_xref) ← cur_xref;
    end
  else mid_xref ← 0; { first list null }
  cur_xref ← 0;
  while this_xref ≠ 0 do
    begin next_xref ← xlink(this_xref); xlink(this_xref) ← cur_xref; cur_xref ← this_xref;
    this_xref ← next_xref;
    end;
  if mid_xref > 0 then xlink(mid_xref) ← cur_xref
  else xlink(first_xref) ← cur_xref;
  cur_xref ← xlink(first_xref)
```

This code is used in section 233.

236. The *footnote* procedure gives cross reference information about multiply defined module names (if the *flag* parameter is *def_flag*), or about the uses of a module name (if the *flag* parameter is zero). It assumes that *cur_xref* points to the first cross-reference entry of interest, and it leaves *cur_xref* pointing to the first element not printed. Typical outputs: '`\A101.`'; '`\Us370\ET1009.`'; '`\As8, 27\*, 51\ETs64.`'.

```
procedure footnote(flag : sixteen_bits); { outputs module cross-references }
  label done, exit;
  var q: xref_number; { cross-reference pointer variable }
  begin if num(cur_xref) ≤ flag then return;
  finish_line; out("\");
  if flag = 0 then out("U") else out("A");
  out8("␣")("s")("e")("c")("t")("i")("o")("n");
  ⟨ Output all the module numbers on the reference list cur_xref 237 ⟩;
  out(".");
exit: end;
```

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

```
⟨ Output all the module numbers on the reference list cur_xref 237 ⟩ ≡
  q ← cur_xref;
  if num(xlink(q)) > flag then out("s"); { plural }
  out("~");
  loop begin out_mod(num(cur_xref) − flag); cur_xref ← xlink(cur_xref);
        { point to the next cross reference to output }
    if num(cur_xref) ≤ flag then goto done;
    if (num(xlink(cur_xref)) > flag) ∨ (cur_xref ≠ xlink(q)) then out(","); { not the last of two }
    out("␣");
    if num(xlink(cur_xref)) ≤ flag then out4("a")("n")("d")("~"); { the last }
    end;
done:
```

This code is used in section 236.

238.

```
⟨ Output the code for the end of a module 238 ⟩ ≡
  out6("\")("m")("i")("n")("i")("%"); beta_out; flush_buffer(out_ptr, false, false);
  out4("}")("\")("F")("I"); finish_line; flush_buffer(0, false, false); { insert a blank line, it looks nice }
```

This code is used in section 220.

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.

240. Just before the index comes a list of all the changed modules, including the index module itself.

```
⟨ Globals in the outer block 9 ⟩ +≡
k_module: 0 .. max_modules; { runs through the modules }
```

241.

```
⟨ Tell about changed modules 241 ⟩ ≡
  begin { remember that the index is already marked as changed }
  k_module ← 1;
  while ¬changed_module[k_module] do incr(k_module);
  out4("\")("c")("h")("␣"); out_mod(k_module);
  repeat repeat incr(k_module) until changed_module[k_module];
    out2(",")("␣"); out_mod(k_module);
  until k_module = module_count;
  out(".");
  end
```

This code is used in section 239.

242. A left-to-right radix sorting method is used, since this makes it easy to adjust the collating sequence and since the running time will be at worst proportional to the total length of all entries in the index. We put the identifiers into 230 different lists based on their first characters. (Uppercase letters are put into the same list as the corresponding lowercase letters, since we want to have 't < *TeX* < **to**'.) The list for character c begins at location *bucket*[c] and continues through the *blink* array.

