September 1983

Report No. STAN-CS-83-980

The WEB System of Structured Documentation

by

Donald E. Knuth

Department of Computer Science

Stanford University
Stanford, CA 94305

The WEB System of Structured Documentation

by Donald E. Knuth
Stanford University

(Version 2.3, September, 1983)

The preparation of this report was supported in part by the National Science Foundation under grants IST-8201926 and MCS-8300084, and by the System Development Foundation.
'TEX' is a trademark of the American Mathematical Society.

The WEB System of Structured Documentation

This memo describes how to write programs in the WEB language; and it also includes the full WEB documentation for WEAVE and TANGLE, the programs that read WEB input and produce TEX and PASCAL output, respectively. The philosophy behind WEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TEX for formatting, and a language like PASCAL for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.

The structure of a software program may be thought of as a "web" that is made up of many interconnected pieces. To document such a program, we want to explain each individual part of the web and how it relates to its neighbors. The typographic tools provided by TEX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by PASCAL make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

Since WEB is an experimental system developed for internal use within the TEX project at Stanford, this report is rather terse, and it assumes that the reader is an experienced programmer who is highly motivated to read a detailed description of WEB's rules. Furthermore, even if a less terse manual were to be written, the reader would have to be warned in advance that WEB is not for beginners and it never will be: The user of WEB must be familiar with both TEX and PASCAL. When one writes a WEB description of a software system, it is possible to make mistakes by breaking the rules of WEB and/or the rules of TEX and/or the rules of PASCAL. In practice, all three types of errors will occur, and you will get different error messages from the different language processors. In compensation for the sophisticated expertise needed to cope with such a variety of languages, however, experience has shown that reliable software can be created quite rapidly by working entirely in WEB from the beginning; and the documentation of such programs seems to be better than the documentation obtained by any other known method. Thus, WEB users need to be highly qualified, but they can get some satisfaction and perhaps even a special feeling of accomplishment when they have successfully created a software system with this method.

To use WEB, you prepare a file called COB.WEB (say), and then you apply a system program called WEAVE to this file, obtaining an output file called COB.TEX. When TEX processes COB.TEX, your output will be a "pretty printed" version of COB.WEB that takes appropriate care of typographic details like page layout and the use of indentation, italics, boldface, etc.; this output will contain extensive cross-index information that is gathered automatically. You can also submit the same file COB.WEB to another system program called TANGLE, which will produce a file COB.PAS that contains the PASCAL code of your COB program. The PASCAL compiler will convert COB.PAS into machine-language instructions corresponding to the algorithms that were so nicely formatted by WEAVE and TEX. Finally, you can (and should) delete the files COB.TEX and COB.PAS, because COB.WEB contains the definitive source code. Examples of the behavior of WEAVE and TANGLE are appended to this manual.

Besides providing a documentation tool, WEB enhances the PASCAL language by providing a rudimentary macro capability together with the ability to permute pieces of the program text, so that a large system can be understood entirely in terms of small modules and their local interrelationships. The TANGLE program is so named because it takes a given web and moves the modules from their web structure into the order required by PASCAL; the advantage of programming in WEB is that the algorithms can be expressed in "untangled" form, with each module explained separately. The WEAVE program is so named because it takes a given web and intertwines the TEX and PASCAL portions contained in each module, then it knits the whole fabric into a structured document. (Get it? Wow.) Perhaps there is some deep connection here with the fact that the German word for "weave" is "web"? and the corresponding Latin imperative is "*texe*"!

It is impossible to list all of the related work that has influenced the design of WEB, but the key contributions should be mentioned here. (1) Myrtle Kellington, as executive editor for ACM publications, developed excellent typographic standards for the typesetting of Algol programs during the 1960s, based on the original designs of Peter Naur; the subtlety and quality of this influential work can be appreciated only by people who have seen what happens when other printers try to typeset Algol without the advice of ACM's copy editors. (2) Bill McKeeman introduced a program intended to automate some of this task [Algorithm 268, "Algol 60 reference language editor," *CACM* 8 (1965), 667–668]; and a considerable flowering of such programs has occurred in recent years [see especially Derek Oppen, "Prettyprinting," ACM *TOPLAS* 2 (1980), 465-483; G. A. Rose and J. Welsh, "Formatted programming languages," SOFTWARE Practice *& Exper.* 11 (1981), 651–669]. (3) The top-down style of exposition encouraged by WEB was of course chiefly influenced by Edsger Dijkstra's essays on structured programming in the late 1960s. The less well known work of Pierre-Arnoul de Marneffe ["Holon programming: A survey," Univ. de Liege, Service Informatique, Liege, Belgium, 1973; 135 pp.] also had a significant influence on the author as WEB was being formulated. (4) Edwin Towster has proposed a similar style of documentation in which the programmer is supposed to specify the relevant data structure environment in the name of each submodule ["A convention for explicit declaration of environments and top-down refinement of data," *IEEE Tram.* on Software *Eng.* SE-5 (1979), 374–386]; this requirement seems to make the documentation a bit too verbose, although experience with WEB has shown that any unusual control structure or data structure should definitely be incorporated into the module names on psychological grounds. (5) Discussions with Luis Trabb Pardo in the spring of 1979 were extremely helpful for setting up a prototype version of WEB that was called DOC. *(6)* Ignacio Zabala's extensive experience with DOG, in which he created a full implementation of TEX in PASCAL that was successfully transported to many different computers, was of immense value while WEB was taking its present form. (7) David R. Fuchs made several crucial suggestions about how to make WEB more portable; he and Arthur L. Samuel coordinated the initial installations of WEB on dozens of computer systems, making changes to the code so that it would be acceptable to a wide variety of PASCAL compilers. *(8)* The name WEB itself was chosen in honor of my wife's mother, Wilda Ernestine Bates.

The appendices to this report contain complete WEB programs for the WEAVE and TANGLE processors. A study of these examples, together with an attempt to write WEB programs by yourself, is the best way to understand why WEB has come to be like it is.

General rules. A WEB file is a long string of text that has been divided into individual lines. The exact line boundaries are not terribly crucial, and a programmer can pretty much chop up the WEB file in whatever way seems to look best as the file is being edited; but string constants and control texts must end on the same line on which they begin, since this convention helps to keep errors from propagating. The end of a line means the same thing as a blank space.

Two kinds of material go into WEB files: TEX text and PASCAL text. A programmer writing in WEB should be thinking both of the documentation and of the PASCAL program that he or she is creating; i.e., the programmer should be instinctively aware of the different actions that WEAVE and TANGLE will perform on the WEB file. TEX text is essentially copied without change by WEAVE, and it is entirely deleted by TANGLE, since the TEX text is "pure documentation." PASCAL text, on the other hand, is formatted by WEAVE and it is shuffled around by TANGLE, according to rules that will become clear later. For now the important point to keep in mind is that there are two kinds of text. Writing WEB programs is something like writing TEX documents, but with an additional "PASCAL mode" that is added to TEX's horizontal mode, vertical mode, and math mode.

A WEB file is built up from units called modules that are more or less self-contained. Each module has three parts:

1) A TEX part, containing explanatory material about what is going on in the module.
2) A definition part, containing macro definitions that serve as abbreviations for PASCAL constructions that would be less comprehensible if written out in full each time.
3) A PASCAL part, containing a piece of the program that TANGLE will produce. This PASCAL code should ideally be about a dozen lines long, so that it is easily comprehensible as a unit and so that its structure is readily perceived.

The three parts of each module must appear in this order; i.e., the TEX commentary must come first, then the definitions, and finally the PASCAL code. Any of the parts may be empty.

A module begins with the pair of symbols '`@␣`' or '`@*`', where '␣' denotes a blank space. A module ends at the beginning of the next module (i.e., at the next '`@␣`' or '`@*`'), or at the end of the file, whichever comes first. The WEB file may also contain material that is not part of any module at all, namely the text (if any) that occurs before the first module. Such text is said to be "in limbo"; it is ignored by TANGLE and copied essentially verbatim by WEAVE, so its function is to provide any additional formatting instructions that may be desired in the TEX output. Indeed, it is customary to begin a WEB file with TEX code in limbo that loads special fonts, defines special macros, changes the page sizes, and/or produces a title page.

Modules are numbered consecutively, starting with 1; these numbers appear at the beginning of each module of the TEX documentation, and they appear as bracketed comments at the beginning of the code generated by that module in the **PASCAL** program.

Fortunately, you never mention these numbers yourself when you are writing in WEB. You just say '`@␣`' or '`@*`' at the beginning of each new module, and the numbers are supplied automatically by WEAVE and TANGLE. As far as you are concerned, a module has a name instead of a number; such a name is specified by writing '`@<`' followed by TEX text followed by '`@>`'. When WEAVE outputs a module name, it replaces the '`@<`' and '`@>`' by angle brackets and inserts the module number in small type. Thus, when you read the output of WEAVE it is easy to locate any module that is referred to in another module.

For expository purposes, a module name should be a good description of the contents of that module, i.e., it should stand for the abstraction represented by the module; then the module can be "plugged into" one or more other modules so that the unimportant details of its inner workings are suppressed. A module name therefore ought to be long enough to convey the necessary meaning. Unfortunately, however, it is laborious to type such long names over and over again, and it is also difficult to specify a long name twice in exactly the same way so that WEAVE and TANGLE will be able to match the names to the modules. Therefore a module name can be abbreviated after its first appearance in the WEB file, by typing '`@<`α`...@>`', where α is any string that is a prefix of exactly one module name that appears in the file. For example, '`@<Clear the arrays@>`' can be abbreviated to '`@<Clear...@>`' if no other module name begins with the five letters 'Clear'. Module names must otherwise match character for character, except that consecutive blank spaces and/or tab marks are treated as equivalent to single spaces, and such spaces are deleted at the beginning and end of the name. Thus, '`@< Clear   the arrays @>`' will also match the name in the previous example.

We have said that a module begins with '`@␣`' or '`@*`', but we didn't say how it gets divided up into a TEX part, a definition part, and a **PASCAL** part. The definition part begins with the first appearance of '`@d`' or '`@f`' in the module, and the **PASCAL** part begins with the first appearance of '`@p`' or '`@<`'. The latter option '`@<`' stands for the beginning of a module name, which is the name of the module itself. An equals sign (=) must follow the '`@>`' at the end of this module name; you are saying, in effect, that the module name stands for the PASCAL text that follows, so you say '⟨module name⟩ = PASCAL text'. Alternatively, if the PASCAL part begins with '`@p`' instead of a module name, the current module is said to be unnamed. Note that module names cannot appear in the definition part of a module, because the first '`@<`' in a module signals the beginning of its PASCAL part. Any number of module names might appear in the PASCAL part, however, once it has started.

The general idea of TANGLE is to make a PASCAL program out of these modules in the following way: First all the PASCAL parts of unnamed modules are copied down, in order; this constitutes the initial approximation T_0 to the text of the program. (There should be at least one unnamed module, otherwise there will be no program.) Then all module names that appear in the initial text T_0 are replaced by the PASCAL parts of the corresponding modules, and this substitution process continues until no module names remain. Then all defined macros are replaced by their equivalents, according to certain rules that are explained later. The resulting PASCAL code is "sanitized" so that it will be acceptable to an average garden-variety PASCAL compiler; i.e., lowercase letters are converted to uppercase, long identifiers are chopped, and the lines of the output file are constrained to be at most 72 characters long. All comments will have been removed from this PASCAL program except for the meta-comments delimited by '`@{`' and '`@}`', as explained below, and except for the module-number comments that point to the source location where each piece of the program text originated in the WEB file.

If the same name has been given to more than one module, the **PASCAL** text for that name is obtained by putting together all of the **PASCAL** parts in the corresponding modules. This feature is useful, for example, in a module named 'Global variables in the outer block', since one can then declare global variables in whatever modules those variables are introduced, When several modules have the same name, WEAVE assigns the first module number as the number corresponding to that name, and it inserts a note at the bottom of that module telling the reader to 'See also sections so-and-so'; this footnote gives the numbers of all the other modules having the same name as the present one. The PASCAL text corresponding to a module is usually formatted by WEAVE so that the output has an equivalence sign in place of the equals sign in the WEB file; i.e., the output says '(module name) ≡ PASCAL text'. However, in the case of the second and subsequent appearances of a module with the same name, this '≡' sign is replaced by '+≡', as an indication that the PASCAL text that follows is being appended to the PASCAL text of another module.

The general idea of WEAVE is to make a TEX file from the WEB file in the following way: The first line of the TEX file will be '`\input webmac`'; this will cause TEX to read in the macros that define WEB's documentation conventions. The next lines of the file will be copied from whatever TEX text is in limbo before the first module. Then comes the output for each module in turn, possibly interspersed with end-of-page marks and the limbo material that precedes the next module after a page ends. Finally, WEAVE will generate a cross-reference index that lists each module number in which each PASCAL identifier appears, and it will also generate an alphabetized list of the module names, as well as a table of contents that shows the page and module numbers for each "starred" module.

What is a "starred" module, you ask? A module that begins with '`@*`' instead of '`@␣`' is slightly special in that it denotes a new major group of modules. The '`@*`' should be followed by the title of this group, followed by a period. Such modules will always start on a new page in the TEX output, and the group title will appear as a running headline on all subsequent pages until the next starred module. The title will also appear in the table of contents, and in boldface type at the beginning of its module. Caution: Do not use TEX control sequences in such titles, unless you know that the webmac macros will do the right thing with them. The reason is that these titles are converted to uppercase when they appear as running heads, and they are converted to boldface when they appear at the beginning of their modules, and they are also written out to a table-of-contents tie used for temporary storage while TEX is working; whatever control sequences you use must be meaningful in all three of these modes.

The TEX output produced by WEAVE for each module consists of the following: First comes the module number (e.g., '`\M123.`' at the beginning of module 123, except that '`\N`' appears in place of '`\M`' at the beginning of a starred module). Then comes the TEX part of the module, copied almost verbatim except as noted below. Then comes the definition part and the PASCAL part, formatted so that there will be a little extra space between them if both are nonempty. The definition and PASCAL parts are obtained by inserting a bunch of funny looking TEX macros into the PASCAL program; these macros handle typographic details about fonts and proper math spacing, as well as line breaks and indentation.

When you arc typing TEX text, you will probably want to make frequent reference to variables and other quantities in your PASCAL code, and you will want those variables to have the same typographic treatment when they appear in your text as when they appear in your program. Therefore the WEB language allows you to get the effect of PASCAL editing within TEX text, if you place '`|`' marks before and after the PASCAL material. For example, suppose you want to say something like this:

The characters are placed into *buffer*, which is a **packed array** $[1\,..\,n]$ **of** *char*.

The TEX text would look like this in your WEB file:

```
The characters are placed into |buffer|, which is a |packed array [1..n] of char|.
```

And WEAVE translates this into something you are glad you didn't have to type:

```
The characters are placed into \\{buffer},
which is a \&{packed} \&{array} $[1\to\|n]$ \&{of} \\{char}.
```

Incidentally, the cross-reference index that WEAVE would make, in the presence of a comment like this, would include the current module number as one of the index entries for *buffer* and *char*, even though *buffer* and *char* might not appear in the PASCAL part of this module. Thus, the index covers references to identifiers in the explanatory comments as well as in the program itself, you will soon learn to appreciate this feature. However, the identifiers packed and array and n and of would not be indexed, because WEAVE does not make index entries for reserved words or single-letter identifiers. Such identifiers are felt to be so ubiquitous that it would be pointless to mention every place where they occur.

Speaking of identifiers, the author of WEB thinks that *IdentifiersSeveralWordsLong* look terribly ugly when they mix uppercase and lowercase letters. He recommends that *identifiers_several_words_long* be written with underline characters to get a much better effect. The actual identifiers sent to the PASCAL compiler by TANGLE will have such underlines removed, and TANGLE will check to make sure that two different identifiers do not become identical when this happens. (In fact, TANGLE even checks that the first seven characters of identifiers are unique, when lowercase letters have been converted to uppercase; the number seven in this constraint is more strict than PASCAL's eight, and it can be changed if desired.) The WEAVE processor will properly alphabetize identifiers that have embedded underlines when it makes the index.

Although a module begins with TEX text and ends with PASCAL text, we have noted that the dividing line isn't sharp, since PASCAL text can be included in TEX text if it is enclosed in '|...|'. Conversely, TEX text also appears frequently within PASCAL text, because everything in comments (i.e., between left and right braces) is treated as TEX text. Furthermore, a module name consists of TEX text; thus, a WEB file typically involves constructions like 'if x = 0 then @<Empty the |buffer| array@>' where we go back and forth between PASCAL and TEX conventions in a natural way.

Macros. A WEB programmer can define three kinds of macros to make the programs shorter and more readable:

'@d *identifier* = *constant*' defines a numeric macro, allowing TANGLE to do rudimentary arithmetic.

'@d *identifier* == PASCAL text' defines a *simple* macro, where the identifier will be replaced by the PASCAL text when TANGLE produces its output.

'@d *identifier* (#) == PASCAL text' defines a parametric macro, where the identifier will be replaced by the PASCAL text and where occurrences of # in that PASCAL text will be replaced by an argument.

In all three cases, the identifier must have length greater than one; it must not be a single letter. Furthermore, the identifier must be making its first appearance in the WEB file; a macro must be defined before it is used.

Numeric macros are subject to the following restrictions: (1) The right-hand side of the numeric definition must be made entirely from integer constants, numeric macros, preprocessed strings (see below), and plus signs or minus signs. No other operations or symbols are allowed, not even parentheses, except that PASCAL-like comments (enclosed in braces) can appear. Indeed, comments are recommended, since it is usually wise to give a brief explanation of the significance of each identifier as it is defined. (2) The numeric value must be less than 2^{15} = 32768 in absolute value. (For larger values, you can use '==' in place of '=', thus making use of a simple macro instead of a numeric one. Note, however, that simple macros sometimes have a different effect. For example, consider the three definitions '@d n1=2 @d n2=2+n1 @d n3==2+n1'; then 'x-n2' will expand into 'x-4', while 'x-n3' will expand into 'x-2+2' which is quite different! It is wise to include parentheses in non-numeric macros, e.g., '@d n3==(2+n1)', to avoid such errors.)

When constants are connected by plus signs or minus signs in a PASCAL program, TANGLE does the arithmetic before putting the constant into the output file. Therefore it is permissible to say, for example, 'array [0.. *size* − 1]' if *size* has been declared as a macro; note that PASCAL doesn't allow this kind of compile-time arithmetic if *size* is a constant quantity in the program. Another use of TANGLE's arithmetic is to make case statement labels such as '*flag* + 1' legitimate. Of course, it is improper to change 2+2 into 4 without looking at the surrounding context; many counterexamples exist, such as the phrases '-2+2', 'x/2+2', and '2+2E5'. The program for TANGLE, in the appendix, gives precise details about this conversion, which TANGLE does only when it is safe.

The right-hand sides of simple and parametric macros are required to have balanced parentheses, and the PASCAL texts of modules must have balanced parentheses too. Therefore when the argument to a parametric macro appears in parentheses, both parentheses will belong to the same PASCAL text.

The appendices to this report contain hundreds of typical examples of the usefulness of WEB macros, so it is not necessary to dwell on the subject here. However, the reader should know that WEB's apparently primitive macro capabilities can actually do a lot of rather surprising things. Here is a construction that sheds further light on what is possible: After making the definitions

```
@d two_cases(#)==case j of 1:#(1); 2:#(2); end
@d reset_file(#)==reset(input_file@&#)
```

one can write 'two_cases(reset_file)' and the resulting PASCAL output will be

```
case j of 1: reset(input_file1); 2: reset(input_file2); end
```

(but in uppercase letters. and with _'s removed). The '@&' operation used here joins together two adjacent tokens into a single token, as explained later; otherwise the PASCAL file would contain a space between input_file and the digit that followed it. This trick can be used to provide the effect of an array of files, if you are unfortunate enough to have a PASCAL compiler that doesn't allow such arrays. Incidentally, the cross-reference index made by WEAVE from this example would contain the identifier *input_file* but it would not contain *input_file1* or *input_file2*. Furthermore, TANGLE would not catch the error that INPUTFILE1 and INPUTFILE2 both begin with the same nine letters; one should be more careful when using '@&'! But such aspects of the construction in this trick are peripheral to our main point, which is that a parametric macro name without arguments can be used as an argument to another parametric macro.

Although WEB's macros are allowed to have at most one parameter, the following example shows that this is not as much of a restriction as it may seem at first. Let *amac* and *bmac* be any parametric macros, and suppose that we want to get the effect of

```
@d cmac(#1,#2) == amac(#1) bmac(#2)
```

which WEB doesn't permit. The solution is to make the definitions

```
@d cmac(#) == amac(#) dmac
@d dmac(#) == bmac(#)
```

and then to say 'cmac(x)(y)'.

There is one restriction in the generality of WEB's parametric macros, however: the argument to a parametric macro must not come from the expansion of a macro that has not already been "started." For example, here is one of the things WEB cannot handle:

```
@d arg == (p)
@d identity(#) == #
@p identity arg
```

In this case TANGLE will complain that the identity macro is not followed by an argument in parentheses.