```
⟨ Globals in the outer block 9 ⟩ +≡
bucket: array [ASCII_code] of name_pointer;
next_name: name_pointer; { successor of cur_name when sorting }
c: ASCII_code; { index into bucket }
h: 0 .. hash_size; { index into hash }
blink: array [0 .. max_names] of sixteen_bits; { links in the buckets }
```

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

```
⟨ Do the first pass of sorting 243 ⟩ ≡
  for c ← 0 to 255 do bucket[c] ← 0;
  for h ← 0 to hash_size − 1 do
    begin next_name ← hash[h];
    while next_name ≠ 0 do
      begin cur_name ← next_name; next_name ← link[cur_name];
      if xref[cur_name] ≠ 0 then
        begin c ← byte_mem[cur_name mod ww, byte_start[cur_name]];
        if (c ≤ "Z") ∧ (c ≥ "A") then c ← c + ´40;
        blink[cur_name] ← bucket[c]; bucket[c] ← cur_name;
        end;
      end;
    end
```

This code is used in section 239.

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

```
  define depth ≡ cat { reclaims memory that is no longer needed for parsing }
  define head ≡ trans { ditto }
  define sort_ptr ≡ scrap_ptr { ditto }
  define max_sorts ≡ max_scraps { ditto }

⟨ Globals in the outer block 9 ⟩ +≡
cur_depth: eight_bits; { depth of current buckets }
cur_byte: 0 .. max_bytes; { index into byte_mem }
cur_bank: 0 .. ww − 1; { row of byte_mem }
cur_val: sixteen_bits; { current cross reference number }
  stat max_sort_ptr: 0 .. max_sorts; tats { largest value of sort_ptr }
```

245.
```
⟨ Set initial values 10 ⟩ +≡
  stat max_sort_ptr ← 0; tats
```

246. The desired alphabetic order is specified by the *collate* array; namely, *collate*[0] < *collate*[1] < ··· < *collate*[229].

```
⟨ Globals in the outer block 9 ⟩ +≡
collate: array [0 .. 229] of ASCII_code; { collation order }
```

247.
```
⟨ Local variables for initialization 16 ⟩ +≡
c: ASCII_code; { used to initialize collate }
```

248. We use the order null $<$ ␣ $<$ other characters $<$ `_` $<$ `A` $=$ `a` $<\cdots<$ `Z` $=$ `z` $<$ `0` $<\cdots<$ `9`.

```
⟨ Set initial values 10 ⟩ +≡
  collate[0] ← 0; collate[1] ← "␣";
  for c ← 1 to "␣" − 1 do collate[c + 1] ← c;
  for c ← "␣" + 1 to "0" − 1 do collate[c] ← c;
  for c ← "9" + 1 to "A" − 1 do collate[c − 10] ← c;
  for c ← "Z" + 1 to "_" − 1 do collate[c − 36] ← c;
  collate["_" − 36] ← "_" + 1;
  for c ← "z" + 1 to 255 do collate[c − 63] ← c;
  collate[193] ← "_";
  for c ← "a" to "z" do collate[c − "a" + 194] ← c;
  for c ← "0" to "9" do collate[c − "0" + 220] ← c;
```

249. Procedure *unbucket* goes through the buckets and adds nonempty lists to the stack, using the collating sequence specified in the *collate* array. The parameter to *unbucket* tells the current depth in the buckets. Any two sequences that agree in their first 255 character positions are regarded as identical.

```
  define infinity = 255 { ∞ (approximately) }
procedure unbucket(d : eight_bits); { empties buckets having depth d }
  var c: ASCII_code; { index into bucket }
  begin for c ← 229 downto 0 do
    if bucket[collate[c]] > 0 then
      begin if sort_ptr > max_sorts then overflow('sorting');
      incr(sort_ptr);
      stat if sort_ptr > max_sort_ptr then max_sort_ptr ← sort_ptr; tats
      if c = 0 then depth[sort_ptr] ← infinity
      else depth[sort_ptr] ← d;
      head[sort_ptr] ← bucket[collate[c]]; bucket[collate[c]] ← 0;
      end;
  end;
```

250.