The WEB language has another feature that is somewhat similar to a numeric macro. A preprocessed string is a string that is like a PASCAL string but delimited by double-quote marks (") instead of single-quotes. Double-quote marks inside of such strings are indicated by giving two double-quotes in a row. If a preprocessed string is of length one (e.g., "A" or """"), it will be treated by TANGLE as equivalent to the corresponding ASCII-code integer (e.g., 65 or 34). And if a preprocessed string is not of length one, it will be converted into an integer equal to 128 or more. A string pool containing all such strings will be written out by the TANGLE processor; this string pool file consists of string 128, then string 129, etc., where each string is followed by an end-of-line and prefixed by two decimal digits that define its length. Thus, for example, the empty string "" would be represented in the string pool file by a line containing the two characters '00', while the string """String""" would be represented by '08"String"'. A given string appears at most once in the string pool; the use of such a pool makes it easier to cope with PASCAL's restrictions on string manipulation. The string pool ends with '*nnnnnnnnn', where nnnnnnnnn is a decimal number called the string pool check sum. If any string changes, the check sum almost surely changes too; thus, the '@$' feature described below makes it possible for a program to assure itself that it is reading its own string pool.

Here is a simple example that combines numeric macros with preprocessed strings of length one:

```
@d upper_case_Y = "Y"
@d case_difference = -"y"+upper_case_Y
```

The result is to define *upper_case_Y* = 89, *case_difference* = -32.

Control codes. We have seen several magic uses of '@' signs in WEB files, and it is time to make a systematic study of these special features. A WEB control code is a two-character combination of which the first is '@'.

Here is a complete list of the legal control codes. The letters *L*, *T*, *P*, *M*, *C*, and/or S following each,code indicate whether or not that code is allowable in limbo, in TEX text, in PASCAL text, in module names, in comments, and/or in strings. A bar over such a letter means that the control code terminates the present part of the WEB file; for example, $\overline{L}$ means that this control code ends a section that is in limbo and begins non-L material.

@@ [C, *L*, *M*, P, S, *T*] A double @ denotes the single character '@'. This is the only control code that is legal in limbo, in comments, and in strings.

@␣ [$\overline{L}$, $\overline{P}$, $\overline{T}$] This denotes the beginning of a new (unstarred) module. A tab mark or end-of-line (carriage return) is equivalent to a space when it follows an @ sign.

@* [$\overline{L}$, $\overline{P}$, $\overline{T}$] This denotes the beginning of a new starred module, i.e., a module that begins a new major group. The title of the new group should appear after the @*, followed by a period. As explained above, TEX control sequences should be avoided in such titles unless they are quite simple. When WEAVE and TANGLE read a @*, they print an asterisk followed by the current module number, so that the user can see some indication of progress. The very first module should be starred.

@d [$\overline{P}$, $\overline{T}$] Macro definitions begin with @d (or @D), followed by the PASCAL text for one of the three kinds of macros, as explained earlier.

@f [$\overline{P}$, $\overline{T}$] Format definitions begin with @f (or @F); they cause WEAVE to treat identifiers in a special way when they appear in PASCAL text. The general form of a format definition is 'Qf *l* == *r*', followed by an optional comment enclosed in braces, where 1 and *r* are identifiers; WEAVE will subsequently treat identifier 1 as it currently treats *r*. This feature allows a WEB programmer to invent new reserved words and/or to unreserve some of PASCAL's reserved identifiers. The definition part of each module consists of any number of macro definitions (beginning with Qd) and format definitions (beginning with @f), intermixed in any order.

6p [$\overline{P}$, $\overline{T}$] The PASCAL part of an unnamed module begins with @p (or @P). This causes TANGLE to append the following PASCAL code to the initial program text T_0 as explained above. The WEAVE processor does not cause a '@p' to appear explicitly in the TEX output, so if you are creating a WEB file based on a w-printed WEB documentation you have to remember to insert @p in the appropriate places of the unnamed modules.

@< [*P*, $\overline{T}$] A module name begins with @< followed by TEX text followed by @>; the TEX text should not contain any WEB control sequences except @@, unless these control sequences appear in PASCAL text that is delimited by I . . . |. The module name may be abbreviated, after its first appearance in a WEB file, by giving any unique prefix followed by . . . . , where the three dots immediately precede the closing @>. Module names may not appear in PASCAL text that is enclosed in | . . . |, nor may they appear in the definition part of a module (since the appearance of a module name ends the definition part and begins the PASCAL part).

@' [P, *T*] This denotes an octal constant, to be formed from the succeeding digits. For example, if the WEB file contains '@' 100', the TANGLE processor will treat this an equivalent to '64'; the constant will be formatted as "*'100*" in the TEX output produced via WEAVE. You should use octal notation only for positive constants; don't try to get, e.g., -1 by saying '@'777777777777'.

@" [*P*, *T*] A hexadecimal constant; '@"DODO' tangles to 53456 and weaves to '"DODO'.

@$ [*P*] This denotes the string pool check sum.

@{ [P] The beginning of a "meta comment," i.e., a comment that is supposed to appear in the PASCAL code, is indicated by @{ in the WEB file. Such delimiters can be used as isolated symbols in macros or modules, but they should bc properly nested in the final PASCAL program. The TANGLE processor will convert '@{' into '{' in the PASCAL output file, unless the output is already part of a meta-comment; in the latter case '@{' is converted into ' [', since PASCAL does not, allow nested comments. Incidentally, module numbers are automatically inserted as meta-comments into the PASCAL program, in order to help correlate the outputs of WEAVE and TANGLE (see Appendix C). Mcta-comments can be used to put conditional text into a **PASCAL** program; this helps to overcome one of the limitations of WEB, since the simple macro processing routines of TANGLE do not include the dynamic evaluation of boolean expressions.

@} [P] The end of a "meta comment" is indicated by '@}'; this is converted either into '}' or ']' in the **PASCAL** output, according to the conventions explained for @{ above.

@& [P] The @& operation causes whatever is on its left to be adjacent to whatever is on its right, in the **PASCAL** output. No spaces or line breaks will separate these two items. However, the thing on the left should not be a semicolon, since a line break might occur after a semicolon.

@^ [*P*, *T*] The "control text" that follows, up to the next '@>', will be entered into the index together with the identifiers of the **PASCAL** program; this text will appear in roman type. For example, to put the phrase "system dependencies" into the index, you can type '@^system dependencies@>' in each module that you want to index as system dependent. A control text, like a string, must end on the same line of the WEB file as it began. Furthermore, no WEB control sequences are allowed in a control text, not even @@. (If you need an @ sign you can get around this restriction by typing '\AT!'.)

@. [*P*, *T*] The "control text" that follows will be entered into the index in typewriter type; see the rules for '@^', which is analogous.

@: [*P*, *T*] The "control text" that follows will be entered into the index in a format controlled by the TEX macro '\9', which the user should define as desired; see the rules for '@^', which is analogous.

@t [P] The "control text" that follows, up to the next '@>', will be put into a TEX \hbox and formatted along with the neighboring PASCAL program. This text is ignored by TANGLE, but it can be used for various purposes within WEAVE. For example, you can make comments that mix PASCAL and classical mathematics, as in '$size < 2^{15}$', by typing '|size < @t2^{15}@>|'. A control text must end on the same line of the WEB file as it began, and it may not contain any WEB control codes.

@= [*P*] The "control text" that follows, up to the next '@>', will be passed verbatim to the **PASCAL** program.

@\ [P] Force end-of-line here in the **PASCAL** program file.

@! [P, *T*] The module number in an index entry will be underlined if '@!' immediately precedes the identifier or control text being indexed. This convention is used to distinguish the modules where an identifier is defined, or where it is explained in some special way, from the modules where it is used. A reserved word or an identifier of length one will not be indexed except for underlined entries. An '@!' is implicitly inserted by WEAVE just after the reserved words function, procedure, program, and var, and just after @d and @f. But you should insert your own '@!' before the definitions of types, constants, variables, parameters, and components of records and enumerated types that are not covered by this implicit convention, if you want to improve the quality of the index that you get.

@? [P, *T*] This cancels an implicit (or explicit) '@!', so that the next index entry will not be underlined.

@, [*P*] This control code inserts a thin space in WEAVE's output; it is ignored by TANGLE. Sometimes you need this extra space if you are using macros in an unusual way, e.g., if two identifiers are adjacent.

@/ [P] This control code causes a line break to occur within a **PASCAL** program formatted by WEAVE; it is ignored by TANGLE. Line breaks are chosen automatically by TEX according to a scheme that works 99% of the time, but sometimes you will prefer to force a line break so that the program is segmented according to logical rather than visual criteria. Caution: '@/' should be used only after statements or clauses, not in the middle of an expression; use @| in the middle of expressions, in order to keep WEAVE's parser happy.

@| [*P*] This control code specifies an optional line break in the midst of an expression. For example, if you have a long condition between if and then, or a long expression on the right-hand side of an assignment statement, you can use '@|' to specify breakpoints more logical than the ones that TEX might choose on visual grounds.

@# [P] This control code forces a line break, like @/ does, and it also causes a little extra white space to appear between the lines at this break. You might use it, for example, between procedure definitions or between groups of macro definitions that are logically separate but within the same module,

@+ [P] This control code cancels a line break that might otherwise be inserted by WEAVE, e.g., before the word 'else', if you want to put a short if-then-else construction on a single line. It is ignored by TANGLE.

@; [P] This control code is treated like a semicolon, for formatting purposes, except that it is invisible. You can use it, for example, after a module name when the PASCAL text represented by that module name ends with a semicolon.

The last six control codes (namely '@,', '@/', '@|', '@#', '@+', and '@;') have no effect on the PASCAL program output by TANGLE; they merely help to improve the readability of the w-formatted PASCAL that is output by WEAVE, in unusual circumstances. WEAVE's built-in formatting method is fairly good, but it is incapable of handling all possible cases, because it must deal with fragments of text involving macros and module names; these fragments do not necessarily obey PASCAL's syntax. Although WEB allows you to override the automatic formatting, your best strategy is not to worry about such things until you have seen what WEAVE produces automatically, since you will probably need to make only a few corrections when you are touching up your documentation.

Because of the rules by which every module is broken into three parts, the control codes '@d', '@f', and '@p' are not allowed to occur once the PASCAL part of a module has begun.

Additional features and caveats.

1. The character pairs '(*', '*)', '(.', and '.)' are converted automatically in PASCAL text as though they were '@{', '@}', '[', and ']', respectively, except of course in strings. Furthermore in certain installations of WEB that have an extended character set, the characters '≠', '≤', '≥', '←', '≡', '∧', '∨', '¬', and '∈' can be used as abbreviations for '<>', '<=', '>=', ':=', '==', 'and', 'or', 'not', and 'in', respectively. However, the latter abbreviations are not used in the standard versions of WEAVE. WEB and TANGLE. WEB that are distributed to people who are installing WEB on other computers, and the programs are designed to produce only standard ASCII characters as output if the input consists entirely of ASCII characters.

2. If you have an extended character set, all of the characters listed in Appendix C of The *TEXbook* can be used in strings. But you should stick to standard ASCII characters if you want to write programs that will be useful to the all the poor souls out there who don't have extended character sets.

3. The TEX file output by WEAVE is broken into lines having at most 80 characters each. The algorithm that does this line breaking is unaware of TEX's convention about comments following '%' signs on a line. When TEX text is being copied, the existing line breaks are copied as well, so there is no problem with '%' signs unless the original WEB file contains a line more than eighty characters long or a line with PASCAL text in |...| that expands to more than eighty characters long. Such lines should not have '%' signs.

4. PASCAL text is translated by a "bottom up" procedure that identifies each token as a "part of speech" and combines parts of speech into larger and larger phrases as much as possible according to a special grammar that is explained in the documentation of WEAVE. It is easy to learn the translation scheme for simple constructions like single identifiers and short expressions, just by looking at a few examples of what WEAVE does, but the general mechanism is somewhat complex because it must handle much more than PASCAL itself. Furthermore the output contains embedded codes that cause TEX to indent and break lines as necessary, depending on the fonts used and the desired page width. For best results it is wise to adhere to the following restrictions:

a) Comments in PASCAL text should appear only after statements or clauses; i.e., after semicolons, after reserved words like then and do, or before reserved words like end and else. Otherwise WEAVE's parsing method may well get mixed up.

b) Don't cm-lose long PASCAL texts in |...|, since the indentation and line breaking codes are omitted when the |...| text is translated from PASCAL to TEX. Stick to simple expressions or statements.

5. Comments and module names are not permitted in |...| text. After a '|' signals the change from TEX text to PASCAL text, the next '|' that is not part of a string or control text ends the PASCAL text.

6. A comment must have properly nested occurrences of left and right braces, otherwise WEAVE and TANGLE will not know where the comment ends. However, the character pairs '\{' and '\}' do not count as left and right braces in comments, and the character pair '\|' does not count as a delimiter that begins PASCAL text. (The actual rule is that a character after '\' is ignored; hence in '\\{' the left brace does count.) At present,

TANGLE and WEAVE treat comments in slightly different ways, and it is necessary to satisfy both conventions: TANGLE ignores '|' characters entirely, while WEAVE uses them to switch between TEX text and PASCAL text. Therefore, a comment that includes a brace in a string in |...|—e.g., '{ look at this |"{"|}'—will be handled correctly by WEAVE, but TANGLE will think there is an unmatched left brace. In order to satisfy both processors, one can write '{ look at this \leftbrace\ }', after setting up '\def \leftbrace{|"{"|}'.

7. Reserved words of PASCAL must appear entirely in lowercase letters in the WEB file; otherwise their special nature will not be recognized by WEAVE. You could, for example, have a macro named *END* and it would not be confused with PASCAL's end.

However, you may not want to capitalize macro names just to distinguish them from other identifiers. Here is a way to unreserve PASCAL's reserved word 'type' and to substitute another word 'mtype' in the WEB file.

```
@d type(#) == mem[#].t
@d mtype == t @& y @& p @& e
@f mtype == type
Of type == true
```

In the output of TANGLE, the macro mtype now produces 'TYPE' and the macro type(x) now produces 'MEM[X].T'. In the output of WEAVE, these same inputs produce mtype and type (x), respectively.

8. The @f feature allows you to define one identifier to act like another, and these format definitions are carried out sequentially, as the example above indicates. However, a given identifier has only one printed format throughout the entire document (and this format will even be used before the @f that defines it). The reason is that WEAVE operates in two passes; it processes @f's and cross-references on the first pass and does the output on the second.

9. You may want some @f formatting that doesn't correspond to any existing reserved word. In that case, WEAVE could be extended in a fairly obvious way to include new "reserved words" in its vocabulary. The identifier 'xclause' has in fact been included already as a reserved word, so that it can be used to format the 'loop' macro, where 'loop' is defined to be equivalent to 'while true do'.

10. Sometimes it is desirable to insert spacing into PASCAL code that is more general than the thin space provided by '@,'. The @t feature can be used for this purpose; e.g., '@t\hskip 1in@>' will leave one inch of blank space. Furthermore, '@t\4@>' can be used to backspace by one unit of indentation, since the control sequence \4 is defined in webmac to be such a backspace. (This control sequence is used, for example, at the beginning of lines that contain labeled statements, so that the label will stick out a little at the left.)

11. WEAVE and TANGLE are designed to work with two input files, called *web_file* and change-file, where change-file contains data that overrides selected portions of *web_file*. The resulting merged text is actually what has been called the WEB file elsewhere in this report.

Here's how it works: The change file consists of zero or more "changes," where a change has the form '@x⟨old lines⟩@y⟨new lines⟩@z'. The special control codes @x, @y, @z, which are allowed only in change files, must appear at the beginning of a line; the remainder of such a line is ignored. The ⟨old lines⟩ represent material that exactly matches consecutive lines of the *web-file;* the ⟨new lines⟩ represent zero or more lines that are supposed to replace the old. Whenever the first "old line" of a change is found to match a line in the *web_file*, all the other lines in that change must match too.

Between changes, before the first change, and after the last change, the change file can have any number of lines that do not begin with '@x', '@y', or '@z'. Such lines are bypassed and not used for matching purposes.

This dual-input feature is useful when working with a master WEB file that has been received from elsewhere (e.g., TANGLE.WEB or WEAVE.WEB or TEX.WEB), when changes are desirable to customize the program for your local computer system. You will be able to debug your system-dependent changes without clobbering the master web file; and once your changes are working, you will be able to incorporate them readily into new releases of the master web file that you might receive from time to time.

Appendices. The basic ideas of WEB can be understood most easily by looking at examples of "real" programs. Appendix A shows the WEB input that generated modules 55–59 of the WEAVE program; Appendix B shows the corresponding TEX code output by WEAVE; and Appendix C shows excerpts from the corresponding PASCAL code output by TANGLE.

The complete webs for WEAVE and TANGLE appear as the bulk of this report, in Appendices D and E. The reader should first compare Appendix A to the corresponding portion of Appendix D; then the same material should be compared to Appendices B and C. Finally, if time permits, the reader may enjoy, studying the complete programs in Appendices D and E, since WEAVE and TANGLE contain several interesting aspects, and since an attempt has been made in these appendices to evolve a style of programming that makes good use of the WEB language.

Finally, Appendix F is the 'webmac' file that sets TEX up to accept the output of WEAVE; Appendix G discusses how to use some of its macros to vary the output formats; and Appendix H discusses what needs to be done when WEAVE and TANGLE are installed in a new operating environment.

Performance statistics. The programs in Appendices D and E will optionally keep statistics on how much memory they require. Here is what they printed out when processing themselves:

TANGLE applied to TANGLE (cpu time 15 sec)
```
  Memory usage statistics:
  456 names, 215 replacement texts;
  3396+3361 bytes, 6683+7314+5803 tokens.
```
TANGLE applied to WEAVE (cpu time 29 sec)
```
  Memory usage statistics:
  692 names, 339 replacement texts;
  4576+4294 bytes, 10181+9867+9141 tokens.
```
WEAVE applied to TANGLE (cpu time 45 sec)
```
  Memory usage statistics: 478 names, 2044 cross references, 4158+3725 bytes;
  parsing required 684 scraps, 1300 texts, 3766 tokens, 119 levels;
  sorting required 34 levels.
```
WEAVE applied to WEAVE (cpu time 64 sec)
```
  Memory usage statistics: 737 names, 3305 cross references, 4894+4958 bytes;
  parsing required 684 scraps, 1300 texts, 3766 tokens, 119 levels;
  sorting required 73 levels.
```
The cpu time for PASCAL to process TANGLE.PAS was approximately 13 seconds, and WEAVE.PAS took approximately 26 seconds; thus the tangling time was slightly more than the compiling time. The cpu time for TEX to process TANGLE.TEX was approximately 500 seconds, and WEAVE.TEX took approximately 750 seconds (i.e., about 7 seconds per printed page, where these pages are substantially larger than the pages in a normal book). All cpu times quoted are for a DECsystem-10.

The file TANGLE.WEB is about 125K characters long; TANGLE reduces it to a file TANGLE.PAS whose size is about 42K characters, while WEAVE expands it to a file TANGLE.TEX of about 185K. The corresponding file sizes for WEAVE.WEB, WEAVE.PAS, and WEAVE.TEX are 180K, 89K, and 265K.

The much larger file TEX.WEB led to the following numbers:

TANGLE applied to TEX (cpu time 110 sec)
```
  Memory usage statistics:
  3752 names, 1768 replacement texts;
  41766+41466 bytes, 42445+45061+41039 tokens.
```
WEAVE applied to TEX (cpu time 270 sec)
```
  Memory usage statistics: 3410 names, 19699 cross references, 38899+39362 bytes;
  parsing required 685 scraps, 1303 texts, 3784 tokens, 104 levels;
  sorting required 52 levels.
```
PASCAL did TEX.PAS in about 75 seconds; TEX did TEX.TEX in about 3600.