```
⟨ Sort and output the index 250 ⟩ ≡
  sort_ptr ← 0; unbucket(1);
  while sort_ptr > 0 do
    begin cur_depth ← cat[sort_ptr];
    if (blink[head[sort_ptr]] = 0) ∨ (cur_depth = infinity) then
      ⟨ Output index entries for the list at sort_ptr 252 ⟩
    else ⟨ Split the list at sort_ptr into further lists 251 ⟩;
    end
```

This code is used in section 239.

251. ⟨ Split the list at *sort_ptr* into further lists 251 ⟩ ≡

```
begin next_name ← head[sort_ptr];
repeat cur_name ← next_name; next_name ← blink[cur_name];
  cur_byte ← byte_start[cur_name] + cur_depth; cur_bank ← cur_name mod ww;
  if cur_byte = byte_start[cur_name + ww] then c ← 0   { we hit the end of the name }
  else begin c ← byte_mem[cur_bank, cur_byte];
    if (c ≤ "Z") ∧ (c ≥ "A") then c ← c + ´40;
    end;
  blink[cur_name] ← bucket[c]; bucket[c] ← cur_name;
until next_name = 0;
decr(sort_ptr); unbucket(cur_depth + 1);
end
```

This code is used in section 250.

252. ⟨ Output index entries for the list at *sort_ptr* 252 ⟩ ≡

```
begin cur_name ← head[sort_ptr];
debug if trouble_shooting then debug_help; gubed
repeat out2("\")(":"); ⟨ Output the name at cur_name 253 ⟩;
  ⟨ Output the cross-references at cur_name 254 ⟩;
  cur_name ← blink[cur_name];
until cur_name = 0;
decr(sort_ptr);
end
```

This code is used in section 250.

253. ⟨ Output the name at *cur_name* 253 ⟩ ≡

```
case ilk[cur_name] of
normal: if length(cur_name) = 1 then out2("\")("|") else out2("\")("\");
roman: do_nothing;
wildcard: out2("\")("9");
typewriter: out2("\")(".");
othercases out2("\")("&")
endcases;
out_name(cur_name)
```

This code is used in section 252.

254. Section numbers that are to be underlined are enclosed in '\[...]'.

```
⟨ Output the cross-references at cur_name 254 ⟩ ≡
  ⟨ Invert the cross-reference list at cur_name, making cur_xref the head 255 ⟩;
  repeat out2(",")("␣"); cur_val ← num(cur_xref);
    if cur_val < def_flag then
      begin out_mod(cur_val);
      if cur_val + 1 = num(xlink(cur_xref)) then
        begin out2("-")("-");
        repeat cur_xref ← xlink(cur_xref); incr(cur_val);
        until cur_val + 1 ≠ num(xlink(cur_xref));
        out_mod(cur_val);
        end;
      end
    else begin out2("\")("["); out_mod(cur_val − def_flag); out("]");
      end;
    cur_xref ← xlink(cur_xref);
  until cur_xref = 0;
  out("."); finish_line
```

This code is used in section 252.

255. List inversion is best thought of as popping elements off one stack and pushing them onto another. In this case *cur_xref* will be the head of the stack that we push things onto.

```
⟨ Invert the cross-reference list at cur_name, making cur_xref the head 255 ⟩ ≡
  this_xref ← xref[cur_name]; cur_xref ← 0;
  repeat next_xref ← xlink(this_xref); xlink(this_xref) ← cur_xref; cur_xref ← this_xref;
    this_xref ← next_xref;
  until this_xref = 0
```

This code is used in section 254.

256. The following recursive procedure walks through the tree of module names and prints them.

```
procedure mod_print(p : name_pointer); { print all module names in subtree p }
  begin if p > 0 then
    begin mod_print(llink[p]);
    out2("\")(":");
    tok_ptr ← 1; text_ptr ← 1; scrap_ptr ← 0; init_stack; app(p + mod_flag); make_output; footnote(0);
        { cur_xref was set by make_output }
    finish_line;
    mod_print(rlink[p]);
    end;
  end;
```

257. ⟨ Output all the module names 257 ⟩ ≡ *mod_print*(*root*)

This code is used in section 239.

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 ≡ 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 ← true; debug_cycle ← 1; debug_skipped ← 0; tracing ← 0;
trouble_shooting ← false; debug_cycle ← 99999; { use these when it almost works }
gubed
```