O, what a tangled web we weave
When first we practise to deceive!
—SIR WALTER SCOTT, *Marmion* 6:17 (1808)

O, what a tangled WEB we weave
When TEX we practise to conceive!
—RICHARD PALAIS (1982)

Appendix A. This excerpt from **WEAVE. WEB** produced modules 55-59 in Appendix D. Note that some of the lines are indented to show the program structure. The indentation is ignored by **WEAVE** and **TANGLE,** but users find that **WEB files** are quite readable if they have some such indentation.

```
@* 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]|.

@<Glob...@>=
@!id_first:0..long_buf_size; {wh ere the current identifier begins in the buffer)
@!id_loc:0..long_buf_size; (just after the current Identifier in the buffer)
@#
O!hash:array [0..hash_size] of sixteen-bits; {heads of hash lists)

0 Initially all the hash lists are empty.

@<Local variables for init...@>=
@!h:0..hash_size; (index Into hash-head array)

0 @<Set init...@>=
for h:=0 to hash size-1 do hash [h] :=0;

0 Acre 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.

@p 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|@>;
@<Compute the name location |p|@>;
if p=name_ptr then @<Enter a new name into the table at position |p|@>;
id_lookup:=p;
end ;

0 A simple hash code is used: If the sequence of
ASCII codes is $c_1c_2\ldots c_m$, Its hash value will be
$$(2^{n-1}c_1+2^{n-2}c_2+\cdots+c_n)\,\bmod\,|hash_size|.$$

@<Compute the hash...@>=
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
```

Appendix B. This excerpt from WEAVE.TEX corresponds to Appendix A.

```
\N55.  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
f irst character is $\\{buf f er) [\\{id\_f irst}] $, and the last is $\\{buf fer}[%
\\{id\_loc}-1]$.

\Y\P$\4\X9:Globals In the outer block\X\mathrel{+}\S$\6
\4\\{id\_first}: \37$0\to\\{long\_buf\_size}$;\C{where the current identifier
begins in the buffer}\6
\4\\{id\_loc}: \37$0\to\\{long\_buf\_size}$;\C{just after the current
identifier in the buffer)\7
\4\\{hash}: \37\&{array} $[0\to\\{hash\_size}]$ \1\&{of}\5
\\{sixteen\_bits};\C{heads of hash lists}\2\par
\fi

\M56. Initially all the hash lists are empty.

\Y\P$\4\X16:Local variables for initialization\X\mathrel{+}\S$\6
\4\|h: \37$0\to\\{hash\_size}$;\C{index into hash-head array}\par
\fi

\M57. \P$\X10:Set initial values\X\mathrel{+}\S$\6
\&{for} $\|h\K0\mathrel{\&{to}}\\{hash\_size}-1$ \1\&{do}\5
$\\{hash}[\|h]\K0$;\2\par
\fi

\M58.  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.

\Y\P\4\&{function}\1\  \37$\\{id\_lookup}(\|t:\\{eight\_bits})$:  \37\\{name%
\_pointer};\C{finds current identifier}\6
\4\&{label} \37\\{found};\6
\4\&{var} \37\|i: \37$0\to\\{long\_buf\_size}$;\C{index i n t o  \\{buffer}}\6
\|h: \37$0\to\\{hash\_size}$;\C{hash code}\6
\|k: \37$0\to\\{max\_bytes}$;\C{index i n t o  \\{byte\_mem}}\6
\|w: \37$0\to\\{ww}-1$;\C{row o f  \\{byte\_mem}}\6
\|l: \37$0\to\\{long\_buf\_size}$;\C{length of the given identifier}\6
\|p: \37\\{name\_pointer};\C{where the identifier is being sought}\2\6
\&{begin} \37$\|l\K\\{id\_loc}-\\{id\_first}$;\C{compute the length)\6
\X59:Compute the hash code \|h\X;\6
\X60:Compute the name location \|p\X;\6
\&{if} $\|p=\\{name\_ptr}$ \1\&{then}\5
\X62:Enter a new name into the table at position \|p\X;\2\6
$\\{id\_lookup}\K\|p$;\6
\&{end};\par
\fi

\M59. A simple hash code is used: If the sequence of
ASCII codes is $c_1c_2\ldots c_m$, its hash value will be
$$(2^{n-1}c_1+2^{n-2}c_2+\cdots+c_n)\,\bmod\,\\{hash\_size}.$$

\Y\P$\4\X59:Compute the hash code \|h\X\S$\6
$\|h\K\\{buffer}[\\{id\_first}]$;\5
$\|i\K\\{id\_first}+1$;\6
\&{while} $\|i<\\{id\_loc}$ \1\&{do}\6
\&{begin} \37$\|h\K(\|h+\|h+\\{buffer}[\|i])\mathbin{\&{mod}}\\{hash\_size}$;\5
$\\{incr}(\|i)$;\6
\&{end}\2\par
\U section~58.\fi
```

Appendix C. The TANGLE processor converts WEAVE.WEB into a syntactically correct (but not very pretty) PASCAL program WEAVE.PAS. The first three and last two lines of output are shown here, together with the lines of code generated by modules 55-62 and the environments of those lines. There are 1546 lines in all; the notation '. . .' stands for portions that are not shown.

Note that, for example, the code corresponding to module 55 begins with '{55:}' and ends with '{:66}'; the code from modules 59-62 has been tangled into the code from module 58.

```
{2:}{4:}{$C-,A+,D-}{[$C+,D+]}{:4}
PROGRAMWEAVE(WEBFILE,CHANGEFILE,TEXFILE);LABEL9999;CONST{8:}
MAXBYTES=45000;MAXNAMES=5000;MAXMODULES=2000;HASHSIZE=353;BUFSIZE=100;
    . . .
TOKPTR:0..MAXTOKS;{MAXTOKPTR,MAXTXTPTR:0..MAXTOKS;}{:53}{55:}
IDFIRST:0..LONGBUFSIZE;IDLOC:0..LONGBUFSIZE;
HASH:ARRAY[0..HASHSIZE]OF SIXTEENBITS;{:55}{63:}CURNAME:NAMEPOINTER;
    . . .
PROCEDURE INITIALIZE;VAR{16:}I:0..127;{:16}{40:}WI:0..1;{:40}{56:}
H:0..HASHSIZE;{:56}{247:}C:ASCIICODE;{:247}BEGIN{10:}HISTORY:=0;{:10}
    . . .
TOKPTR:=1;TEXTPTR:=1;TOKSTART[0]:=1;TOKSTART[1]:=1;{MAXTOKPTR:=1;
MAXTXTPTR:=1;}{:54}{57:}FOR H:=0 TO HASHSIZE- DO HASH[H]:=0;{:57}{94:}
SCANNINGHEX:=FALSE;{:94}{102:}MODTEXT[0]:=32;{:102}{124:}OUTPTR:=1;
    . . .
IF R=0 THEN XREF[P]:=XREFPTR ELSE XMEM[R].XLINKFIELD:=XREFPTR;END;{:51}
{58:}FUNCTION IDLOOKUP(T:EIGHTBITS):NAMEPOINTER;LABEL 31;
VAR I:0..LONGBUFSIZE;H:0..HASHSIZE;K:0..MAXBYTES;W:0..1;
L:0..LONGBUFSIZE;P:NAMEPOINTER;BEGINL:=IDLOC-IDFIRST;{59:}
H:=BUFFER[IDFIRST];I:=IDFIRST+1;
WHILE I<IDLOC DO BEGIN H:=(H+H+BUFFER[I])MOD HASHSIZE;I:=I+1;END{:59};
{60:}P:=HASH[H];
WHILE P<>0 DO BEGIN IF(BYTESTART[P+2]-BYTESTART[P]=L)AND((ILK[P]=T)OR((T
=0)AND(ILK[P]>3)))THEN{61:}BEGIN I:=IDFIRST;K:=BYTESTART[P];W:=P MOD 2;
WHILE(I<IDLOC)AND(BUFFER[I]=BYTEMEM[W,K])DOBEGINI:=I+1;K:=K+1;END;
IF I=IDLOC THEN GOTO 31;END{:61};P:=LINK[P];END;P:=NAMEPTR;
LINK[P]:=HASH[H];HASH[H]:=P;31:{:60};IF P=NAMEPTR THEN{62:}
BEGIN W:=NAMEPTR MOD 2;
IF BYTEPTR[W]+L>MAXBYTES THEN BEGIN WRITELN(TERMOUT);
WRITE(TERMOUT,'! Sorry, ','byte memory',' capacity exceeded');ERROR;
HISTORY:=3;JUMPOUT;END;
IF NAMEPTR+2>MAXNAMES THEN BEGIN WRITELN(TERMOUT);
WRITE(TERMOUT,'! Sorry, ','name',' capacity exceeded');ERROR;HISTORY:=3;
JUMPOUT;END;I:=IDFIRST;K:=BYTEPTR[W];
WHILE I<IDLOC DO BEGIN BYTEMEM[W,K]:=BUFFER[I];K:=K+1;I:=I+1;END;
BYTEPTR[W]:=K;BYTESTART[NAMEPTR+2]:=K;NAMEPTR:=NAMEPTR+1;ILK[P]:=T;
XREF[P]:=0;END{:62};IDLOOKUP:=P;END;{:58}{66:}
FUNCTION MODLOOKUP(L:SIXTEENBITS):NAMEPOINTER;LABEL 31;VAR C:0..4;
    . . .
WRITE(TERMOUT,'(That was a fatal error, my friend.)');END;END{:263};
END.{:261}
```

The WEAVE processor

(Version 2.3)

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

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

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

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

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

define *banner* ≡ 'This␣is␣WEAVE,␣Version␣2.3'

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, which goes to the label *end_of_WEAVE*.

```
define end_of_WEAVE = 9999 {go here to wrap it up}
⟨Compiler directives 4⟩
program WEAVE(web_file, change_file, tex_file);
  label end_of_WEAVE; {go here to finish}
  const ⟨Constants in the outer block 8⟩
  type ⟨Types in the outer block 11⟩
  var ⟨Globals in the outer block 9⟩
    ⟨Error handling procedures 30⟩
  procedure initialize;
    var ⟨Local variables for initialization 16⟩
    begin ⟨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 {g0 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. Hcrc 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. Wc assume that case statements may include a default case that applies if no matching label is found. Thus, wc shall use constructions like

case x of
1: ⟨code for x = 1⟩;
3: ⟨ code for x = 3⟩;
othercases ⟨code for x ≠ 1 and x ≠ 3⟩
endcases

since most PASCAL compilers have plugged this hole in thr language by incorporating some sort of default mechanism. For cxamplc, 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, *othercasea* ≡ *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 ⟩ ≡
  mm-bytes = 45000; { 1/ww times the number of bytes in identifiers, index entries, and module names; must bc less than 65536)
  maz-names = 5000; { number of identifiers, index entries, and module names; must be less than 10240)
  max_modules = 2000; { greater than the total number of modules}
  hash-size = 353; {should be prime }
  buf-size = 100; { maximum length of input line }
  longest-name = 400; {module names shouldn't be longer than this }
  long_buf_size = 500; { ***buf-size + longest-name*** }
  line-length = 80; {lines of TEX output have at most this many characters, should be less than 256)
  max_refs = 20000; { number of cross references; must be less than 65536 }
  maz-toks = 20000; {number of symbols in PASCAL texts being parsed; must be less than 65536 }
  max_texts = 2000; {number of phrases in **PASCAL** texts being parsed; must be less than 10240 }
  max_scraps = 1000; { number of tokens in **PASCAL** texts being parsed }
  stuck-size = 200; { number of simultaneous output levels }

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, 20, 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, and 258.

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 seven-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.

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 as 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 . . 127; { seven-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* ≡ *char* { 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* = 127 {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 stan-
' dard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize most of ***the*** ***xchr*** array properly, without nceding 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] ← '+'; x c h r['54]←●
  xchr['55] ← '-'; xchr['56] ← '.'; xchr['57] ← '/';
  xchr ['60]← '0': xchr['61] t '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] ← ';'; x c h r['74] ← '<';
  xchr['75] ← '='; xchr['76] ← ●   >; xchr['77] ← '?';
  xchr ['100]← '@'; xchr ['101 | ← 'A'; xchr['102] t 'B'; xchr['103] ← 'C'; xchr['104] ← 'D';
  xchr ['105]← 'E'; xchr [ '106 | t 'F'; xchr['107]←●  G;
  xchr ['110]← 'H'; xchr ['111 | ← 'I'; xchr['112] t 'J'; xchr['113]←●  K;xchr['114]←'L';
  xchr ['115]← 'M'; xchr ['116 | t●  N'; xchr['117]←'O';
  xchr ['120]← 'P'; xchr ['121 | ←  ●  Q'; xchr[ '122] t 'R'; xchr['123]← S;xchr['124]←'T';
  xchr ['125]← 'U'; xchr [ '126 | t 'V'; xchr ['127] ← 'W ';
  xchr ['130]← 'X'; xchr ['131 | t 'Y '; xchr [ '132] ← 'Z'; xchr['133] ← '['; xchr ['134] ← '\';
  xchr['135] ← '] '; xchr['136 | t '^'; xchr['137]← ●  -;
  xchr['140] ← '`'; xchr['141 | t 'a'; xchr [ '142] ← 'b'; xchr['143] ← 'c '; xchr['144]← 'd';
  xchr['145] ← 'e'; xchr['146 | t 'f '; xchr['147]←●  g;
  xchr ['150]← 'h'; xchr ['151 | t 'i'; xchr['152]←'j';xchr['153]←●  k'; xchr['154]← '1';
  xchr['155] ← 'm'; xchr['156 | ← 'n'; xchr['157] ← 'o';
  xchr['160] ← 'p'; xchr['161 | t 'q'; xchr [ '162] ← 'r'; xchr[ '163] ← 's'; xchr['164] ← 't';
  xchr['165] ← 'u'; xchr['166 | ♦  ●  v;xchr['167]←'w';
  xchr['170] t 'x'; xchr['171] t 'y'; xchr['172] ← 'z'; xchr['173] ← '{'; x c h r '174]←' I ';
  xchr['175] t '}'; 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-aign = '4    { equivalent to 'and' }
define not-sign = '5    { equivalent to 'not' }
define set-element-sign = '6    { equivalent to 'in' }
define tab-murk = '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 curriage-return = '15    { ASCII code used at end of line}
define left_arrow = '30    {equivalent to ':='}
define not-equal = '32    {equivalent to '<>'}
define less-or-equul = '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 . . last-text-char;

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 at MIT, for example, the code in this module should be changed to

for ***i t*** 1 to '37 do *xchr*[*i*] ← *chr*(*i*);

WEB's character set is essentially identical to MIT'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 ← 1 to '37 do *xchr*[*i*] t '␣';

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*** ← ***first-text-char*** to *last_text_char* do *xord*[*chr*(*i*)] ← ***'40;***
for ***i*** ← 1 to ***'176*** do *xord*[*xchr* [i]] t i;

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 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
```

⟨ Globals in the outer block 9 ⟩ +≡
term-out : text-file ; { the terminal as an output file.}

21. Different systems have different ways of specifying that the output on a certain file will appear on the user's terminal. Here is one way to do this on the PASCAL system that was used in TANGLE's initial development:

⟨ Set initial values 10 ⟩ +≡
  rewrite (term-out, 'TTY: '); { send *term-out* output to the terminal}

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 ≡ break (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. Since these files were listed in the program header, we assume that the PASCAL runtime system has already checked that suitable file names have been given; therefore no additional error checking needs to be done. We will see below that WEAVE reads through the entire input twice.

```
procedure open-input ;  { prepare to read web_file and change-file }
  begin reset (web-file); reset (change-file);
  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);
```

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[f↑]; get(f); incr(limit);
      if buffer[limit − 1] ≠ "␣" then final_limit ← limit ;
      if limit = buf_size then
        begin while ¬eoln(f) do get(f);
        decr(limit);   { keep buffer[buf_size] empty }
        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, 1: 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. This is the only non-local **goto** statement in WEAVE. It is used when no recovery from a particular error has been provided.

Some PASCAL compilers do not implement non-local **goto** statements. In such cases the code that appears at label *end-of-WEAVE* should be copied into the *jump-out* procedure, followed by a call to a system procedure that terminates the program.

```
  define fatal_error(#) ≡
          begin new_line; print(#); error; mark_fatal; jump_out;
          end
⟨Error handling procedures 30⟩ +≡
procedure jump_out;
  begin goto end_of_WEAVE;
  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 seven-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 ***fink, ilk,*** and ***xref*** arrays give further information about namcs. These auxiliary arrays consist of sixteen-bit items.

(Types in the outer block 11 ⟩ +≡
  eight-bits = 0 . . 255; {unsigned one-byte quantity}
  sizteen-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 arc programs that need more than 65536 bytes; TEX is one of these. 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 or 1. (For generality, the first index is actually allowed to run between 0 and ww − 1, where ***ww*** is defined to be 2; the program will work for any positive value of ww, and it can be simplified in obvious ways if ww = 1.)

  define ww = 2 {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 . . ***maz-names]*** of ***sizteen-bits;*** { directory into ***byte_mem*** }
link: array [0 . . ***maz-names]*** of ***sixteen_bits***; { hash table or tree links }
ilk: array [0 . . ***max_names***] of ***sizteen-bits*** ; { type codes or tree links }
xref : array [0 . . ***max_names***] of ***sisteen-6its*** ; { heads of cross-reference lists }

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-sturt ***[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 pointcr 0 is used for undefined module names; we don't want to usc it for the names of identifiers, since 0 stands for a null pointer in a linked list.

Wc usually have ***byte-start [name-ptr*** + ***w]*** = ***byte-ptr [(name-ptr + w)*** mod ww] for $0 \le w < ww$, since these arc 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 . . ***muz-bytes*** ; { first unused position in ***byte-mem*** }

40. (Local variables for initialization IS) +≡
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-sturt [***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, . . . , war-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 *in-like* = 15 { in }
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 *vnr-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 bc able to recognize a module name when given a prefix of that name. A conventional binary seach 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 def_flag = 10240 {must be strictly larger than max_modules}
```

47. ⟨Types in the outer block 11⟩ +≡

```
  xref_number = 0 . . max_refs;
```

48. ⟨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}
    end;
xref_ptr: xref_number; {the largest occupied position in xmem}
xref_switch, mod_xref_switch: 0 . . def_flag; {either zero or def_flag}
```

49. ⟨Set initial values 10⟩ +≡

```
  xref_ptr ← 0; xref_switch ← 0; mod_xref_switch ← 0; num(0) ← 0; 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.

```
define append-xref (#) ≡
        if zref-ptr = max_refs then overflow ('cross␣reference')
        else begin incr (zref-ptr); num(xref_ptr) ← #;
          end
procedure new-xref (p : name-pointer);
  label exit;
  var q: xref_number; {pointer to previous cross reference}
    m, n: sixteen-bits; { new and previous cross-reference value }
  begin if (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 + dej-jlag then
        begin num(q) ← m; return;
        end;
    end;
  append-zref (m); xlink (xref-ptr ) ← q; xref [p] ← xref_ptr ;
exit: 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*, ..., m_2 + *def_flag*, n_l, ..., n_1, in this order. After Phase II, however, the order will be m_1 + *def-flag*, ..., 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] t zref-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 arc 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 ; {fist 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

65. 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-site ; { 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;
. vari: 0.. long-buj-sixe ; {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 }
    1: 0 . . long-buf-size ; {length of the given identifier }
    p: name-pointer; {where the identifier is being sought }
  begin l ← id-lot − id_first; { compute the length }
  ( Compute the hash code h 59 );
  ( Compute the name location p GO);
  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 $c_1c_2 \ldots c_m$, its hash value will be

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

```
( Compute the hash code h 59 ) ≡
  h ← buffer[id_first]; i ← id-first + 1;
  while i < id-lot do
    begin h t (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, $p = name\_ptr$.

```
⟨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
  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] ← #; *aid7*
define *sid5* (#) ≡ *buffer* [5] ← #; *sid6*
define *sid4* (#) ≡ *buffer* [4] ← #; *sid5*
define *sid3* (#) ≡ *buffer* [3] t #; *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; *aid2*
define *id9* ≡ *id-first* ← 1; *sid1*

⟨ Globals in the outer block 9 ⟩ +≡
cur-name : *name-pointer* ; { points to the identifier just inserted}

64. Thc intendcd use of the macros above might not bc immediately obvious, but thc 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*);
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*);
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 . . mux-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 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 t 0 ;
    end;
  mod-lookup ← p;
  end;
```

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

```
  w ← nnme-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] t 0; xref [p] t 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 c to the result of comparing the given name to name p 68⟩ ≡

```
begin $k \gets byte\_start[p]$; $w \gets p \bmod ww$; $c \gets equal$; $j \gets 1$;
while $(k < byte\_start[p + ww]) \land (j \le l) \land (mod\_text[j] = byte\_mem[w, k])$ do
  begin incr(k); incr(j);
  end;
if $k = byte\_start[p + ww]$ then
  if $j > l$ then $c \gets equal$
  else $c \gets extension$
else if $j > l$ then $c \gets prefix$
  else if $mod\_text[j] < byte\_mem[w, k]$ then $c \gets less$
    else $c \gets greater$;
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 $q \gets 0$; $p \gets root$; $count \gets 0$; $r \gets 0$;   {begin search at root of tree}
  while $p \ne 0$ do
    begin ⟨ Set c to the result of comparing the given name to name p 68⟩;
    if c = less then $p \gets llink[p]$
    else if c = greater then $p \gets rlink[p]$
      else begin $r \gets p$; incr(count); $q \gets rlink[p]$; $p \gets llink[p]$;
        end;
    if $p = 0$ then
      begin $p \gets q$; $q \gets 0$;
      end;
    end;
  if $count \ne 1$ 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⟩ +≡
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-buj-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 . . buj-size] of ASCII_code;
change-limit: 0 . . buj-size;  { the last position occupied in change_buffer }
```

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