260. **define** *breakpoint* = 888 { place where a breakpoint is desirable }

```
  debug procedure debug_help; { routine to display various things }
  label breakpoint, exit;
  var k: integer; { index into various arrays }
  begin incr(debug_skipped);
  if debug_skipped < debug_cycle then return;
  debug_skipped ← 0;
  loop begin print_nl('#'); update_terminal; { prompt }
    read(term_in, ddt); { read a debug-command code }
    if ddt < 0 then return
    else if ddt = 0 then
        begin goto breakpoint; @\ { go to every label at least once }
      breakpoint: ddt ← 0; @\
        end
      else begin read(term_in, dd);
        case ddt of
        1: print_id(dd);
        2: print_text(dd);
        3: for k ← 1 to dd do print(xchr[buffer[k]]);
        4: for k ← 1 to dd do print(xchr[mod_text[k]]);
        5: for k ← 1 to out_ptr do print(xchr[out_buf[k]]);
        6: for k ← 1 to dd do
            begin print_cat(cat[k]); print('␣');
            end;
        othercases print('?')
        endcases;
        end;
    end;
exit: end;
  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.

```
⟨ Functions alpha_out, beta_out, and helpers 280 ⟩
procedure Phase_I;
  begin ⟨ Phase I: Read all the user's text and store the cross references 109 ⟩;
  end;
procedure Phase_II;
  var lhs: integer;
  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.
```

262. ⟨ Print statistics about memory usage 262 ⟩ ≡

```
print_nl('Memory␣usage␣statistics:␣', name_ptr : 1, '␣names,␣', xref_ptr : 1,
    '␣cross␣references,␣', byte_ptr[0] : 1);
for cur_bank ← 1 to ww − 1 do print('+', byte_ptr[cur_bank] : 1);
print('␣bytes;'); print_nl('parsing␣required␣', max_scr_ptr : 1, '␣scraps,␣', max_txt_ptr : 1,
    '␣texts,␣', max_tok_ptr : 1, '␣tokens,␣', max_stack_ptr : 1, '␣levels;');
print_nl('sorting␣required␣', max_sort_ptr : 1, '␣levels.')
```

This code is used in section 33.

263. 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 we simply report the history to the user.

```
⟨ Print the job history 263 ⟩ ≡
  case history of
  spotless: print_nl('(No␣errors␣were␣found.)');
  harmless_message: print_nl('(Did␣you␣see␣the␣warning␣message␣above?)');
  error_message: print_nl('(Pardon␣me,␣but␣I␣think␣I␣spotted␣something␣wrong.)');
  fatal_message: print_nl('(That␣was␣a␣fatal␣error,␣my␣friend.)');
  end { there are no other cases }
```

This code is used in section 33.

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(TWILL_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, and 269.

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. 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*);

268. ⟨ Globals in the outer block 9 ⟩ +≡
no_xref: *c_int_type*;

269. 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;

270. Global filenames.

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

271. New material for TWILL. Here's a new subroutine needed for TWILL. Assuming that *next_control* is the beginning of a numeric constant, and that string constants have length 1, the *scan_const* function returns the value of the constant and sets *next_control* to the following token.

```
⟨ Functions scan_const and scan_exp 271 ⟩ ≡
function scan_const: integer;
  label done;
  var radix, accum, p: integer;
  begin if next_control = string then
    begin accum ← buffer[id_first + 1]; next_control ← get_next; goto done;
    end
  else if next_control = identifier then
      begin p ← id_lookup(normal); new_blank_xref(p); accum ← def_val[p]; next_control ← get_next;
      goto done;
      end
    else begin accum ← 0;
      if next_control = hex then radix ← 16
      else if next_control = octal then radix ← 8
        else begin radix ← 10; accum ← next_control − "0";
          end;
      loop
        begin next_control ← get_next;
        if next_control < "0" then goto done;
        if radix = 16 then
          begin if (next_control ≥ "A") ∧ (next_control ≤ "F") then
            next_control ← next_control − "A" + "0" + 10
          else if next_control > "9" then goto done;
          end
        else if next_control ≥ "0" + radix then goto done;
        accum ← accum ∗ radix + next_control − "0";
        end;
      end;
done: scan_const ← accum;
  end;
```

See also section 272.

This code is used in section 111.

272. Simple linear arithmetic is handled by the following subroutine, which doesn't complain about certain syntactic errors.