```
function lines-dent-match: boolean;
  label exit;
  var k: 0 . . buj-size;  {index into the buffers }
  begin lines-dent-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-bufler* 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)* A ¬*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-bufler;
  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-bufler 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") A (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.&) 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;
  for k ← 0 to limit 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␣f ile␣ended␣bef ore␣@y ´); changelimit ← 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-c hanging ; {now it's false }
    incr (line);
    if ¬input_ln (web-fife) then
      begin err-print ( ´ ! ␣WEB␣f ile␣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 Qy, report any discrepancies and return 80⟩ ≡

```
  if limit > 1 then
    if buffer[0] = "@" then
      begin if (buffer[1] ≥ "X") A (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-bufler ; 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-moduZe[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 maybo turn ***off changing*** 84⟩ ≡

```
  begin incr (line );
  if ¬input_ln (change-file) then
    begin err-print ('! ␣Change␣f ile␣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") A (buffer [I] ≤ "Z") then buffer [I] ← 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 loc ← 0 to change-limit do buffer[loc] ← change_buffer[loc];
    limit ← change-limit; changing ← true; line ← other-fine; 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 begincomment = '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-fine-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 bc 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;
  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-fine;
      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(Zoc);
      if c = "I" then got0 done;
    until c = "@";
    if loc ≤ limit then
      begin c t 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-nezt 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-nezt : 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-hez 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 }
  othercases 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 (#) ≡
        if loc ≤ limit then
          begin c ← #; incr (loc);
          end
```

```
⟨ Compress two-symbol combinations like ':=' 97 ⟩ ≡
".": if buffer [loc] = "." then compress (double-dot )
  else if buffer [Zoc] = ")" then compress ("]");
":": if buffer [loc] = "=" then compress (left-arrow);
"=": if buffer [loc] = "=" then compress (equivalence-sign);
">": if buffer [Zoc] = "=" 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")) A (loc > 1) then
    if (buffer [loc − 2] ≤ "9") A (buffer[loc − 2] ≥ "0") then c ← exponent;
  if c ≠ exponent then
    begin decr(loc); id-first ← loc;
    repeat incr (Zoc); d ← buffer [loc];
    until ((d < "0") ∨ ((d > "9") A (d < "A")) ∨ ((d > "Z") A (d < "a")) ∨ (d > "z")) A (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.

```
⟨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);
  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-tezt [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-tezt [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 = tub-murk) ∨ (d = "*") then
    begin err-print ('!␣Section␣name␣didn''t␣end'); goto done;
    end;
  incr (k); mod-tezt [k] ← "@"; incr (Zoc);   {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-tezt [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 (Zoc);
id-lot ← loc;
if loc > limit then
  begin err-print (.! ␣Control␣text␣didn ' 't␣end '); loc ← limit;
  end
else begin loc ← loc + 2;
  if buffer [Zoc − 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-lot* to its ending-plus-one location in the buffer. We also set *Zoc* to the position just after the ending delimiter.

⟨ Scan a verbatim string 107 ⟩ ≡

```
begin id-first ← Zoc; incr (loc); buffer [limit + 1] ← "@"; buffer [limit + 2] ← ">";
while (buffer [loc] ≠ "@") ∨ (buffer [Zoc + 1] ≠ ">") do incr (Zoc);
if Zoc ≥ 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; 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.

```
procedure PASCAL-xref; { makes cross references for PASCAL identifiers }
  label exit;
  var p: name-pointer ;  { a referenced name }
  begin while next-control < format do
    begin if (next-control ≥ identifier) ∧ (next-control ≤ xref-typewriter) then
      begin p ← id-lookup (next_control − identifier); new-xref (p);
      if (ilk [p] = proc_like ) ∨ (ilk [p] = var-like) then xref_switch ← def_flag ;  { implied '@!' }
      end;
    next-control ← get-next ;
    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 next_control ← get_next
    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 $next\_control \ge begin\_pascal$.

```
⟨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.

```
procedure flush_buffer(b : eight_bits; per_cent : boolean);  {outputs out_buf[1 . . b], where b ≤ out_ptr }
  label done;
  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 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)
  else begin for k ← 0 to limit do
      if (buffer[k] ≠ "␣") ∧ (buffer[k] ≠ tab_mark) then return;
    flush_buffer(0, 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 webmac'.

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

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 out ≡ begin oot1
  define out2 ≡ begin oot2
  define out3 ≡ begin oot3
  define out4 ≡ begin oot4
  define out5 ≡ begin oot5
```

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 }
    c, d: ASCII_code; { characters 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); return;
      end;
    if (d = "\") ∧ (out_buf[k − 1] ≠ "\") then   {in this case k > 1}
      begin flush_buffer(k − 1, 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); 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. If the module is changed, we output '*' just after the number.

```
procedure out_mod(m : integer);  {output a module number }
  var k: 0 . . 5; {index into dig }
    a: integer ;  { accumulator }
  begin k ← 0; a t 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;
  if changed-module [m] then out2 ("\")("*");
  end;
```

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

```
procedure out-nume (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 (Zoc);
  if buffer [Zoc − 1] ≠ "@" then
    begin err-print ('! ␣Illegal␣use␣of ␣@␣in␣comment'); Zoc ← Zoc − 2; decr (tok_ptr );
    ⟨ Clear bal and goto done 138⟩;
    end;
  end
else if (c = "\") A (buffer [Zoc] ≠ "@") then
    begin upp-tok (buffer [Zoc]); incr (Zoc);
    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⟩ ≡

```
upp-tok ("␣"); {this is done in case the previous character was '\'}
repeat upp-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 find to find the longest production applicable to the next substring $s_2 s_3 \ldots$; and if that fails, we try to match $s_3 s_4 \ldots$, etc.

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

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

and it means that three consecutive scraps whose respective categories are *open*, *muth*, 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:

$$O_2 = O_1$$
$$M_2 = M_1\, S \,\backslash,\, \textit{opt}\ 6$$

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 arc 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 arc 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* → *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 ezp = 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 vat-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-cut (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');
elaie: 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-apace 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 I . . . I 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-apace, 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* \␣
got0	*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:* \Xn : 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* ⟦$M = O$ \, 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* ⟦$M_2 = M_1\,S$ \, *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***	⟦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***	⟦I = \$ M \$ C⟧	variable declaration
53 ***var-head simp colon*** $\rightarrow$ ***var-head intro***		variable declaration
54 ***var-head stmt*** $\rightarrow$ ***var-head***	⟦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 $O_2 = O_1$, $M_2 = M_1C$.

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] . . . 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 ≥ *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* ≥ *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.

⟨ Globals in the outer block 9 ⟩ +≡
cat: **array** [0 . . *max_scraps*] **of** *eight_bits*; {category codes of scraps}
trans: **array** [0 . . *max_scraps*] **of** *text_pointer*; {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 ‘I . . . I’}
define Zbrace ≡ 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 ( '\\ ' , Zbrace); print-id(r); print (rbrace);
      end; { id_flag }
    2: begin print ( '\&', Zbrace); 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 ´, Zbrace);
  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 ( ´ [f orcel ´);
  big-force : print ( ´ [f f orcel ´);
  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 translations 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* → *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* → *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 ≤ *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 of 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 WC 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 WC 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);
    uar-head: ( Cases for war-head 170⟩;
    othercases do-nothing
    endcases;
    incr (pp);   { if no match was found, WC 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 that starts 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 cut [pp + 1] = moth then
    begin if cat[pp + 2] = colon then squash(pp + 1,2, math,0)(1)
    else if cut [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 cut [pp + 1] = omega then
      begin appl (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 (cut [pp + 2] = terminator) ∨ (cut [pp + 2] = stmt ) then squash (pp, 3, stmt , −2)(5);
    end
  else if cat [pp + 1] = stmt then
      begin appl (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 cut [pp + 1] = casey then
  begin if cut [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 cut [pp + 2] = terminator then
      begin app1 (pp); app(cancel); app (outdent); app2 (pp + 1); reduce(pp, 3, atmt , −2)(8);
      end;
    end
  else if cut [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] =clauae then squash (pp, 2, case-head, 0)(10)
```
This code is used in section 149.

155. ⟨ Cases for *clause* 155⟩ ≡
```
if cut [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 (cut [pp + 1] =cluuse) A (cut [pp + 2] = atmt ) then
  if cat [pp + 3] = elsie then
    begin app2 (pp); app (break_space j; app2 (pp + 2); upp ("␣"); upp (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 cut [pp + 1] = math then
  begin if cut [pp + 2] = simp then
    begin app3(pp); app("}"); reduce(pp,3, math, −1)(15);
    end;
  end
else if cut [pp + 1] = sirnp then
    begin app2(pp); app("}"); reduce(pp,2, muth, −1)(16);
    end
```
This code is used in section 149.

159. ⟨ Cases for intro 159⟩ ≡

```
if cat[pp + 1] = atmt then
  begin app1(pp); app("␣"); app(opt); app("7"); app(cancel); app1(pp + 1);
  reduce(pp, 2, atmt, -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, atmt, -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] = atmt 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)(31);
                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); app1(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 'aq'. 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 . . mux-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 > mux-texts) then
      begin stat if tok-ptr > max_tok_ptr then mux-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 cut [pp . . (pp + 3)] are defined 176 ⟩ ≡
  if lo-ptr < pp + 3 then
    begin repeat if hi-ptr ≤ scrap-ptr then
        begin incr (lo-ptr );
        cut [lo-ptr] ← cut [hi_ptr]; truns [lo-ptr ] ← truns [hi-ptr];
        incr (hi-.ptr );
        end;
    until (hi-ptr > scrap-ptr) ∨ (lo-ptr = pp + 3);
    for i ← lo-ptr + 1 to pp + 3 do cut [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 primed out when *tracing* is set to 1.

```
⟨ Globals in the outer block 9 ⟩ +≡
  debug tracing : 0 . . 2;  { can bc 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 ( ●   u );
    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 ≤ *max-texts*** and ***tok-ptr* + 6 ≤ *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 }
    k: 0 . . long_buf_size ; { index into buffer }
  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) A (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 wc are going to usc 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 }
  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] ← #; trons [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 34
  define sc3 ≡ begin s3
  define sc2 ≡ begin s2
  define sc1 ≡ begin s1
  define sc0(#) ≡
          begin incr (scrap-ptr); cut (scrap-ptr] ← #; trans[scrap_ptr] ← 0;
          end
  define comment-scrap (#) ≡
          begin upp (#); 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: commentscrap(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 tokcns, 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);
no t-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-Zoc do
  begin case buffer[j] of
  "␣","\","#","%","$","^","'","`","{","}","~","&","_": begin app("\");
    end;
  "@": if buffer[j + 1] = "@" then incr(j)
    else err-print('!UDoubleUQ,shouldLbe,used,inUstrings');
  othercases do-nothing
  endcases;
  app_tok(buffer[j]); incr(j);
  end;
sc1("}")(simp);
end
```

This rode 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);
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 subproccdures that convert constants into tokens.

```
⟨ Declaration of the app_octal and app_hex procedures 196 ⟩ ≡
procedure upp-octal;
  begin app ("\"); upp ("0"); upp ("{");
  while (buffer [loc] ≥ "0") A (buffer [loc] ≤ "7") do
    begin app_tok (buffer [loc]); incr (loc);
    end;
  sc1("}")(simp);
  end;
procedure upp-hex;
  begin app ("\"); upp ("H"); upp ("{");
  while ((buffer [loc] ≥ "0") A (buffer [loc] ≤ "9")) V ((buffer [loc] ≥ "A") A (buffer [loc] ≤ "F")) do
    begin upp-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 ***cut*** and ***truns*** arrays, they are unaffected by this translation process.

```
function PASCAL-translate: text-poinfer;
  var p: text-pointer; { points to the translation }
    save-base: 0 . . mux-scraps; { holds original value of scrap-buse }
  begin save_base ← scrap-base; scrap-base ← scrap-ptr + 1; PASCAL-purse; { get the scraps together }
  if next-control ≠ "|" then err_print('!␣Missing␣"|"␣after␣PASCAL␣text');
  upp-tok (cancel); app-comment; { place a cancel token as a final "comment" }
  p ← translute; { make the translation }
  stat if scrap-ptr > mux-scr-ptr then mux-scr-ptr ← scrap_ptr; tats
  scrap-ptr ← scrap-base − 1; scrap-base ← save-base; { scrap the scraps }
  PASCAL_translate ← p;
  end;
```

198. *The outer-purse* routine is to *PASCAL-purse as outer_xref* is to *PASCAL-xref:* It constructs a sequence of scraps for PASCAL text until *next-control* $\geq$ *format.* Thus, it takes care of embedded comments.

```
procedure outer-purse ;  {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-purse
    else begin ⟨Make sure that there is room for at least seven more tokens, three more texts, and one
          more scrap 199⟩;
      upp ("\"); upp ( "C"); upp ( "{"); bal ← copy-comment (1); next-control ← " | ";
      while bal > 0 do
        begin p ← text-ptr ; freeze-text; q ← PASCAL-trunslute;
            { at this point we have tok-ptr + 6 ≤ mux-toks }
        upp (tok-flag + p); upp (inner-tok-flag + q);
        if next-control = " | " then bal ← copy-comment (bal)
        else bal ← 0;  {an error has been reported }
        end;
      upp (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 > mux-toks) ∨ (text-ptr + 3 > max_texts) ∨ (scrap-ptr ≥ mux-scraps) then
    begin stat if scrap-ptr > max_scr_ptr then max_scr_ptr ← acrup-ptr;
    if tok-ptr > max_tok_ptr then max_tok_ptr ← tok-ptr;
    if text-ptr > max_txt_ptr then mux-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. Three main considerations had to be kept in mind when this part of **WEAVE** was designed: (a) There are two modes of output, ***outer*** mode that translates tokens like *force* into line-breaking control sequences, and *inner* mode that 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 arc 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 ***stuck*** array. We call the current ***values cur-end, cur-tok,*** and ***cur-mode.***

The global variable ***stuck-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-stute.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 .modc-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-stute;  { info for non-current levels }
stuck-ptr : 0 . . stuck-size;  { first unused location in the output state stack }
  stat mux-stuck-ptr : 0 . . stack-size;  {largest value assumed by stuck-ptr }
  tats
```

203.
```
⟨ Set initial values 10⟩ +≡
  stat mux-stuck-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 u 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-nume }
    5: begin push-level (cur-name ); cur-mode ← inner; goto restart;
      end; {a = inner_tok_flag + cur-name }
    othercascs a ← identifier { a = id_flag + cur-name }
    cndcases;
    end;
  debug if trouble-•ilootitly 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 ← " I "; p ← PASCAL-translate ; upp (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 reawitch;   { 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 reawitch;
      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-nZ('!␣PASCAL␣text␣in␣section␣name␣didn''t␣end:'); print_nl('<');
        print-id (cur-mod-name); print ('>␣'); mark-error; goto found;
        end;
    b ← byte-men? [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; copy_limbo; finish_line;
  flush_buffer(0, false);  {insert a blank line, it looks nice}
  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);
  ⟨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-apace, math-break, line-break, big-line-break,
          no-line-breuk,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("\")("O")("{");
  while (buffer[loc] ≥ "0") ∧ (buffer[loc] ≤ "7") do
    begin out(buffer[Zoc]); incr(Zoc);
    end; { since buffer[Zimit] = "␣", 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(Zoc);
    end;
  out("}");
  end
```

This code is used in section 222.

225. When we get to the following code we have *next-control* $\geq$ *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-nezt;
      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* $\geq$ *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-purse; ⟨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 wed 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: '\A␣section 101.'; '\U␣sections 370 and 1009.'; '\A,sections 8, 27*, and 64.'.

```
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");
  out4 ("␣")("s")("e")("c"); out4 ("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-zref ← xlink (cur-xref );
          { point to the next cross reference to output }
    if num (cur-xref ) ≤ flag then goto done;
    if (num( xlink (cur-zref )) > 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 230.

238. ⟨ Output the code for the end of a module 238 ⟩ ≡

```
  out3 ("\")("f")("i"); finish-line; flush_buffer (0, 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.

```
⟨ Phase III: Output the cross-reference index 239 ⟩ ≡
  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; 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 102 different lists based on their first characters. (Uppercase letters are put into the same list as the corresponding lowercase letters, since we want to have '$t < TeX <$ to'.) The list for character c begins at location **bucket** [c] and continues through the **blink** array.

```
⟨ 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 127 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*[100].

```
⟨Globals in the outer block 9⟩ +≡
collate: array [0 . . 100] 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 < · ·· < Z = z < 0 < · · · < 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 126 do collate[c − 63] ← c;
  collate[64] ← "_";
  for c ← "a" to "z" do collate[c − "a" + 65] ← c;
  for c ← "0" to "9" do collate[c − "0" + 91] ← 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 ← 100 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 nest-name ← head[sort_ptr];
repeat cur-name t 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 t byte-mem (cur-bank, cur-byte];
    if (c ≤ "Z") A (c ≥ "A") then c ← c + '40;
    end;
  blink [cur-name] ← bucket [c]; bucket [c] ← cur-name;
until next_name = 0; .
decr (sort&r); unbucket (cur-depth + 1);
end
```

This code is used in section 250.

2 5 2 . ⟨ ⓞ t ψ tindex 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 ("\")("I ") 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-zref );
  if cur-val < def_flag then out-mod (cur-val)
  else begin out% ("\")("["); out_mod(cur_val − def_flag); out ("]");
    end;
  cur_xref ← xlink (cur-zref );
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-zref ← xref [cur-name]; cur-xref ← 0;
  repeat next-xref ← xlink (this-xref ); xlink (this-xref ) ← cur-xref ; cur-xref t 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 t 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*** ≤ 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 you 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.

⟨ Globals in the outer block 9 ⟩ +≡
  debug ***trouble-shooting: boolean;*** {is ***debug-help*** wanted? }
ddt : sixteen-bits ; {operation code for the ***debug-help*** routine }
dd: sixteen-bits; { 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 }
term-in: text_file ; { the user's terminal as an input file }
  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 }
  reset (term-in, 'TTY: ', '/I '); {open ***term-in*** as the terminal, don't do a ***get*** }
  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: sixteen-bits ;  { index into various arrays }
  begin incr (debug-skipped);
  if debug-skipped < debug-cycle then return;
  debug-skipped ← 0;
  loop begin write (term-out, ´#´); update-terminal;  { prompt }
    read(term-in, ddt);  { read a list of integers }
    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 [Cc]]);
        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.

```
procedure Phase-I;
  begin ⟨Phase I: Read all the user's text and store the cross references 109⟩;
  end;
procedure Phase-II;
  begin ⟨Phase II: Read all the text again and translate it to TeX form 218⟩;
  end;
  begin initialize; {beginning of the main program}
  print_ln(banner); {print a "banner line"}
  ⟨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⟩;
end-of-WEAVE: stat ⟨Print statistics about memory usage 262⟩; tats
{here files should be closed if the operating system requires it}
  ⟨Print the job history 263⟩;
  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 261.

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 261.

264. System-dependent changes. **This** module should be replaced, if necessary, by changes to the program that **are necessary** to make **WEAVE** work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will bc consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

265. **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.

⟨ 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 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.

The TANGLE processor

(Version 2.0)

1. Introduction. This program converts a WEB file to a PASCAL file. It was written by D. E. Knuth in September, 1981; a somewhat similar SAIL program had been developed in March, 1979. Since this program describes itself, a bootstrapping process involving hand-translation had to be used to get started.

For large WEB files one should have a large memory, since TANGLE keeps all the PASCAL text in memory (in an abbreviated form). The program 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 TANGLE can be identified by looking at the entries for 'system dependencies' in the index below.

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

```
define banner ≡ 'This␣is␣TANGLE,␣Version␣2.0'
```

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*, the PASCAL output goes to file *pascal_file*, and the string pool output goes to file *pool*.