```
define start_of_const(#) ≡ (((# ≥ "0")∧(# ≤ "9"))∨(# = hex)∨(# = octal)∨(# = string)∨(# = identifier))
define sign(#) ≡ (abs(# − ",") = 1)
define start_of_num(#) ≡ (start_of_const(#) ∨ sign(#))
⟨ Functions scan_const and scan_exp 271 ⟩ +≡
function scan_exp: integer;
  label done;
  var accum, s: integer;
  begin if sign(next_control) then accum ← 0
  else accum ← scan_const;
  loop
    begin if ¬sign(next_control) then goto done;
    s ← "," − next_control; next_control ← get_next;
    if ¬start_of_const(next_control) then goto done;
    accum ← accum + s ∗ scan_const;
    end;
done: scan_exp ← accum;
  end;
```

273.

```
define found_it(#) ≡
        begin def_type ← #; goto found;
        end
⟨ Figure out the def_type and def_name, etc. 273 ⟩ ≡
  begin next_control ← get_next;
  if next_control = "," then found_it(dtype_comma);
  if (next_control = ":") ∨ (next_control = "=") then
    ⟨ Figure out a type and goto either found or not_found 274 ⟩;
not_found: def_type ← dtype_none; new_xref(p); goto done;
found: end
```

This code is used in section 111.

274. ⟨ Figure out a type and **goto** either *found* or *not_found* 274 ⟩ ≡

```
begin eq ← (next_control − ":") div ("=" − ":"); next_control ← get_next;
if next_control = identifier then
  ⟨ Figure out a type starting with an identifier; goto found or not_found unless it's a subrange 275 ⟩
else if (next_control = string) ∧ (id_loc − id_first > 2) then found_it(dtype_string)
  else if start_of_num(next_control) then
      ⟨ Figure out a type starting with a constant; goto found or not_found unless it's a subrange 276 ⟩
    else goto not_found;
next_control ← get_next;
if next_control = identifier then
  begin def_subname ← id_lookup(normal);
  if ilk[def_subname] ≠ normal then goto not_found;
  if def_val[def_subname] = undef_val then
    begin new_blank_xref(def_subname); def_subtype ← dtype_colon_ital_dots; goto found;
    end;
  end;
if start_of_num(next_control) then
  begin def_subname ← scan_exp; def_subtype ← dtype_colon_const_dots;
  if abs(def_subname) ≥ 32768 then goto not_found;
  if def_subname < 0 then def_subname ← def_subname + 65536;
  goto found;
  end;
goto not_found;
end
```

This code is used in section 273.

275. ⟨ Figure out a type starting with an identifier; **goto** *found* or *not_found* unless it's a subrange 275 ⟩ ≡

```
begin def_name ← id_lookup(normal);
if ilk[def_name] = goto_like then { packed }
  begin next_control ← get_next; eq ← eq + 2;
  if next_control ≠ identifier then goto not_found;
  def_name ← id_lookup(normal);
  end;
if (ilk[def_name] = array_like) ∨ (ilk[def_name] = record_like) then found_it(dtype_colon_bold + eq);
if ilk[def_name] ≠ normal then goto found;
new_blank_xref(def_name);
if def_val[def_name] = undef_val then
  begin next_control ← get_next;
  if next_control = double_dot then def_type ← dtype_colon_ital_dots + eq
  else found_it(dtype_colon_ital + eq);
  end
else ⟨ Figure out a type starting with a constant; goto found or not_found unless it's a subrange 276 ⟩;
end
```

This code is used in section 274.

276. ⟨ Figure out a type starting with a constant; **goto** *found* or *not_found* unless it's a subrange 276 ⟩ ≡