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

```
define end_of_TANGLE = 9999 { go here to wrap it up }
⟨ Compiler directives 4 ⟩
program TANGLE (web_file, change_file, pascal_file, pool);
  label end_of_TANGLE; { go here to finish }
  const ⟨ Constants in the outer block 8 ⟩
  type ⟨ Types in the outer block 11 ⟩
  var ⟨ Globals in the outer block 9 ⟩
    ⟨ Error handling procedures 30 ⟩
  procedure initialize;
    var ⟨ Local variables for initialization 16 ⟩
    begin ⟨ 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 TANGLE'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 TANGLE 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 ≠ 1 and x ≠ 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. The author would have taken the trouble to modify TANGLE so that such extensions were done automatically, if he had not wanted to encourage PASCAL compiler writers to make this important change in PASCAL, where it belongs.)

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 TANGLE.

⟨ Constants in the outer block 8⟩ ≡
buf_size = 100; { maximum length of input line }
max_bytes = 45000; { 1/*ww* times the number of bytes in identifiers, strings, and module names; must be less than 65536 }
max_toks = 50000;
{ 1/*zz* times the number of bytes in compressed PASCAL code; must be less than 65536 }
max_names = 4000; { number of identifiers, strings, module names; must be less than 10240 }
max_texts = 2000; { number of replacement texts, must be less than 10240 }
hash_size = 353; { should be prime }
longest_name = 400; { module names shouldn't be longer than this }
line_length = 72; { lines of PASCAL output have at most this many characters }
out_buf_size = 144; {length of output buffer, should be twice *line_length* }
stack_size = 50; { number of simultaneous levels of macro expansion }
max_id_length = 12; { long identifiers are chopped to this length, which must not exceed *line_length* }
unambig_length = 7; { identifiers must be unique if chopped to this length }
{note that 7 is more strict than PASCAL's 8, but this can be varied }

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, 20, 23, 25, 27, 29, 38, 40, 44, 50, 65, 70, 79, 80, 82, 86, 94, 95, 100, 124, 126, 143, 156, 164, 171, 179, and 185.

This code is used in section 2.

10. ⟨ Set initial values 10 ⟩ ≡
history ← *spotless*;

See also sections 14, 17, 18, 21, 26, 42, 46, 48, 52, 71, 144, 152, and 180.

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 seven-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 arc output.

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 as an upward arrow (t), 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 . . 127; { seven-bit numbers, a subrange of the integers }

See also sections 12, 37, 39, 43, and 78.

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-chur)*** through *chr*(*last_text_char*), inclusive. The following definitions should be adjusted if necessary.

define ***text-char*** ≡ ***char*** { 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*** = 127 {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 . . last-text-char;

See also sections 41, 45, and 51.

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 at MIT, for example, the 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 MIT'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 TANGLE. 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 t 1 to '37 do *xchr*$[i]$ t '␣';

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 ***lust-text-char*** do *xord*$[$*chr*$(i)] \leftarrow$ ***'40;***
for $i \leftarrow$ 1 to '176 do *xord*$[$*xchr*$[i]] \leftarrow i$;

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 WEAVE. 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 TANGLE 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 print (#) ≡ write (term-out, #)   { 'print' means write on the terminal }
define print-172(#) ≡ 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
```

⟨ Globals in the outer block 9 ⟩ +≡
term-out: text-file; { the terminal as an output file }

21. Different systems have different ways of specifying that the output on a certain file will appear on the user's terminal. Here is one way to do this on the PASCAL system that was used in TANGLE's initial development:

(Set initial values 10 ⟩ +≡
  rewrite (term-out, 'TTY: '); { send ***term-out*** output to the terminal }

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 ≡ breuk(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.)

(Glohals in the outer block 9 ⟩ +≡
web_file : ***text-jile;*** { primary input }
change-file: text-jile; { updates }

24. The following code opens the input files. Since these files were listed in the program header, we assume that the PASCAL runtime system has already checked that suitable file names have been given; therefore no additional error chocking needs to be done.

```
procedure open_input ;   { prepare to read web_file and change-file }
  begin reset (web-file); reset (change-file);
  end:
```

25. The main output goes to *pascal_file*, and string pool constants are written to the pool file.

(Globals in the outer block 9⟩ +≡
pascal_file : text _file ;
pool: *text_file* ;

26. The following code opens *pascal_file* and *pool*. Since these files were listed in the program header, we assume that the PASCAL runtime system has checked that suitable external file names have been given.

```
⟨ Set initial values 10 ⟩ +≡
  rewrite(pascal_file); rewrite(pool);
```

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

```
⟨ Globals in the outer block 9 ⟩ +≡
buffer: array [0 . . 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*.

```
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[f↑]; get(f); incr(limit);
        if buffer[limit − 1] ≠ " " then final_limit ← limit;
        if limit = buf_size then
          begin while ¬eoln(f) do get(f);
          decr(limit);   { keep buffer[buf_size] empty }
          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 TANGLE processor operates in two phases: first it inputs the source file and stores a compressed representation of the program, then it produces the PASCAL output from the compressed representation.

The global variable *phase_one* tells whether we are in Phase I or not.

⟨ Globals in the outer block 9 ⟩ +≡
phase_one: *boolean*; { true in Phase I, false in Phase 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 34.

This code is used in section 2.

31. During the first phase, syntax errors are reported to the user by saying

'*err_print*('!␣Error␣message')',

followed by '*jump_out*' if no recovery from the error is provided. This will print the error message followed by 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.

Errors that are noticed during the second phase are reported to the user in the same fashion, but the error message will be followed by an indication of where the error was spotted in the output file.

The actual error indications are provided by a procedure called *error*.

```
  define err_print(#) ≡
          begin new_line; print(#); error;
          end
```

⟨ Error handling procedures 30 ⟩ +≡

```
procedure error; { prints '.' and location of error message }
  var j: 0 . . out_buf_size; { index into out_buf }
    k, l: 0 . . buf_size; { indices into buffer }
  begin if phase_one then ⟨ Print error location based on input buffer 32 ⟩
  else ⟨ Print error location based on output buffer 33 ⟩;
  update_terminal; mark_error;
  debug debug_help; gubed
  end;
```

32. The error locations during Phase I can bc 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( '1. ', 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}
  print( '␣');  {this space separates the message from future asterisks }
  end
```

This code is used in section 31.

33. The position of errors detected during the second phase can be indicated by outputting the partially-filled output buffer, which contains ***out-ptr*** entries.

```
(Print error location based on output buffer 33 ⟩ ≡
  begin print_ln ('.␣(1.', line : 1, ')');
  for j ← 1 to out-ptr do print (xchr [ out-buf [j − 1]]);  { print current partial line }
  print ('...␣');  { indicate that this information is partial }
  end
```

This code is used in section 31.

34. The ***jump-out*** procedure just cuts across all active procedure levels and jumps out of the program. This is the only non-local **goto** statement in TANGLE. It is used when no recovery from a particular error has been provided.

Some PASCAL compilers do not implement non-local **goto** statements. In such cases the code that appears at label *end_of_TANGLE* should be copied into the ***jump-out*** procedure, followed by a call to a system procedure that terminates the program.

```
define fatal-error (#) ≡
        begin new-line ; print (#); error ; mark-fatal; jump-out ;
        end

( Error handling procedures 30 ⟩ +≡
procedure jump-out;
  begin goto end-of-TANGLE;
  end;
```

35. Sometimes the program's behavior is far different from what it should be, and TANGLE prints an error message that is really for the TANGLE 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␣(',#,')')
```

36. An overflow stop occurs if TANGLE's tables aren't large enough.

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

37. **Data structures.** Most of the user's PASCAL code is packed into seven- or eight-bit integers in two large arrays called ***byte-mem*** and ***tok-mem.*** The ***byte-mem*** array holds the names of identifiers, strings, and modules; the *tok_mem* array holds the replacement texts for macros and modules. Allocation is sequential, since things are deleted only during Phase II, and only in a last-in-first-out manner.

Auxiliary arrays ***byte-stud*** and *tok_start* are used as directories to ***byte-mem*** and *tok_mem*, and the link, ***ilk,*** *equiv* , and ***text-link*** 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 }

38. TANGLE 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 tokens, and some programs even need more than 65536 bytes; TEX is one of these. To get around this problem, a slight complication has been added to the data structures: ***byte-mem*** and ***tok-mem*** are two-dimensional arrays, whose first index is either 0 or 1. (For generality, the first index is actually allowed to run between 0 and ww − 1 in ***byte-mem,*** or between 0 and *zz* − 1 in ***tok-mem,*** where ww and *zz* are set to 2 and 3; the program will work for any positive values of ww and *zz*, and it can be simplified in obvious ways if ww = 1 or *zz* = 1.)

  define ww = 2 {we multiply the byte capacity by approximately this amount }
  define *zz* = 3 {we multiply the token 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 }
tok-mem: packed array [0 . . *zz* − 1, 0 . . *max_toks*] of ***eight-bits;*** { tokens }
byte-stud: array [0 . . ***mux-names]*** of ***sixteen-bits*** ; { directory into ***byte-mem*** }
tok-start: array [0 . . ***mux-texts]*** of ***sixteen-bits;*** { directory into ***tok-mem*** }
link: array (0.. ***mux-names]*** of ***sixteen-bits*** ; { hash table or tree links }
ilk: array [0 . . ***mux-numes]*** of ***sixteen-bits*** ; { type codes or tree links }
equiv: array [0 . . ***mux-names]*** of ***sixteen-bits*** ; { info corresponding to names }
text-link: array [0 . . *max_texts*] of ***sixteen-bits*** ; { relates replacement texts }

39. 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-stud*** [p + ww] − 1, inclusive, in the segment of ***byte-mcm*** 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[l,*** ∗]. 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.

Strings arc treated like identifiers; the first character (a double-quote) distinguishes a string from an alphabetic name, but for TANGLE's purposes strings behave like numeric macros. (A 'string' here refers to the strings delimited by double-quotes that TANGLE processes. PASCAL string constants delimited by single-quote marks are not given such special treatment; they simply appear as sequences of characters in the PASCAL texts.) The total number of strings in the string pool is called ***string-ptr*** , and the total number of names in ***byte-mem*** is called *name_ptr*. The total number of bytes occupied in ***byte-mem[w,*** ∗] is called *byte_ptr* [*w*].

We usually have ***byte-dart*** [*name_ptr* + *w*] = ***byte-ptr[(name-ptr*** + *w*) mod *ww*] for 0 ≤ w < ww, since these arc the starting positions for the next ww names to be stored in ***byte-men?.***

  define *length* (#) ≡ ***byte-sttrrt*** [# + ***ww]*** − ***byte-start*** [#] { the length of a name }

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

40. ⟨Globals in the outer block 9⟩ +≡
name-ptr : *name-pointer* ; { first unused position in *byte-start* }
string-ptr : *name-pointer* ; {next number to be given to a string of length $\neq 1$ }
byte-ptt: array [0 . . *ww* − 1] of 0 . . mar-bytes; {first unused position in *byte-mem* }
pool-check-sum: integer ; { sort of a hash for the whole string pool }

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

42. ⟨ 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 t 1; string-ptr ← 128; pool-check-sum ← 271828;
```

43. Replacement texts are stored in *tok-mem,* using similar conventions. A '*text-pointer*' variable is an index into *tok-start,* and the replacement text that corresponds to *p* runs from positions *tok_start*[*p*] to *tok-start* [*p* + *zz*] − 1, inclusive, in the segment of *tok-mem* whose first index is *p* mod *zz*. Thus, when *zz* = 2 the even-numbered replacement texts appear in *tok-mem* [0, ∗] and the odd-numbered ones appear in *tok_mem* [1, ∗]. Furthermore, *text_link*[*p*] is used to connect pieces of text that have the same name, as we shall see later. The pointer 0 is used for undefined replacement texts.

The first position of *tok_mem*[*z*, ∗] that is unoccupied by replacement text is called *tok-ptr [z],* and the first unused location of *tok-start* is called *text-ptr.* We usually have the identity *tok_start*[*text_ptr* + *z*] = *tok-ptr* [(*text-ptr* + *z*) mod *zz*], for $0 \le z < zz$, since these are the starting positions for the next *ww* replacement texts to be stored in *tok-mem.*

⟨ Types in the outer block 11⟩ +≡
text-pointer = *0* . . *mux-texts;* {identifies a replacement text }

44. It is convenient to maintain a variable *z* that is equal to *text-ptr* mod *zz*, so that we always insert tokens into segment *z* of *tok-mem.*

⟨ Globals in the outer block 9⟩ +≡

```
text-ptr : text-pointer ;  {first unused position in tok-start }
tok-ptr: array [0 . . zz − 1] of 0.. mux-toks;  {first unused position in a given segment of tok-mem }
z: 0 .. zz − 1;  { current segment of tok-mem }
stat max_tok_ptr: array [0 . . zz − 1] of 0 . . max-toks;  {largest values assumed by tok-ptr }
tats
```

45. ⟨ Local variables for initialization 16 ⟩ +≡
zi: 0 .. *zz* - 1 ; { to initialize the *tok-mem* indices }

46. ⟨ Set initial values 10⟩ +≡

```
for zi ← 0 to zz − 1 do
  begin tok_start[zi] ← 0; tok_ptr[zi] ← 0;
  end;
tok-start [zz] ← 0;   { this makes replacement text 0 of length zero }
text-ptr ← 1; z ← 1 mod zz;
```

47. Four types of identifiers are distinguished by their ***ilk*** **:**

normal identifiers will appear in the PASCAL program as ordinary identifiers since they have not been defined to be macros; the corresponding value in the *equiv* array for such identifiers is a link in a secondary hash table that is used to check whether any two of them agree in their first ***unambig-length*** characters after underline symbols are removed and lowercase letters are changed to uppercase.

numeric identifiers have been defined to be numeric macros; their ***equiv*** value contains the corresponding numeric value plus 2^{15}. Strings are treated as numeric macros.

simple identifiers have been defined to be simple macros; their equiv value points to the corresponding replacement text.

parametric identifiers have been defined to be parametric macros; like simple identifiers, their ***equiw*** value points to the replacement text.

```
define normal = 0  { ordinary identifiers have normal ilk }
define numeric = 1  { numeric macros and strings have numeric ilk }
define simple = 2  { simple macros have simple ilk }
define parametric = 3  { parametric macros have parametric ilk }
```

40. The names of modules are stored in ***byte-mem*** together with the identifier names, but a hash table is not used for them because TANGLE needs to be able to recognize a module name when given a prefix of that name. A conventional binary seach 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]. If ***p*** is a pointer to a module name, ***equiv*** [p] points to its replacement text, just as in simple and parametric macros, unless this replacement text has not yet been defined (in which case ***equiv [p] = 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 }
```

```
( Set initial values 10 ⟩ +≡
  rlink[0] ← 0;  { the binary search tree starts out with nothing in it }
  equiv [0] ← 0;  { the undefined module has no replacement text }
```

49. Here is a little procedure that prints the text of a given name.

```
procedure print-id(p : name_pointer);  { print identifier or module name }
  var k: 0 . . max_bytes;  {index into byte-mem }
    w: 0.. ww − 1;  {segment of byte-mcm }
  begin if p ≥ name-ytr 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;
```

50. 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.*** If the identifier was not already present, it is inserted with a given ***ilk*** code; and an error message is printed if the identifier is being doubly defined.

Because of the way TANGLE'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* $[id\_loc - 1]$. Furthermore, if the identifier is really a string, the global variable ***double-chars*** tells how many of the characters in the buffer appear twice (namely @@ and ""), since this additional information makes it easy to calculate the true length of the string. The final double-quote of the string is not included in its "identifier," but the first one is, so the string length is *id_loc* − ***id-first*** − ***double-chars*** − 1.

We have mentioned that normal identifiers belong to two hash tables, one for their true names as they appear in the WEB file and the other when they have been reduced to their first ***unurnbig-length*** characters. The hash tables are kept by the method of simple chaining, where the heads of the individual lists appear in the ***hash*** and ***chop-hash*** arrays. If ***h*** is a hash code, the primary hash table list starts at *hash*$[h]$ and proceeds through ***link*** pointers; the secondary hash table list starts at ***chop-hush[h]*** and proceeds through *equiv* pointers. Of course, the same identifier will probably have two different values of ***h.***

The ***id-lookup*** procedure uses an auxiliary array called ***chopped-id*** to contain up to ***unnmbig-length*** characters of the current identifier, if it is necessary to compute the secondary hash code. (This array could be declared local to ***id-lookup,*** but in general we are making all array declarations global in this program, because some compilers and some machine architectures make dynamic array allocation inefficient.)

```
⟨ Globals in the outer block 9 ⟩ +≡
id-first: 0 . . buf_size ;  {where the current identifier begins in the buffer }
id_loc : 0 . . buf-size ;  {just after the current identifier in the buffer }
double-chars : 0 . . buf_size ;  { correction to length in case of strings }

hush, chop-hash: array [0 . . hash-size] of sixteen-bits;  {heads of hash lists}
chopped-id: array (0 . . unambig_length] of ASCII-code;  {chopped identifier }
```

51. Initially all the hash lists are empty.

```
(Local variables for initialization 16) +≡
h: 0 . . hash-size;  { index into hash-head arrays }
```

52. (Set initial values 10) +≡

```
for h ← 0 to hush-size − 1 do
   begin hash[h] ← 0; chop_hash[h] ← 0;
   end;
```

53. Here now is the main procedure for finding identifiers (and strings). The parameter ***t*** is set to *normal* except when the identifier is a macro name that is just being defined; in the latter case, ***t*** will be ***numeric, simple,*** or ***parametric.***

```
function id-lookup (t : eight-bits): name-pointer ;  { finds current identifier }
  label found, not-found;
  var c: eight-bits;  { byte being chopped }
    i: 0 . . buf_size;  { index into buffer }
    h: 0 . . hash-size;  { hash code }
    k: 0 . . mas-bytes; {index into byte-mem }
    w: 0.. ww − 1;  {segment of byte-mem }
    l: 0 . . buf-size;  { length of the given identifier }
    p, q: name-pointer ;  { where the identifier is being sought }
    3: 0.. unambig-length;  { index into chopped-id }
  begin l ← id_loc − id-first;  { compute the length }
  ⟨ Compute the hash code h 54);
  ⟨ Compute the name location p 55 ⟩;
  if (p = name-ptr) ∨ (t ≠ normal) then ⟨Update the tables and check for possible errors 57⟩
  id-lookup ← p;
  end;
```

54. A simple hash code is used: If the sequence of ASCII codes is c_1c_2 . . . c,, its hash value will be

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

```
⟨ Compute the hash code h 54 ⟩ ≡
  h ← buffer [id-first]; i ← id-first + 1;
  while i < id-lot do
    begin h t (h + h + buffer [i]) mod hash-size; incr (i);
    end
```

This code is used in section 53.

55. If the identifier is new, it will be placed in position ***p = name-ptt,*** otherwise ***p*** will point to its existing location.

```
⟨ Compute the name location p 55 ⟩ ≡
  p ← hash [h];
  while p ≠ 0 do
    begin if length(p) = 1 then ⟨ Compare name p with current identifier, goto found if equal 56);
    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 53.

```
56. ⟨ Compare name p with current identifier, goto found if equal 56⟩ ≡
  begin i ← id_first ; k ← byte-start [p]; w t p mod ww;
  while (i < id_loc) A (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 55.

57. ⟨Update the tables and check for possible errors 57⟩ ≡

```
begin if ((p ≠ name-ptr) A (t ≠ normal) A (ilk [p] = normal)) ∨ ((p = name-ptr) A (t =
        normal) A (buffer [id-first] ≠ """")) then ⟨Compute the secondary hash code h and put the first
        characters into the auxiliary array chopped-id 58⟩;
  if p ≠ name-ptr then ⟨Give double-definition error and change p to type t 59⟩
  else ⟨Enter a new identifier into the table at position p 61⟩;
  end
```

This code is used in section 53.

58. The following routine, which is called into play when it is necessary to look at the secondary hash table, computes the same hash function as before (but on the chopped data), and places a zero after the chopped identifier in ***chopped-id*** to serve as a convenient sentinel.

⟨Compute the secondary hash code ***h*** and put the first characters into the auxiliary array ***chopped-id*** 58⟩ ≡

```
begin i ← id-first; s ← 0; h ← 0;
while (i < id-lot) A (s < unambig-length) do
  begin if buffer [i] ≠ "_" then
    begin if buffer [i] ≥ "a" then chopped-id[s] ← buffer [i] − '40
    else chopped-id [s] ← buffer [i];
    h ← (h + h + chopped-id [s]) mod hash-size ; incr (a);
    end;
  incr (i);
  end;
chopped-id [s] ← 0;
end
```

This code is used in section 57.

59. If a macro has appeared before it was defined, TANGLE will still work all right; after all, such behavior is typical of the replacement texts for modules, which act very much like macros. However, an undefined numeric macro may not be used on the right-hand side of another numeric macro definition, so TANGLE finds it simplest to make a blanket rule that macros should be defined before they are used. The following routine gives an error message and also fixes up any damage that may have been caused.

⟨Give double-definition error and change ***p*** to type ***t*** 59⟩ ≡

```
    { now p ≠ name-ptr and t ≠ normal }
  begin if ilk[p] = normal then
    begin err-print ('!␣This␣identifier␣has␣already␣appeared');
    ⟨Remove p from secondary hash table 60⟩;
    end
  else err-print ('!␣This␣identifier␣was␣defined␣before')
  ilk[p] ← t;
  end
```

This code is used in section 57.

60. When we have to remove a secondary hash entry, because a *normal* identifier is changing to another ***ilk,*** the hash code ***h*** and chopped identifier have already been computed.

⟨Remove ***p*** from secondary hash table 60⟩ ≡

```
q ← chop-hash [h];
if q = p then chop-hash[h] ← equiv[p]
else begin while equiv [q] ≠ p do q ← equiv[q];
  cquiv [q] ← cquiv [p];
  end
```

This code ix used in section 59.

61. The following routine could make good use of a generalized *pack* procedure that puts items into just part of a packed array instead of the whole thing.

```
(Enter a new identifier into the table at position p 61) ≡
  begin if (t = normal) A (buffer[id_first] ≠ """") then
    ⟨Check for ambiguity and update secondary hash 62 ⟩;
  w ← name-ptr mod ww; k ← byte_ptr[w];
  if k + l > maz-bytes then overflow( 'byte␣memory');
  if name-ptr > max-names − ww then overflow ( 'name ');
  i ← id-first;  {get ready to move the identifier into byte-mem }
  while i < id-Jot 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);
  if buffer [id-first] ≠ """" then ilk[p] ← t
  else (Define and output a new string of the pool 64 );
  end
```

This code is used in section 57.

62. (Check for ambiguity and update secondary hash 62 ⟩ ≡

```
  begin q ← chop_hash[h];
  while q ≠ 0 do
    begin ( Check if q conflicts with p 63 ) ;
    q ← equiv [q];
    end;
  equiv [p] ← chop_hash[h]; chop_hash[h] ← p;   { put p at front of secondary list }
  end
```

This code is used in section 61.

63. (Check if q conflicts with p 63) ≡

```
  begin k ← byte-start [q]; s ← 0; w ← q mod ww;
  while (k < byte-sturt [q + ww]) A (a < unambig_length) do
    begin c ← byte-mem [w, k];
    if c ≠ "_" then
      begin if c ≥ "a" then c ← c − '40;  { convert to uppercase }
      if chopped_id[s] ≠ c then goto not-found;
      incr (a);
      end;
    incr (k);
    end;
  if (k = byte_start[q + ww]) A (chopped_id[s] ≠ 0) then goto not_found;
  print_nl( '!␣Identifier␣conflict␣with␣');
  for k ← byte_start [q] to byte_start [q + ww] − 1 do print (xchr [byte_mem [w, k]]);
  error; q ← 0;  { only one conflict will be printed, since equiv [0] = 0}
not-found: end
```

This code is used in section 62.

64. We compute the string pool check sum by working modulo a prime number that is large but not so large that overflow might occur.

```
define check-sum-prime ≡ '3777777671   { $2^{29} - 73$ }
```

⟨Define and output a new string of the pool 64⟩ ≡

```
begin ilk [p] ← numeric;   { strings are like numeric macros }
if l − double-chars = 2 then   { this string is for a single character }
  equiv [p] ← buffer [id_first + 1] + '100000
else begin equiv [p] ← string-ptr + '100000; l ← l − double-chars − 1;
  if l > 99 then err-print ('!␣Preprocessed␣string␣is␣too␣long');
  incr (string-ptr); write (pool, xchr ["0" + l div 10], xchr ["0" + l mod 10]);   { output the length }
  pool-check-sum ← pool-check-sum + pool-check-sum + l;
  while pool-check-sum > check-sum-prime do pool-check-sum ← pool-check-sum − check-sum-prime;
  i ← id-first + 1;
  while i < id_loc do
    begin write (pool, xchr [buffer [i]]);   { output characters of string }
    pool-check-sum ← pool-check-sum + pool-check-sum + buffer [i];
    while pool-check-sum > check-sum-prime do pool-check-sum ← pool-check-sum − check-sum-prime;
    if (buffer [i] = """") ∨ (buffer [i] = "@") then i ← i + 2
        { omit second appearance of doubled character }
    else incr (i);
    end;
  write_ln (pool);
  end;
end
```

This code is used in section 61.

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;   {segment 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 ← rlink[0];   { rlink [0] is the root of the tree}
  while p ≠ 0 do
    begin ⟨ Set 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; c ← equal; equiv [p] ← 0;
  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 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 t 1;
while (k < byte_start [p + ww]) A (j ≤ l) A (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 f- extension
else if j > 1 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.

```
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;   {segment 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 ← rlink[0]; count ← 0; r ← 0;   {begin search at root of tree }
  while p ≠ 0 do
    begin ⟨ Set 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 t rlink[p]; p ← llink [p];
        end;
    if p = 0 then
      begin p ← q; q t 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. Tokens. Replacement texts, which represent PASCAL code in a compressed format, appear in *tok-mem* as mentioned above. The codes in these texts are called 'tokens'; some tokens occupy two consecutive eight-bit byte positions, and the others take just one byte.

If $p > 0$ points to a replacement text, *tok-start* $[p]$ is the *tok-mem* position of the first eight-bit code of that text. If *text_link*$[p] = 0$, this is the replacement text for a macro, otherwise it is the replacement text for a module. In the latter case *text_link*$[p]$ is either equal to *module-flag,* which means that there is no further text for this module, or *text_link*$[p]$ points to a continuation of this replacement text; such links are created when several modules have PASCAL texts with the same name, and they also tie together all the PASCAL texts of unnamed modules. The replacement text pointer for the first unnamed module appears in *text-link* [0], and the most recent such pointer is *last-unnamed.*

define *module_flag* $\equiv$ *max_texts* {*final link* in module replacement texts }

⟨ Globals in the outer block 9 ⟩ $+\equiv$
last-unnamed: text-pointer ; { **most** recent replacement text of unnamed module }

71. ⟨Set initial values 10⟩ $+\equiv$
last-unnamed ← 0; *text_link*[0] ← 0;

72. If the first byte of a token is less than '200, the token occupies a single byte. Otherwise we make a sixteen-bit token by combining two consecutive bytes a and b. If $'200 \le a < '250$, then $(a - '200) \times 2^8 + b$ points to an identifier: if $'250 \le a < '320$, then $(a - '250) \times 2^8 + b$ points to a module name; otherwise, i.e., if $'320 \le a < '400$, then $(a - '320) \times 2^8 + b$ is the number of the module in which the current replacement text appears.

Codes less than '200 are 7-bit ASCII codes that represent themselves. In particular, a single-character identifier like 'x' will be a one-byte token, while all longer identifiers will occupy two bytes.

Some of the 7-bit ASCII codes will not be present, however, so we can use them for special purposes. The following symbolic names are used:

param denotes insertion of a parameter. This occurs only in the replacement texts of parametric macros, outside of single-quoted strings in those texts.
begin-comment denotes `@{`, which will become either `{` or `[`.
end-comment denotes `@}`, which will become either `}` or `]`.
octal denotes the `@'` that precedes an octal constant.
hex denotes the `@"` that precedes a hexadecimal constant.
check-sum denotes the `@$` that denotes the string pool check sum.
join denotes the concatenation of adjacent items with no space or line breaks allowed between them (the `@&` operation of WEB).
double-dot denotes '`..`' in PASCAL.
verbatim denotes the `@=` that begins a verbatim PASCAL string. It is also used for the end of the string.
force-line denotes the `@\` that forces a new line in the PASCAL output.

define *param* = 0 { ASCII null code will not appear }
define *verbatim* = '2 { extended ASCII alpha should not appear }
define *force_line* = '3 { extended ASCII beta should 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 *check-sum* = '175 { will not be confused with right brace }
define *join* = '177 { ASCII delete will not appear }

73. The following procedure is used to enter a two-byte value into tok-mem when a replacement text is being generated.

```
procedure store-two-bytes (x : &teen-bits);  { stores high byte, then low byte }
  begin if tok-ptr [z] + 2 > max_toks then overflow ('token');
  tok-mem [z, tok-ptr [z]] ← x div '400;  { this could be done by a shift command }
  tok_mem[z, tok-ptr [z] + 1] ← xmod '400;  { this could be done by a logical and }
  tok-ptr [z] ← tok-ptr [z] + 2;
  end;
```

74. When TANGLE is being operated in debug mode, it has a procedure to display a replacement text in symbolic form. This procedure has not been spruced up to generate a real great format, but at least the results are not as bad as a memory dump.

```
  debug procedure print_repl(p : text-pointer);
  var k: 0 . . max_toks;  { index into tok-mem }
    a: sixteen-bits ;  { current byte(s) }
    zp: 0 . . zz - 1;  { segment of tok-mem being accessed }
  begin if p ≥ text_ptr then print ( 'BAD ')
  else begin k ← tok-start [p]; zp ← pmod zz;
    while k < tok-start [p + zz] do
      begin a ← tok_mem[zp, k];
      if a ≥ 200 then ( Display two-byte token starting with a 75 )
      else (Display one-byte token a 76);
      incr (k);
      end;
    end;
  end;
  gubed
```

75. (Display two-byte token starting with a 75) ≡

```
  begin incr (k);
  if a < '250 then  { identifier or string }
    begin a ← (a - '200) * '400 •t tok_mem[zp, k]; print-id(u);
    if byte-mem [a mod ww , byte-start [a]] = """" then print ( • "')
    else print ('␣');
    end
  else if a < '320 then  { module name}
      begin print ( '@<'); print_id( (a - '250) * '400 + tok_mem[zp, k]); print ('@>');
      end
    else begin a t (a - 320) * '400 + tok-mem [zp, k];  { module number }
      print('@', xchr["{"], a : 1, '@', xchr["}"]);  {can't use right brace between debug and gubed }
      end;
  end
```

This code is used in section 74.

76. ⟨Display one-byte token *a* 76⟩ ≡

```
case a of
begin_comment: print('@', xchr["{"]);
end_comment: print('@', xchr["}"]); {can't use right brace between debug and gubed}
octal: print('@''');
hex: print('@"');
check_sum: print('@$');
param: print('#');
"@": print('@@');
verbatim: print('@=');
force_line: print('@\');
othercases print(xchr[a])
endcases
```

This code is used in section 74.

77. Stacks for output. Let's make sure that our data structures contain enough information to produce the entire **PASCAL** program as desired, by working next on the algorithms that actually do produce that program.

78. The output process uses a stack to keep track of what is going on at different "levels" as the macros are being expanded. Entries on this stack have five parts:

end-field is the ***tok-mem*** location where the replacement text of a particular level will end;
byte-field is the ***tok-mem*** location from which the next token on a particular level will be read;
name-field points to the name corresponding to a particular level;
repl_field points to the replacement text currently being read at a particular level.
mod-field is the module number, or zero if this is a macro.

The current values of these five quantities are referred to quite frequently, so they are stored in a separate place instead of in the ***stuck*** array. We call the current values ***cur-end, cur-byte, cur-name, cur-repl,*** and ***cur-mod.***

The global variable ***stack-ptr*** tells how many levels of output are currently in progress. The end of all output occurs when the stack is empty, i.e., when ***stuck-ptr*** = 0.

⟨ Types in the outer block 11 ⟩ $+\equiv$
output-state = **record** ***end-field: sixteen-bits;*** { ending location of replacement text }
  byte-field: sixteen-bits ; {present location within replacement text }
  name-field: name-pointer ; { ***byte-start*** index for text being output }
  repl_field: ***text-pointer*** ; { ***tok-start*** index for text being output }
  mod-field: 0 . . '27777; { module number or zero if not a module }
  end;

79. define ***cur-end*** $\equiv$ *cur_state.end_field* {current ending location ***in tok-mem*** }
  define ***cur-byte*** $\equiv$ ***cur-state. byte-field*** {location of next output byte in ***tok-mem*** }
  define ***cur-name*** $\equiv$ ***cur-stute.nume-field*** {pointer to current name being expanded}
  define ***cur-repl*** $\equiv$ ***cur-state*** *.repl_field* { pointer to current replacement text }
  define ***cur-mod*** $\equiv$ *cur_state.mod_field* {current module number being expanded }

⟨ Globals in the outer block 9 ⟩ $+\equiv$
cur-state : *output_state* ; { ***cur-end, cur-byte,*** *cur_name*, *cur_repl* }
stuck: **array [1 . .** ***stuck-size]*** **of** ***output-state;*** { info for non-current levels }
stuck-ptr: 0 . . ***stuck-size;*** { first unused location in the output state stack }

80. It is convenient to keep a global variable *zo* equal to ***cur-repl*** **mod** *zz*.

⟨Globals in the outer block 9⟩ $+\equiv$
zo: 0 . . *zz* - 1 ; { the segment of ***tok-mem*** from which output is coming }

81. Parameters must also be stacked. They **arc** placed in *tok_mem* just above the other replacement texts, and dummy parameter 'names' are placed in *byte_start* just after the other names. The variables ***text-ptr*** and ***tok-ptr***[*z*] essentially serve as parameter stack pointers during the output phase, so there is no need for a separate data structure to handle this prohlcm.

82. There is an implicit stack corresponding to meta-comments that are output via `@{` and `@}`. But this stack need not be represented in detail, because we only need to know whether it is empty or not. A global variable ***brace-level*** tells how many items would be on this stack if it were present.

⟨ Globals in the outer block 9 ⟩ $+\equiv$
brute-level: eight-bits; { current depth of `@{` . . . `@}` nesting }

83. **To get** the output process started, we will perform the following initialization steps. We may assume that *text_link* [0] is nonzero, since it points to the PASCAL text in the first unnamed module that generates code; if there are no such modules, there is nothing to output, and an error message will have been generated before we do any of the initialization.

```
⟨Initialize the output stacks 83⟩ ≡
  stack_ptr ← 1; brace_level ← 0; cur_name ← 0; cur_repl ← text_link[0]; zo ← cur_repl mod zz;
  cur_byte ← tok_start[cur_repl]; cur_end ← tok_start[cur_repl + zz]; cur_mod ← 0;
```

This code is used in section 112.

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

```
procedure push_level(p : name_pointer);  { suspends the current level }
  begin if stack_ptr = stack_size then overflow('stack')
  else begin stack[stack_ptr] ← cur_state;  { save cur_end, cur_byte, etc. }
    incr(stack_ptr); cur_name ← p; cur_repl ← equiv[p]; zo ← cur_repl mod zz;
    cur_byte ← tok_start[cur_repl]; cur_end ← tok_start[cur_repl + zz]; cur_mod ← 0;
    end;
  end;
```

85. When we come to the end of a replacement text, the ***pop_level*** subroutine does the right thing: It either moves to the continuation of this replacement text or returns the state to the most recently stacked level. Part of this subroutine: which updates the parameter stack, will be given later when we study the parameter stack in more detail.

```
procedure pop_level;  { do this when cur_byte reaches cur_end }
  label exit;
  begin if text_link[cur_repl] = 0 then  { end of macro expansion }
    begin if ilk[cur_name] = parametric then ⟨Remove a parameter from the parameter stack 91⟩;
    end
  else if text_link[cur_repl] < module_flag then  { link to a continuation }
      begin cur_repl ← text_link[cur_repl];  { we will stay on the same level }
      zo ← cur_repl mod zz; cur_byte ← tok_start[cur_repl]; cur_end ← tok_start[cur_repl + zz]; return;
      end;
  decr(stack_ptr);  { we will go down to the previous level }
  if stack_ptr > 0 then
    begin cur_state ← stack[stack_ptr]; zo ← cur_repl mod zz;
    end;
exit: end;
```

86. The heart of the output procedure is the ***get_output*** routine, which produces the next token of output that is not a reference to a macro. This procedure handles all the stacking and unstacking that is necessary. It returns the value ***number*** if the next output has a numeric value (the value of a numeric macro or string), in which case *cur_val* has been set to the number in question. The procedure also returns the value ***module_number*** if the next output begins or ends the replacement text of some module, in which case *cur_val* is that module's number (if beginning) or the negative of that value (if ending). And it returns the value *identifier* if the next output is an identifier of length two or more, in which case *cur_val* points to that identifier name.

```
  define number = '200  { code returned by get_output when next output is numeric }
  define module_number = '201  { code returned by get_output for module numbers }
  define identifier = '202  { code returned by get_output for identifiers }

⟨Globals in the outer block 9⟩ +≡
cur_val: integer;  { additional information corresponding to output token }
```

87. If *get-output* finds that no more output remains, it returns the value zero.

```
function get-output: sixteen-bits; {returns next token after macro expansion }
  label restart, done ,found;
  var a: sixteen-bits ; { value of current byte }
    b: eight-bits; {byte being copied }
    bal: sixteen-bits; { excess of ( versus ) while copying a parameter }
    k: 0 . . max-bytes; {index into byte-mem }
    w: 0.. ww - 1; {segment of byte-mem }
  begin restart: if stack_ptr = 0 then
    begin a ← 0; goto found;
    end;
  if cur-byte = cur-end then
    begin cur-val ← -cur-mod; pop-level;
    if cur-val = 0 then goto restart;
    a ← module-number; goto found;
    end;
  a ← tok_mem[zo, cur-byte]; incr (cur-byte);
  if a < '200 then { one-byte token }
    if a = param then ⟨ Start scanning current macro parameter, goto restart 92 ⟩
    else goto found;
  a ← (a - '200) * '400 + tok_mem[zo, cur-byte]; incr (cur-byte);
  if a < '24000 then { '24000 = ( '250 - '200) * '400 }
    ⟨Expand macro a and goto found, or goto restart if no output found 89⟩;
  if a < '50000 then { '50000 = ( '320 - '200 ) * '400 }
    ⟨Expand module a - '24000, goto restart 88 ⟩;
  cur-val ← a - '50000; a ← module-number; cur-mod ← cur-val;
found: debug if trouble-shooting then debug-help; gubed
  get-output ← a;
  end;
```

88. The user may have forgotten to give any PASCAL text for a module name, or the PASCAL text may have been associated with a different name by mistake.

```
⟨ Expand module a - '24000, goto restart 88 ⟩ ≡
  begin a ← a - '24000;
  if equiv [a] ≠ 0 then push-level(a)
  else if a ≠ 0 then
      begin print_nl ( ´ ! ␣Not␣present : ␣< ´); print-id(a); print ( ´> ´); error;
      end;
  goto restart;
  end
```

This code is used in section 87.

89. ⟨Expand macro a and **goto** *found*, or **goto** *restart* if no output found 89⟩ ≡

```
begin case ilk[a] of
normal: begin cur_val ← a; a ← identifier;
  end;
numeric: begin cur_val ← equiv[a] − '100000; a ← number;
  end;
simple: begin push_level(a); goto restart;
  end;
parametric: begin ⟨Put a parameter on the parameter stack, or goto restart if error occurs 90⟩;
  push_level(a); goto restart;
  end;
othercases confusion('output')
endcases;
goto found;
end
```

This code is used in section 87.

90. We come now to the interesting part, the job of putting a parameter on the parameter stack. First we pop the stack if necessary until getting to a level that hasn't ended. Then the next character must be a '('; and since parentheses are balanced on each level, the entire parameter must be present, so we can copy it without difficulty.

⟨Put a parameter on the parameter stack, or **goto** *restart* if error occurs 90⟩ ≡

```
while (cur_byte = cur_end) ∧ (stack_ptr > 0) do pop_level;
if (stack_ptr = 0) ∨ (tok_mem[zo, cur_byte] ≠ "(") then
  begin print_nl('!␣No␣parameter␣given␣for␣'); print_id(a); error; goto restart;
  end;
⟨Copy the parameter into tok_mem 93⟩;
equiv[name_ptr] ← text_ptr; ilk[name_ptr] ← simple; w ← name_ptr mod ww; k ← byte_ptr[w];
debug if k = max_bytes then overflow('byte␣memory');
byte_mem[w, k] ← "#"; incr(k); byte_ptr[w] ← k;
gubed { this code has set the parameter identifier for debugging printouts }
if name_ptr > max_names − ww then overflow('name');
byte_start[name_ptr + ww] ← k; incr(name_ptr);
if text_ptr > max_texts − zz then overflow('text');
text_link[text_ptr] ← 0; tok_start[text_ptr + zz] ← tok_ptr[z]; incr(text_ptr); z ← text_ptr mod zz
```

This code is used in section 89.

91. The *pop_level* routine undoes the effect of parameter-pushing when a parameter macro is finished:

⟨Remove a parameter from the parameter stack 91⟩ ≡

```
begin decr(name_ptr); decr(text_ptr); z ← text_ptr mod zz;
stat if tok_ptr[z] > max_tok_ptr[z] then max_tok_ptr[z] ← tok_ptr[z];
tats { the maximum value of tok_ptr occurs just before parameter popping }
tok_ptr[z] ← tok_start[text_ptr];
debug decr(byte_ptr[name_ptr mod ww]); gubed
end
```

This code is used in section 85.

92. When a parameter occurs in a replacement text, we treat it as a simple macro in position (*name_ptr* -1):

```
⟨ Start scanning current macro parameter, goto restart 92⟩ ≡
  begin push_level(name_ptr − 1); goto restart;
  end
```

This code is used in section 87.

93. Similarly, a *param* token encountered as we copy a parameter is converted into a simple macro call for *name_ptr* − 1. Some care is needed to handle cases like *macro*(#; *print*('#)')); the # token will have been changed to *param* outside of strings, but we still must distinguish 'real' parentheses from those in strings.