```
begin def_name ← scan_exp;
if next_control ≠ double_dot then
  if eq = 1 then found_it(dtype_equal_bold; def_name ← const_name)
  else goto not_found;
if abs(def_name) ≥ 32768 then goto not_found;
if def_name < 0 then def_name ← def_name + 65536;
def_type ← dtype_colon_const_dots + eq;
end
```

This code is used in sections 274 and 275.

277. Some identifiers can be inserted during Phase II that weren't seen in Phase I (namely, if they appear only in module names); so we have to watch out that *def_val* might be *undef_val*.

⟨ Insert a new reference, if this is new 277 ⟩ ≡

```
begin q ← def_val[p];
if q ≠ 0 then
  if q ≠ undef_val then
    if ¬phase_three then
      begin repeat r ← q; q ← dlink(q);
      until (q = 0) ∨ (num(q) > mm);
      if num(r) ≠ mm then
        begin qq ← 0; q ← ref_link[0];
        while ref_loc[q] > r do
          begin qq ← q; q ← ref_link[qq];
          end;
        if ref_loc[q] = r then
          begin if safety = guaranteed then
            if ref_safety[q] = flaky then ref_safety[q] ← guaranteed;
          end
        else begin if new_ref_ptr = max_new_refs then overflow('new␣references');
          incr(new_ref_ptr); ref_link[new_ref_ptr] ← q; ref_link[qq] ← new_ref_ptr;
          ref_loc[new_ref_ptr] ← r;
          if dlink(def_val[p]) = 0 then ref_safety[new_ref_ptr] ← guaranteed
          else ref_safety[new_ref_ptr] ← safety;
          end;
        end;
      end;
end
```

This code is used in section 191.

278. ⟨ Prepare high-speed access to definitions via *dlink* and *def_val* 278 ⟩ ≡

```
for lhs ← 1 to name_ptr do
  if ilk[lhs] = normal then
    begin def_val[lhs] ← 0; rhs ← xref[lhs];
    while rhs ≠ 0 do
      begin if num(rhs) > def_flag then
        begin dlink(rhs) ← def_val[lhs]; def_val[lhs] ← rhs;
        end;
      rhs ← xlink(rhs);
      end;
    end
```

This code is used in section 218.

279. We keep a separate list of all references made in the current module, sorted by *xref* number. A reference is considered to need manual checking if it appears only in a comment within the section and if the corresponding identifier is multiply defined.

```
⟨ Globals in the outer block 9 ⟩ +≡
mm: integer;  { current module number plus def_flag }
ref_loc: array [0 .. max_new_refs] of sixteen_bits;
ref_link: array [0 .. max_new_refs] of sixteen_bits;
ref_safety: array [0 .. max_new_refs] of guaranteed .. flaky;
new_ref_ptr: 0 .. max_new_refs;
safety: guaranteed .. flaky;
xx: xref_number;
```

280. The *alpha_out* procedure makes entries for all identifiers defined in the current module. (However, I no longer need these!)

```
⟨ Functions alpha_out, beta_out, and helpers 280 ⟩ ≡
procedure alpha_out;
  label exit;
  var p, w, k: integer;
  begin loop
    begin if xx = xref_ptr then return;
    if num(xx + 1) > mm then return;
    incr(xx);
    if num(xx) > def_flag then
      if dback(xx) > 0 then
        if ilk[dback(xx)] = normal then
          begin flush_buffer(out_ptr, false, false); out5("\")("m")("i")("n")("i"); p ← dback(xx);
          w ← p mod ww;
          for k ← byte_start[p] to byte_start[p + ww] − 1 do out(byte_mem[w, k]);
          if dtype(xx) > dtype_comma then incr(xx);
          end;
    end;
exit: end;
```

See also sections 281, 282, and 283.

This code is used in section 261.

281. Here's a procedure that's very much like *out_mod*.

```
⟨ Functions alpha_out, beta_out, and helpers 280 ⟩ +≡
procedure out_const(n : sixteen_bits);
  var a, k: integer;
  begin a ← n; k ← 0;
  if a ≥ 32768 then
    begin out("-"); a ← 65536 − a;
    end;
  repeat dig[k] ← a mod 10; a ← a div 10; incr(k);
  until a = 0;
  repeat decr(k); out(dig[k] + "0");
  until k = 0;
  end;
```

282. And here's something that could have been made a subroutine earlier.