```
  define app_repl(#) ≡
          begin if tok_ptr[z] = max_toks then overflow('token');
          tok_mem[z, tok_ptr[z]] ← #; incr(tok_ptr[z]);
          end
⟨ Copy the parameter into tok_mem 93 ⟩ ≡
  bal ← 1; incr(cur_byte); { skip the opening '(' }
  loop begin b ← tok_mem[zo, cur_byte]; incr(cur_byte);
    if b = param then store_two_bytes(name_ptr + '77777)
    else begin if b ≥ '200 then
        begin app_repl(b); b ← tok_mem[zo, cur_byte]; incr(cur_byte);
        end
      else case b of
        "(": incr(bal);
        ")": begin decr(bal);
          if bal = 0 then goto done;
          end;
        "'": repeat app_repl(b); b ← tok_mem[zo, cur_byte]; incr(cur_byte);
          until b = "'"; { copy string; don't change bal }
        othercases do_nothing
        endcases;
      app_repl(b);
      end;
    end;
done:
```

This code is used in section 90.

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

First, we want to make sure that the output is broken into lines not exceeding *line-length* characters per line, where these breaks occur at valid places (e.g., not in the middle of a string or a constant or an identifier, not between '<' and '>', not at a '@&' position where quantities arc being joined together). Therefore we assemble the output into a buffer before deciding where the line breaks will appear. However, we make very little attempt to make "logical" line breaks that would enhance the readability of the output; people are supposed to read the input of TANGLE or the TEXed output of WEAVE, but not the tangled-up output. The only concession to readability is that a break after a semicolon will be made if possible, since commonly used "pretty printing" routines give better results in such cases.

Second, we want to decimalize non-decimal constants, and to combine integer quantities that are added or subtracted, because PASCAL doesn't allow constant expressions in subrange types or in case labels. This means we want to have a procedure that treats a construction like (E-15+17) as equivalent to '(E+2)', while also leaving '(1E-15+17)' and '(E-15+17*y)' untouched. Consider also '-15+17.5' versus '-15+17..5'. We shall not combine integers preceding or following *, /, div, mod, or @&. Note that if y has been defined to equal -2, we must expand 'x*y' into 'x*(-2)'; but 'x-y' can expand into 'x+2' and we can even change 'x - y mod z' to 'x + 2 mod z' because PASCAL has a nonstandard **mod** operation!

The following solution to these problems has been adopted: An array *out-buf* contains characters that have been generated but not yet output, and there are three pointers into this array. One of these, *out-ptr*, is the number of characters currently in the buffer, and we will have $1 \le$ *out_ptr* $\le$ *line-length* most of the time. The second is *break_ptr*, which is the largest value $\le$ *out_ptr* such that we are definitely entitled to end a line by outputting the characters *out-buf* [1 . . (*break-ptr* − 1)]; we will always have *break-ptr* $\le$ *line-length*. Finally, *semi-ptr* is either zero or the largest known value of a legal break after a semicolon or comment **on** the current line; we will always have *semi-ptr* $\le$ *break-ptr* .

⟨ Globals in the outer block 9 ⟩ +≡
out-buf : **array** [0 . . *out-buf-size*] **of** *ASCII_code* ; { assembled characters }
out_ptr : 0 . . *out-buf-size* ; { first available place in *out_buf* }
break-ptr : 0 . . *out-buf-size* ; { last breaking place in *out_buf* }
semi-ptr: 0 . . *out-buf-size*; {last semicolon breaking place in *out_buf* }

95. Besides having those three pointers, the output process is in one of several states:

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

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

sign means that the last item in the buffer is to be followed by + or -, depending on whether ***out-app*** is positive or negative.

sign_val means that the decimal equivalent of $|$ *out_val* $|$ should be appended to the buffer. ***If out-val*** $<$ 0, or if ***out-val*** = 0 and ***last-sign*** < 0, the number should be preceded by a minus sign. Otherwise it should be preceded by the character ***out-sign*** unless ***out-sign*** = 0; the ***out-sign*** variable is either 0 or "␣" or "+".

sign-val-sign is like sign-val, but also append + or - afterwards, depending on whether ***out-app*** is positive or negative.

sign-val-val is like ***sign-val,*** but also append the decimal equivalent of ***out-app*** including its sign, using ***last-sign in*** case ***out-app*** = 0.

misc means none of the above.

For example, the output buffer and output state run through the following sequence as we generate characters from '(x-15+19-2)':

output	*out-buf*	*out-state*	*out-sign*	*out_val*	*out-app*	*last-sign*
(	(	*misc*				
x	(x	***num-or-id***				
−	(x	***sign***			-1	-1
15	(x	***sign-val***	"+"	-15		-15
+	(x	***sign-val-sign***	"+"	-15	+1	+1
19	(x	*sign_val_val*	"+"	-15	+19	+1
−	(x	***sign-yal-sign***	"+"	+4	-1	-1
2	(x	*sign_val_val*	"+"	+4	-2	-2
)	(x+2)	*misc*				

At each stage wc havc put as much into the buffer as possible without knowing what is coming next. Examples like 'x-0.1' indicate why ***last-sign*** is needed to associate the proper sign with an output of zero.

In states ***num-or-id, unbreakable,*** and *misc* the last item in the buffer lies between *break_ptr* and ***out-ptr*** -1, inclusive; in the other states ***we*** have ***break-ptr = out-ptr.***

The numeric values assigned to ***num-or-id,*** etc., have been chosen to shorten some of the program logic; for example, the program makes use of the fact that ***sign*** + **2** = ***sign-val-sign.***

define *misc* = 0 { state associated with special characters }
define ***num-or-id*** = 1 { state associated with numbers and identifiers}
define sign = 2 {stato associated with pending + or - }
define ***sign-val = num*** *_or_id* + 2 { state associated with pending sign and value }
define ***sign-val-sign = sign + 2*** { ***sign-val*** followed by another pending sign }
define ***sign-val-val*** = *sign_val* + ***2*** { *sign_val* followed by another pending value }
define ***unbreakable*** = *sign_val_val* + 1 { state associated with **@&** }

⟨ Globals in the outer block 9 ⟩ +≡
out-state : eight-bits ; {current status of partial output }
out_val , ***out-app : integer*** ; { pending values }
out-sign : ASCII-code ; {sign to use if appending ***out-val*** $\geq$ 0 }
last-sign: -- 1 . . +1; { sign to use if appending a zero }

96. During the output process, *line* will equal **the** number of the next line to be output.

⟨ Initialize the output buffer 96 ⟩ ≡

```
out-state ← misc; out-ptr ← 0; break-ptr ← 0; semi-ptr ← 0; out_buf[0] ← 0; line ← 1;
```

This code is used in section 112.

97. Here is a routine that is invoked when *out-ptr* > *line-length* or when it is time to flush out the final line. The *flush_buffer* procedure often writes out the line up to the current *break-ptr* position, then moves the remaining information to the front of out-buf . However, it prefers to write only up to *semi-ptr* , if the residual line won't be too long.

```
define check-break ≡
        if out-ptr > line-length then flush_buffer
procedure flush_buffer ; {writes one line to output file }
  var k: 0 . . out-buf-size; { index into out-buf }
    b: 0 . . out-buf-size; {value of break-ptr upon entry }
  begin b ← break-ptr ;
  if (semi-ptr ≠ 0) ∧ (out-ptr − semi-ptr ≤ line-length) then break_ptr ← semi-ptr;
  for k ← 1 to break-ptr do write (pascal_file, xchr[out_buf [k − 1]]);
  write_ln(pascal_file); incr(line);
  if line mod 100 = 0 then
    begin print ('.');
    if line mod 500 = 0 then print (line : 1);
    update-terminal; { progress report }
    end;
  if break-ptr < out-ptr then
    begin if out-buf [ break-ptr] = "␣" then
      begin incr (break-ptr ); { drop space at break }
      if break-ptr > b then b ← break-ptr ;
      end;
    for k ← break-ptr to out-ptr − 1 do out_buf [k − break_ptr] ← out-buf [k];
    end;
  out-ptr ← out-ptr − break-ptr ; break-ptr ← b − break-ptr ; semi-ptr ← 0;
  if out-ptr > line-length then
    begin err-print ('!␣Long␣line␣must␣be␣truncated'); out-ptr ← line-length;
    end;
  end;
```

98. ⟨ Empty the last line from the buffer 98 ⟩ ≡

```
breuk-ptr ← out-ptr ; semi-ptr ← 0; flush_buffer ;
if brace-level ≠ 0 then err-print ('!␣Program␣ended␣at␣brace␣level␣', brace-level : 1);
```

This code is used in section 112.

99. Another simple and useful routine appends the decimal equivalent of a nonnegative integer to thc output buffer.

```
define app (#) ≡
          begin out-buf [out-ptr] ← #; incr (out-ptr ); {append a single character }
          end
procedure app_val (v : integer ); { puts v into buffer, assumes v ≥ 0 }
  var k: 0 . . out-buf-size; { index into out_buf }
  begin k ← out-buf-size; { first we put the digits at the very end of out_buf }
  repeat out-buf [k] ← v 'mod 10; v ← v div 10; decr (k);
  until v = 0;
  repeat incr (k); app (out-6uf [k] + "0");
  until k = out-buf-size ; { then we append them, most significant first }
  end;
```

100. The output states are kept up to date by the output routines, which are called *send-out, send_val*, and *send-sign.* The *send-out* procedure has two parameters: *t* tells the type of information being sent and *v* contains the information proper. Some information may also be passed in the array *out-contrib.*

If *t* = *misc* then v is a character to bc output.
If *t* = *str* then v is the length of a string or something like '<>' *in out_contrib.*
If t = ident then v is the length of an identifier in *out-contrib.*
If *t* = *frac* then v is the length of a fraction and/or exponent in *out-contrib.*

```
define str = 1 { send-out code for a string }
define ident = 2 { send-out code for an identifier}
define frac = 3 { send-out code for a fraction }
( Globals in the outer block 9⟩ +≡
dut-contrib: array [1 . . line-length] of ASCII-code ; {a contribution to out-buf }
```

101. A slightly subtle point in the following code is that the user may ask for a join operation (i.e., @&) following whatever is being sent out. We will see later that *join* is implemented in part by calling *send-out (frac, 0).*

```
procedure send-out (t : eight-bits; v : sixteen-bits); {outputs v of type t }
  label restart;
  var k: 0 . . line-length; {index into out-contrib }
  begin ( Get the buffer rcady for appending the new information 102⟩;
  if t ≠ misc then
    for k ← 1 to v do app( out_contrib[k])
  else upp (v);
  check-break;
  if (t = misc ) A ((v = " ; ") V (v = "}")) then
    begin semi-ptr ← out-ptr; break_ptr ← out-ptr;
    end;
  if t ≥ ident then out-state ← num_or_id {t = ident or frac }
  else out-state ← misc {t = str or misc }
  end;
```

102. Here is where the buffer states for signs and values collapse into simpler states, because we are about to append something that doesn't combine with the previous integer constants.

We use an ASCII-code trick: Since **", " − 1 = "+" and "," + 1 = "-", we have ", " − c = sign of c, when** $|c|$ **= 1.**

```
⟨ Get the buffer ready for appending the new information 102⟩ ≡
restart: case out-state of
  num-or-id: if t ≠ frac then
        begin break-ptr ← out-ptr ;
        if t = ident then app ("␣");
        end;
  sign: begin app (", " − out-app); check-break; break-ptr ← out-ptr ;
     end;
  sign-val, sign-val-sign : begin (Append out-vaf to buffer 103 ⟩;
     out-state t out-state − 2; got0 restart;
     end;
  sign_val_val: ( Reduce sign_val_val to sign-val and goto restart 104 ⟩;
  miac: if t ≠ frac then break-ptr ← out-ptr ;
     othercases do-nothing   { this is for unbreakable state }
  endcases
```

This code is used in section 101.

```
103. (Append out-d to buffer 103⟩ ≡
  if (out-val < 0) v (( out-val = 0) A (lust-sign < 0)) then app ("-")
  else if out-sign > 0 then app( out-sign);
  app_val (aba (out-val)); check-break;
```

This code is used in sections 102 and 104.

```
104.  ( Reduce sign_val_val to sign-val and goto restart 104 ⟩ ≡
  begin if (t = fruc) V ((Contribution is** or / or DIV or MOD 105⟩) then
     begin ( Append out_val to buffer 103 ⟩;
     out-sign ← "+"; out_val ← out-app;
     end
  else out_val ← out-val + out-app;
  out-state ← sign-val; got0 restart;
  end
```

This code is used in section 102.

```
105.  ( Contribution is * or / or DIV or MOD 105⟩ ≡
  ((t = ident) ∧ (v = 3) ∧ (( (out-contrib [1] = "D") A (out-contrib [2] = "I" ) A (out-contrib [3] = "V")) V
        (( out-contrib [1] = "M") A (out-contrib [2] = "O") A (out-contrib [3] = "D")))) V
        ((t = misc) A ((v = "*") V (v = "/")))
```

This code is used in section 104.

106. The following routine is called with v = ±1 when a plus or minus sign is appended **to the output. It** extends PASCAL to allow repeated signs (e.g., '--' is equivalent to '+'), rather than to give an error message. The signs following 'E' in real constants are treated as part of a fraction, so they are not seen by this **routine.**

```
procedure send-sign (v : integer);
  begin case out-state of
  sign, sign-val-sign : out-app ← out-app * v;
  sign-val: begin out-app ← v; out-state ← sign-val-sign;
    end;
  sign_val_val: begin out_val ← out_val + out-app; out-app ← v; out-state ← sign-val-sign;
    end;
  othercases begin break-ptr ← out-ptr; out-app ← v; out-state ← sign;
    end
  endcases;
  last-sign ← out_app;
  end;
```

107. When a (signed) integer value is to be output, *we* call *send_val.*

```
  define bad-case = 666 { this is a label used below }
procedure send_val (v : integer ); { output the (signed) value v }
  label bad-case, { go here if we can't keep v in the output state }
    exit;
  begin case out-state of
  num_or_id: begin ⟨If previous output was DIV or MOD, goto bad-case 110⟩;
    out-sign ← "␣"; out-state ← sign-val; out_val ← v; break-ptr ← out-ptr ; last-sign ← +1;
    end;
  miac: begin ⟨If previous output was * or /, goto bad-case 109⟩;
    out-sign ← 0; out-state ← sign-vul; out-val ← v; break-ptr ← out-ptr ; last-sign ← +l;
    end;
  ⟨Handle cases of send_val when out-state contains a sign 108⟩
  othercases goto bad-case
  endcases;
  return;
bad-case : ⟨ Append the decimal value of v,. with parentheses if negative 111⟩;
exit: end;
```

108. ⟨ Handle cases of *send_val* when *out-state* contains a sign 108 ⟩ ≡

```
sign: begin out-sign t "+"; out_state ← sign-vul; out-val ← out-app * v;
  end;
sign_val: begin out-state t sign-val-val; out-app ← v;
  err-print ('!␣Two␣numbers␣occurred␣without␣a␣sign␣between␣them');
  end;
sign_val_sign: begin out_state ← sign_val_val; out-app ← out-app * v;
  end:
sign-val-val: begin out-val ← out-vnl + out-app; out-app ← v;
  err-print ('!␣Two␣numbers␣occurred␣without␣a␣sign␣between␣them');
  end;
```

This code is used in section 107.

109. ⟨ If previous output was * or /, **goto** *bad-case* 109 ⟩ ≡

```
if (out-ptr = break-ptr + 1) ∧ ((out-buf [break-ptr] = "*") ∨ (out-buf [break-ptr] = "/")) then
   goto bad-case
```

This code is used in section 107.

110. ⟨If previous output was DIV or MOD, **goto** *bad-case* 110⟩ ≡

```
if (out-ptr = break-ptr + 3) ∨ ((out-ptr = break-ptr + 4) ∧ (out-buf [break-ptr] = "␣")) then
   if ((out_buf [out-ptr − 3] = "D") ∧ (out-buf [out-ptr − 2] = "I") ∧ (out-buf [out-ptr − 1] = "V"))∨
   ((out-buf [out-ptr − 3] = "M") ∧ (out-buf [out-ptr − 2] = "O") ∧ (out-buf [out-ptr − 1] = "D")) then
   goto bad-case
```

This code is used in section 107.

111. ⟨Append the decimal value of v, with parentheses if negative 111⟩ ≡

```
if v ≥ 0 then
   begin if out-state = num-or-id then
      begin break-ptr ← out-ptr ; app ("␣");
      end;
   app_val (v); check-break; out-state ← num-or-id;
   end
else begin app ("("); app ("-"); app_val(−v); app (")"); check-break; out-state ← miac;
   end
```

This code is used in section 107.

112. The big output switch. To complete the output process, we need **a** routine that takes the results of *get_output* and feeds them to ***send-out, send_val***, or send-sign. This procedure '***send-the-output*** ' **will be** invoked just once, as follows:

```
⟨Phase II: Output the contents of the compressed tables 112⟩ ≡
  if text_link[0] = 0 then
    begin print_nl('!␣No␣output␣was␣specified.'); mark_harmless;
    end
  else begin print_nl('Writing␣the␣output␣file'); update_terminal;
    ⟨Initialize the output stacks 83⟩;
    ⟨Initialize the output buffer 96⟩;
    send_the_output;
    ⟨Empty the last line from the buffer 98⟩;
    print_nl('Done.');
    end
```

This code is used in section 182.

113. **A** many-way switch is used to send the output:

```
  define get_fraction = 2 { this label is used below }
procedure send_the_output;
  label get_fraction, { go here to finish scanning a real constant }
    reswitch, continue;
  var cur_char: eight_bits; { the latest character received }
    k: 0 .. line_length; {index into out_contrib }
    j: 0 .. max_bytes; {index into byte_mem }
    w: 0 .. ww - 1; {segment of byte_mem }
    n: integer; { number being scanned }
  begin while stack_ptr > 0 do
    begin cur_char ← get_output;
  reswitch: case cur_char of
    0: do_nothing; { this case might arise if output ends unexpectedly }
    ⟨Cases related to identifiers 116⟩
    ⟨Cases related to constants, possibly leading to get_fraction or reswitch 119⟩
    "+", "-": send_sign("," - cur_char);
    ⟨Cases like <> and := 114⟩
    "'": ⟨Send a string, goto reswitch 117⟩;
    ⟨Other printable characters 115⟩: send_out(misc, cur_char);
    ⟨Cases involving @{ and @} 121⟩
    join: begin send_out(frac, 0); out_state ← unbreakable;
      end;
    verbatim: ⟨Send verbatim string 118⟩;
    force_line: ⟨Force a line break 122⟩;
    othercases err_print('!␣Can''t␣output␣ASCII␣code␣', cur_char : 1)
    endcases;
    goto continue;
  get_fraction: ⟨Special code to finish real constants 120⟩;
  continue: end;
  end;
```

114. ⟨**Cases** like <> and := 114⟩ ≡

```
and-sign: begin out-contrib [1] ← "A"; out-contrib [2] ← "N"; out-contrib [3] ← "D"; send_out (ident, 3);
  end;
not-sign: begin out-contrib [1] ← "N"; out-contrib [2] ← "O"; out-contrib [3] ← "T"; send-out (ident, 3);
  end;
set-element-sign: begin out-contrib [1] ← "I"; out-contrib [2] ← "N"; send-out (ident, 2);
  end;
or-sign: begin out-contrib [1] ← "O"; out-contrib [2] ← "R"; send-out (ident, 2);
  end;
left-arrow: begin out-contrib [1] ← ":"; out-contrib [2] ← "="; send-out (atr, 2);
  end;
not-equal: begin out-contrib [1] ← "<"; out-contrib [2] ← ">"; send-out (atr, 2);
  end;
leas-or-equal: begin out-contrib [1] ← "<"; out-contrib [2] ← "="; send-out (atr, 2);
  end;
greater-or-equal: begin out_contrib [1] ← ">"; out-contrib [2] ← "="; send-out (atr, 2);
  end;
equivalence-sign: begin out-contrib [1] ← "="; out-contrib [2] ← "="; send-out (atr, 2);
  end;
double-dot: begin out-contrib [1] ← "."; out-contrib [2] ← "."; send-out (atr, 2);
  end;
```

This code is used in section 113.

115. Please don't ask how all of the following characters can actually get through TANGLE outside of strings. It seems that """" and "{" cannot actually occur at this point of the program, but they have been included just in case TANGLE changes.

If TANGLE is producing code for a PASCAL compiler that uses '(.' and '.)' instead of square brackets (e.g., on machines with EBCDIC code), one should remove "[" and "]" from this list and put them into the preceding module in the appropriate way. Similarly, some compilers want '^' to be converted to '@'.

⟨Other printable characters 115⟩ ≡

```
"!", """", "#", "$", "%", "&", "(", ")", "*", ",", "/", ":", ";", "<", "=", ">", "?", "@", "[", "\", "]", "^",
    "_", "`", "{", "|"
```

This code is used in section 113.

116. Single-character identifiers represent themselves, while longer ones appear in ***byte-mem.*** **All** must be converted to uppercase, with underlines removed. Extremely long identifiers must be chopped.

(Some PASCAL compilers work with lowercase letters instead of uppercase. If this module of TANGLE **is** changed, it's also necessary to **change** from uppercase to lowercase in the modules that **are listed in the index** under "uppercase" .)