```
⟨ Functions alpha_out, beta_out, and helpers 280 ⟩ +≡
procedure out_id(p : integer);
  begin out("\");
  if ilk[p] = normal then
    if length(p) = 1 then out("|")
    else out("\")
  else out("&");
  if length(p) = 1 then out(byte_mem[p mod ww, byte_start[p]])
  else out_name(p);
  end;
```

283. The *beta_out* procedure makes entries for all identifiers used but not defined in the current module.

```
⟨ Functions alpha_out, beta_out, and helpers 280 ⟩ +≡
procedure beta_out;
  label done, 888, found;
  var k, p, q, w, xx, mmm: integer;
  begin p ← ref_link[0];
  while p ≠ 0 do
    begin flush_buffer(out_ptr, false, false); out2("\")("["); xx ← ref_loc[p]; q ← dback(xx);
    w ← q mod ww;
    for k ← byte_start[q] to byte_start[q + ww] − 1 do out(byte_mem[w, k]);
    out("␣"); out_const(num(xx) − def_flag); out("␣"); ⟨ Move past commas 285 ⟩;
    ⟨ Output the reference, based on its dtype 284 ⟩;
    if ref_safety[p] = flaky then out3("␣")("%")("?");
    p ← ref_link[p];
    end;
  end;
```

284. ⟨ Output the reference, based on its *dtype* 284 ⟩ ≡

```
case dtype(xx) of
dtype_none: out5("\")("n")("o")("n")("e");
dtype_macro: out6("=")("m")("a")("c")("r")("o");
dtype_const: begin out2("=")("$"); out_const(dname(xx)); out("$");
  end;
dtype_string: out12("\")(".")("{")(""")("s")("t")("r")("i")("n")("g")(""")("}");
dtype_colon_bold, dtype_colon_ital: begin out(":"); out_id(dname(xx));
  end;
dtype_equal_bold, dtype_equal_ital: begin out("="); out_id(dname(xx));
  end;
dtype_colon_packed: begin out(":"); out_id(packed_name); out2("\")("␣"); out_id(dname(xx));
  end;
dtype_equal_packed: begin out("="); out_id(packed_name); out2("\")("␣"); out_id(dname(xx));
  end;
dtype_colon_const_dots: begin out2(":")("$"); out_const(dname(xx)); goto 888;
  end;
dtype_equal_const_dots: begin out2("=")("$"); out_const(dname(xx)); goto 888;
  end;
dtype_colon_ital_dots: begin out2(":")("$"); out_id(dname(xx)); goto 888;
  end;
dtype_equal_ital_dots: begin out2("=")("$"); out_id(dname(xx)); goto 888;
  end;
end; { there are no other cases }
goto found;
888: out3("\")("t")("o");
if dtype(xx + 1) = dtype_colon_ital_dots then out_id(dname(xx + 1))
else out_const(dname(xx + 1));
out("$");
found:
```

This code is used in section 283.

285. ⟨ Move past commas 285 ⟩ ≡

```
mmm ← num(xx);
loop
  begin if dtype(xx) ≠ dtype_comma then
    if num(xx) = mmm then goto done;
  if xx = xref_ptr then
    begin dtype(xx) ← dtype_none; goto done;
    end;
  if num(xx + 1) > mmm then
    begin dtype(xx) ← dtype_none; goto done;
    end;
  incr(xx);
  end;
done:
```

This code is used in section 283.

286. 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.

⟨ Do special things when $c =$ "@", "\", "{", "}"; **goto** *done* at end 92 ⟩ Used in section 91.
⟨ Do the first pass of sorting 243 ⟩ Used in section 239.
⟨ Emit the scrap for a module name if present 232 ⟩ Used in section 230.
⟨ Enter a new module name into the tree 67 ⟩ Used in section 66.
⟨ Enter a new name into the table at position p 62 ⟩ Used in section 58.
⟨ Error handling procedures 30, 31, 33 ⟩ Used in section 2.
⟨ Figure out a type and **goto** either *found* or *not_found* 274 ⟩ Used in section 273.
⟨ Figure out a type starting with a constant; **goto** *found* or *not_found* unless it's a subrange 276 ⟩ Used in sections 274 and 275.
⟨ Figure out a type starting with an identifier; **goto** *found* or *not_found* unless it's a subrange 275 ⟩ Used in section 274.
⟨ Figure out the *def_type* and *def_name*, etc. 273 ⟩ Used in section 111.
⟨ Functions *alpha_out*, *beta_out*, and helpers 280, 281, 282, 283 ⟩ Used in section 261.
⟨ Functions *scan_const* and *scan_exp* 271, 272 ⟩ Used in section 111.
⟨ Get a string 99 ⟩ Used in section 95.
⟨ Get an identifier 98 ⟩ Used in section 95.
⟨ Get control code and possible module name 100 ⟩ Used in section 95.
⟨ Globals in the outer block 9, 13, 23, 25, 27, 29, 37, 39, 45, 48, 53, 55, 63, 65, 71, 73, 93, 108, 114, 118, 121, 129, 144, 177, 202, 219, 229, 234, 240, 242, 244, 246, 258, 268, 270, 279 ⟩ Used in section 2.
⟨ Go to *found* if c is a hexadecimal digit, otherwise set *scanning_hex* ← *false* 96 ⟩ Used in section 95.
⟨ If end of name, **goto** *done* 104 ⟩ Used in section 103.
⟨ If semi-tracing, show the irreducible scraps 181 ⟩ Used in section 180.
⟨ If the current line starts with @y, report any discrepancies and **return** 80 ⟩ Used in section 79.
⟨ If tracing, print an indication of where we are 182 ⟩ Used in section 179.
⟨ Insert a new reference, if this is new 277 ⟩ Used in section 191.
⟨ Invert the cross-reference list at *cur_name*, making *cur_xref* the head 255 ⟩ Used in section 254.
⟨ Local variables for initialization 16, 40, 56, 247 ⟩ Used in section 2.
⟨ Look ahead for strongest line break, **goto** *reswitch* 212 ⟩ Used in section 211.
⟨ Make sure that there is room for at least four more scraps, six more tokens, and four more texts 187 ⟩ Used in section 185.
⟨ Make sure that there is room for at least seven more tokens, three more texts, and one more scrap 199 ⟩ Used in section 198.
⟨ Make sure the entries $cat[pp \mathinner{..} (pp + 3)]$ are defined 176 ⟩ Used in section 175.
⟨ Match a production at pp, or increase pp if there is no match 149 ⟩ Used in section 175.
⟨ Move past commas 285 ⟩ Used in section 283.
⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 78 ⟩ Used in sections 75 and 79.
⟨ Output a control, look ahead in case of line breaks, possibly **goto** *reswitch* 211 ⟩ Used in section 208.
⟨ Output a \math operator 210 ⟩ Used in section 208.
⟨ Output a module name 213 ⟩ Used in section 208.
⟨ Output all the module names 257 ⟩ Used in section 239.
⟨ Output all the module numbers on the reference list *cur_xref* 237 ⟩ Used in section 236.
⟨ Output an identifier 209 ⟩ Used in section 208.
⟨ Output index entries for the list at *sort_ptr* 252 ⟩ Used in section 250.
⟨ Output the code for the beginning of a new module 221 ⟩ Used in section 220.
⟨ Output the code for the end of a module 238 ⟩ Used in section 220.
⟨ Output the cross-references at *cur_name* 254 ⟩ Used in section 252.
⟨ Output the name at *cur_name* 253 ⟩ Used in section 252.
⟨ Output the reference, based on its *dtype* 284 ⟩ Used in section 283.
⟨ Output the text of the module name 214 ⟩ Used in section 213.
⟨ 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.

⟨ Prepare high-speed access to definitions via *dlink* and *def_val* 278 ⟩ Used in section 218.
⟨ 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 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.