```
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, #
```

```
⟨ Cases related to identifiers 116⟩ ≡
"A", up_to("Z"): begin out_contrib[1] ← cur-char; send-out (ident, 1);
  end;
"a", up-to ( "z"): begin out-contrib [1] ← cur-char − '40; send-out (ident, 1);
  end;
identifier: begin k ← 0; j ← byte-start [cur_val]; w ← cur_val mod ww;
  while (k < max-id-length) ∧ (j < byte-start [cur_val + ww]) do
    begin incr (k); out-contrib [k] ← byte-mem [w, j]; incr (j);
    if out-contrib[k] ≥ "a" then out_contrib[k] ← out-contrib(k] − '40
    else if out-contrib [k] = "_" then decr (k);
    end;
  send-out (ident, k);
  end;
```

This code is used in section 113.

117. After sending a string, we need to look ahead at the next character, in order to see if there **were two** consecutive single-quote marks. Afterwards ***we go*** to ***reswitch*** to process the next character.

```
⟨Send a string, goto reswitch 117⟩ ≡
. begin k ← 1; out-contrib [1] ← "'";
  repeat if k < line-length then incr(k);
    out-contrib [k] ← get-output;
  until (out-contrib [k] = "'") ∨ (stuck-ptr = 0);
  if k = line-length then err-print ('!␣String␣too␣long ');
  send-out (str , k); cur-char ← get-output;
  if cur-char = "'" then out-state t unbreakable;
  goto reswitch;
  end
```

This code is used in section 113.

118. Sending a verbatim string is similar, hut wc don't have to look ahead.

```
⟨Send verbatim string 118⟩ ≡
  begin k ← 0;
  repeat if k < line-length then incr(k);
    out-contrib [k] ← get-output;
  until (out-contrib [k] = verbatim) ∨ (stack-ptr = 0);
  if k = line-length then err_print('!␣Verbatim␣string␣too␣long');
  send-out (str , k−1);
  end
```

This code is used in section 113.

119. In order to encourage portable software, TANGLE complains if the constants get dangerously close to the largest value representable on a 32-bit computer $(2^{31} - 1)$.

define *digits* ≡ "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"

```
⟨ Cases related to constants, possibly leading to get-fraction or reswitch 119 ⟩ ≡
digits: begin n ← 0;
  repeat cur-char ← cur-chat − "0";
    if n ≥ '1463146314 then err_print('!␣Constant␣too␣big')
    else n ← 10 * n + cur-char;
    cur-char ← get-output;
  until (cur-char > "9") ∨ (cur-char < "0");
  send_val(n); k ← 0;
  if cur-char = "e" then cur-char ← "E";
  if cur-char = "E" then goto aet-fraction
  else goto reswitch;
  end;
check-sum: send_val (pool-check-sum);
octal: begin n ← 0; cur-char t "0";
  repeat cur-char ← cur-char − "0";
    if n ≥ '2000000000 then err-print ('!␣Constant␣too␣big')
    else n ← 8 * n + cur-char;
    cur-char ← get-output ;
  until (cur-char > "7") ∨ (cur-char < "0");
  send_val (n); goto reswitch;
  end;
hez: begin n ← 0; cur-char ← "0";
  repeat if cur-char ≥ "A" then cur-char ← cur-char + 10 − "A"
    else cur-char ← cur-char − "0";
    if n ≥ "8000000 then err-print ( '!␣Constant␣too␣big')
    else n ← 16 * n + cur-char;
    cur-char ← get-output ;
  until (cur-char > "F") ∨ (cur-char < "0") ∨ ((cur-char > "9") A (cur-char < "A"));
  send_val(n); goto reswitch;
  end;
number : send_val(cur_val);
".": begin k ← 1; out-contrib [1] ← "."; cur-char ← get-output;
  if cur-char = "." then
    begin out-contrib [2] ← "."; send-out (str ,2);
    end
  else if (cur-char ≥ "0") A (cur-char ≤ "9") then goto get-fraction
    else begin send-out (misc, "."); goto reswitch;
      end;
  end;
```

This code is used in section 113.

120. The following code appears at label '`get_fraction`', when we want to scan to the end of a real constant. The first *k* characters of a fraction have already been placed in *out_contrib*, and *cur_char* is the next character.

```
⟨ Special code to finish real constants 120 ⟩ ≡
  repeat if k < line_length then incr(k);
    out_contrib[k] ← cur_char; cur_char ← get_output;
    if (out_contrib[k] = "E") ∧ ((cur_char = "+") ∨ (cur_char = "-")) then
      begin if k < line_length then incr(k);
      out_contrib[k] ← cur_char; cur_char ← get_output;
      end
    else if cur_char = "e" then cur_char ← "E";
  until (cur_char ≠ "E") ∧ ((cur_char < "0") ∨ (cur_char > "9"));
  if k = line_length then err_print('!␣Fraction␣too␣long');
  send_out(frac, k); goto reswitch
```

This code is used in section 113.

121. Some PASCAL compilers do not recognize comments in braces, so the comments must be delimited by '`(*`' and '`*)`'. In such cases the statement '*send_out*(*misc*, "{")' that appears here should be replaced by '**begin** *out_contrib*[1] ← "("; *out_contrib*[2] ← "*"; *send_out*(*str*, 2); **end**', and a similar change should be made to '*send_out*(*misc*, "}")'.

```
⟨ Cases involving @{ and @} 121 ⟩ ≡
begin_comment: begin if brace_level = 0 then send_out(misc, "{")
  else send_out(misc, "[");
  incr(brace_level);
  end;
end_comment: if brace_level > 0 then
    begin decr(brace_level);
    if brace_level = 0 then send_out(misc, "}")
    else send_out(misc, "]");
    end
  else err_print('!␣Extra␣@}');
module_number: begin if brace_level = 0 then send_out(misc, "{")
  else send_out(misc, "[");
  if cur_val < 0 then
    begin send_out(misc, ":"); send_val(−cur_val);
    end
  else begin send_val(cur_val); send_out(misc, ":");
    end;
  if brace_level = 0 then send_out(misc, "}")
  else send_out(misc, "]");
  end;
```

This code is used in section 113.

```
122. ⟨ Force a line break 122 ⟩ ≡
  begin while out_ptr > 0 do
    begin if out_ptr ≤ line_length then break_ptr ← out_ptr;
    flush_buffer;
    end;
  out_state ← misc;
  end
```

This code is used in section 113.

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

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

124. 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⟩ +≡
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 . . buf-size ;  { the last character position occupied in the buffer }
Zoc: 0 . . 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 }
```

125. 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 }
```

126. 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 }
```

127. 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;
```

128. 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 $(change\_limit = 0) \land \neg changing$ 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 129⟩;
  ⟨Skip to the next nonblank line; return if end of file 130⟩;
  ⟨Move buffer and limit to change_buffer and change_limit 131⟩;
exit: end;
```

129. 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 129⟩ ≡
  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 128.

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

```
⟨Skip to the next nonblank line; return if end of file 130⟩ ≡
  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 128.

131.
```
⟨Move buffer and limit to change_buffer and change_limit 131⟩ ≡
  begin change_limit ← limit;
  for k ← 0 to limit do change_buffer[k] ← buffer[k];
  end
```

This code is used in sections 128 and 132.

132. 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 changing ← ¬changing; {now it's true }
    incr(line);
    if ¬input_ln(change_file) then
      begin err_print('!␣Change␣file␣ended␣before␣@y'); change_limit ← 0; changing ← ¬changing;
          {false again }
      return;
      end;
    ⟨If the current line starts with @y, report any discrepancies and return 133⟩;
    ⟨Move buffer and limit to change_buffer and change_limit 131⟩;
    changing ← ¬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;
```

133. ⟨If the current line starts with `@y`, report any discrepancies and **return** 133⟩ ≡

```
  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 132.

134. ⟨Initialize the input system 134⟩ ≡

```
  open_input; line ← 0; other_line ← 0;
  changing ← true; prime_the_change_buffer; changing ← ¬changing;
  limit ← 0; loc ← 1; buffer[0] ← "␣"; input_has_ended ← false;
```

This code is used in section 182.

135. 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 ⟨Read from change_file and maybe turn off changing 137⟩;
  if ¬changing then
    begin ⟨Read from web_file and maybe turn on changing 136⟩;
    if changing then goto restart;
    end;
  loc ← 0; buffer[limit] ← "␣";
  end;
```

136. ⟨Read from *web_file* and maybe turn on *changing* 136⟩ ≡

```
  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 135.

137. ⟨Read from *change_file* and maybe turn off *changing* 137⟩ ≡

```
  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 135.

138. 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 138⟩ ≡

```
  if change_limit ≠ 0 then { changing is false }
    begin for loc ← 0 to change_limit do buffer[loc] ← change_buffer[loc];
    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 183.

139. Important milestones are reached during the input phase when certain control codes are sensed.

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

```
  define ignore = 0  {control code of no interest to TANGLE}
  define control_text = '203  {control code for '@t', '@^', etc.}
  define format = '204  {control code for '@f'}
  define definition = '205  {control code for '@d'}
  define begin_pascal = '206  {control code for '@p'}
  define module_name = '207  {control code for '@<'}
  define new_module = '210  {control code for '@␣' and '@*'}

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;  {string pool check sum}
  "␣", tab_mark: control_code ← new_module;  {beginning of a new module}
  "*": begin print('*', module_count + 1 : 1); update_terminal;  {print a progress report}
    control_code ← new_module;  {beginning of a new module}
    end;
  "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}
  "T", "t", "^", ".", ":": control_code ← control_text;  {control text to be ignored}
  "&": control_code ← join;  {concatenate two tokens}
  "<": control_code ← module_name;  {beginning of a module name}
  "=": control_code ← verbatim;  {beginning of PASCAL verbatim mode}
  "\": control_code ← force_line;  {force a new line in PASCAL output}
  othercases control_code ← ignore  {ignore all other cases}
  endcases;
  end;
```

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

```
function skip-ahead: eight-bits;   { skip to next control code }
  label done ;
  var c: eight-bits;   { control code found }
  begin loop
    begin if Zoc > limit then
      begin get-line ;
      if input-has-ended then
        begin c ← new-module; goto done;
        end;
      end;
    buffer[limit + 1] ← "@";
    while buffer [loc] ≠ "@" do incr (loc);
    if loc ≤ limit then
      begin loc ← loc + 2; c t control-code (buffer [loc − 1]);
      if (c ≠ ignore) v (buffer [Zoc − 1] = ">") then goto done;
      end;
    end;
done: skip-ahead ← c;
  end;
```

141. The ***skip:comment*** procedure reads through the input at somewhat high speed until finding the first unmatched right brace or until coming to the end of the file. It ignores characters following '\' characters, since all braces that aren't nested are supposed to be hidden in that way. For example, consider the process of skipping the first comment below, where the string containing the right brace has been typed as ` \ . \} ' in the WEB file.

```
procedure skip-comment;   { skips to next unmatched '}' }
  label exit ;
  var bal: eight-bits ;   { excess of left braces }
    c: ASCII-code ;   { current character }
  begin bal ← 0;
  loop begin if loc > limit then
      begin get-line ;
      if input-has-ended then
        begin err-print ('!␣Input␣ended␣in␣mid-comment '); return;
        end;
      end;
    c ← buffer [Zoc]; kncr (Zoc); ⟨Do special things when c = "@", "\", "{", "}"; return at end 142⟩;
    end;
exit: end;
```

142. ⟨Do special things when **c = "@", "\", "{", "}"**; **return** at end 142⟩ ≡

```
if c = "@" then
  begin c ← buffer [foe];
  if (c ≠ "␣") A (c ≠ tab_mark) A (c ≠ "*") A (c ≠ "z") A (c ≠ "Z") then incr(loc)
  else begin err-print ('!␣Section␣ended␣in␣mid-comment'); decr (loc); return;
    end
  end
else if (c = "\") A (buffer [loc] ≠ "@") then incr (foe)
  else if c = "{" then incr (bal)
    else if c = "}" then
        begin if bal = 0 then return;
        decr (bal);
        end
```

This code is used in section 141.

143. Inputting the next token. As stated above, TANGLE's most interesting input procedure is the *get-next* routine that inputs the next token. However, the procedure isn't especially difficult.

In most cases the tokens output by *get-nezt* have the form used in replacement texts, except that two-byte tokens are not produced. An identifier that isn't one letter long is represented by the output *'identifier'*, and in such a case the global variables *id-first* and *id_loc* will have been set to the appropriate values needed by the *id-lookup* procedure. A string that begins with a double-quote is also considered an *identifier*, and in such a case the global variable *double-chars* will also have been set appropriately. Control codes produce the corresponding output of the *control-code* function above; and if that code is *module_name*, the value of *cur-module* will point to the *byte-start* entry for that module name.

Another global variable,- *scanning-hex*, is *true* during the time that the letters A through F should be treated as if they were digits.

```
(Globals in the outer block 9⟩ +≡
cur-module: name-pointer;   { name of module just scanned }
scanning-hex: boolean;   { are we scanning a hexadecimal constant? }
```

144. (Set initial values 10 ⟩ +≡

```
  scanning-hex ← false ;
```

145. At the top level, *get-next* is a multi-way switch based on the next character in the input buffer. A *new-module* code is inserted at the very end of the input file.

```
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-hez then ( Go to found if c is a hexadecimal digit, otherwise set scanning-hex ← false 146⟩;
  case c of
  "A", up_to ("Z"),"a", up-to ( "z"): ( Get an identifier 148⟩;
  """": ( Get a preprocessed string 149⟩;
  "@": ⟨Get control code and possible module name 150⟩;
  ( Compress two-symbol combinations like ':=' 147 ⟩
  "␣", tab-murk: goto restart ;  { ignore spaces and tabs }
  "{": begin skip_comment ; got0 restnrt ;
    end;
  othercases do-nothing
  endcases;
found: debug if trouble-shooting then debug-help; gubed
  get-next ← c;
  end;
```

146. (Go to found if c is a hexadecimal digit, otherwise set scanning-hex ← false 146⟩ ≡

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

This code in used in section 145.

147. 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 c ← #; incr(loc);
        end
```

⟨ Compress two-symbol combinations like ':=' 147 ⟩ ≡

```
".": 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 145.

148. We have to look at the preceding character to make sure this isn't part of a real constant, before trying to find an identifier starting with 'e' or 'E'.

⟨ Get an identifier 148 ⟩ ≡

```
begin if ((c = "e") ∨ (c = "E")) ∧ (loc > 1) then
  if (buffer[loc − 2] ≤ "9") ∧ (buffer[loc − 2] ≥ "0") then c ← 0;
if c ≠ 0 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 ≠ "_");
  if loc > id_first + 1 then
    begin c ← identifier; id_loc ← loc;
    end;
  end
else c ← "E"; {exponent of a real constant }
end
```

This code is used in section 145.

149. **A** string that starts and ends with double-quote marks is converted into an identifier that behaves like a numeric macro by means of the following piece of the program.

```
⟨ Get a preprocessed string 149 ⟩ ≡
  begin double_chars ← 0; id_first ← loc − 1;
  repeat d ← buffer[loc]; incr(loc);
    if (d = """") ∨ (d = "@") then
      if buffer[loc] = d then
        begin incr(loc); d ← 0; incr(double_chars);
        end
      else begin if d = "@" then err_print('!␣Double␣@␣sign␣missing')
        end
    else if loc > limit then
        begin err_print('!␣String␣constant␣didn''t␣end'); d ← """";
        end;
  until d = """";
  id_loc ← loc − 1; c ← identifier;
  end
```

This code is used in section 145.

150. 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 150 ⟩ ≡
  begin c ← control_code(buffer[loc]); incr(loc);
  if c = ignore then goto restart
  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 151 ⟩
      else if c = control_text then
          begin repeat c ← skip_ahead;
          until c ≠ "@";
          if buffer[loc − 1] ≠ ">" then err_print('!␣Improper␣@␣within␣control␣text');
          goto restart;
          end;
  end
```

This code is used in section 145.

151. ⟨ Scan the module name and make *cur_module* point to it 151 ⟩ ≡

```
  begin ⟨ Put module name into mod_text[1 . . k] 153 ⟩;
  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);
  end
```

This code is used in section 150.

152. 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] ← "␣";
```

158. ⟨ Set *accumulator* to the value **of** the right-hand side 158 ⟩ ≡

```
accumulator ← 0; next-sign ← +1;
loop begin next-control ← get-next;
reswitch: case next-control of
  digits: begin ⟨ Set val to value of decimal constant, and set next-control to the following token 160⟩;
    add-in (val); goto reswitch;
    end;
  octal: begin ⟨ Set val to value of octal constant, and set next-control to the following token 161⟩;
    add-in (val); goto reswitch;
    end;
  hex: begin ⟨ Set val to value of hexadecimal constant, and set next-control to the following token 162 ⟩;
    add-in (val); goto reswitch;
    end;
  identifier : begin q ← id-lookup (normal);
    if ilk[q] ≠ numeric then
      begin next-control ← "*"; goto reswitch;   { leads to error }
      end;
    add-in (equiv [q] − '100000);
    end;
  "+": do-nothing;
  "-": next-sign ← -next-sign;
  format, definition, module-name, begin-Pascal, new-module: goto done;
  ";": err-print ('!␣Omit␣semicolon␣in␣numeric␣definition');
  othercases ⟨ Signal error, flush rest of the definition 159⟩
  endcases;
  end;
done:
```

This code is used in section 157.

159. ⟨ Signal error, flush rest of the definition 159⟩ ≡

```
begin err-print ('!␣Improper␣numeric␣definition␣will␣be␣flushed');
repeat next-control ← skip-ahead
until end_of_definition (next-control);
if next-control = module-name then
  begin   {we want to scan the module name too }
  loc ← loc − 2; next-control ← get-next;
  end;
accumulator ← 0; goto done;
end
```

This code is used in section 158.

160. ⟨ Set *val* to value of decimal constant, and set *next-control* to the following token 160⟩ ≡

```
val ← 0;
repeat val ← 10 * val + next-control − "0"; next-control ← get-next;
until (next-control > "9") ∨ (next-control < "0")
```

This code is used in section 158.

161. ⟨ Set *val* to value of octal constant, and set *next-control* to the following token 161⟩ ≡

```
val ← 0; next-control ← "0";
repeat val ← 8 * val + next_control − "0"; next-control ← get-next;
until (next-control > "7") ∨ (next-control < "0")
```

This code is used in section 158.

156. Scanning a numeric definition. When **TANGLE** looks at the PASCAL text following the '=' of a numeric macro definition, it calls on the precedure ***scan-numeric(p),*** where ***p*** points to the name that **is to** be defined. This procedure evaluates the right-hand side, which must consist entirely of integer constants and defined numeric macros connected with + and - signs (no parentheses). It also sets the global variable *next_control* to the control code that terminated this definition.

A definition ends with the control codes ***definition, format, module-name, begin_pascal***, and new-module, all of which can be recognized by the fact that they are the largest values ***get_next*** can return.

```
  define end_of_definition(#) ≡ (# ≥ fotmut)  { is # a control code ending a definition? }
⟨Globals in the outer block 9⟩ +≡
next-control: eight-bitts;  {control code waiting to be acted upon }
```

157. The evaluation of a numeric expression makes use of two variables called the ***accumulator*** and the ***next-sign.*** At the beginning, ***accumulator*** is zero and ***next-sign*** is +1. When a + or - is scanned, ***next-sign*** is multiplied by the value of that sign. When a numeric value is scanned, it is multiplied by ***next-sign*** and added to the ***accumulator,*** then ***next-aign*** is reset to +1.

```
  define add-in (#) ≡
          begin accumulator ← accumulator + next-sign * (#); next-sign ← +1;
          end
procedure scan-numeric (p : name_pointer);  { defines numeric macros}
  label reswitch, done;
  var accumulator : integer ;  { accumulates sums }
    next-sign: -1 . . +1;  {sign to attach to next value }
    q: name-pointer ;  { points to identifiers being evaluated }
    vul: integer ;  { constants being evaluated }
  begin ( Set accumulator to the value of the right-hand side 158 );
  if uba (accumulator) ≥ '100000 then
    begin err-print ( '!␣Value␣too␣big:␣', uccumulutor : 1); accumulator ← 0;
    end;
  equiv [p] ← accumulator + '100000;  { name p now is defined to equal accumulator }
  end;
```

153. ⟨Put module name into *mod-text* [1 . . *k*] 153⟩ ≡

```
k ← 0;
loop begin if loc > limit then
      begin get-line;
      if input-has-ended then
        begin err-print ('!␣Input␣ended␣in␣section␣name'); goto done;
        end;
      end;
    d ← buffer [loc]; ⟨If end of name, goto done 154⟩;
    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 155⟩;
  if (mod-text [k] = "␣") ∧ (k > 0) then decr (k);
```

This code is used in section 151.

154. ⟨If end of name, goto *done* 154⟩ ≡

```
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 153.

155. ⟨Check for overlong name 155⟩ ≡

```
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 153.

162. ⟨ Set *val* to value of hexadecimal constant, and set *next-control* to the following token **162** ⟩ ≡

```
val ← 0; next-control ← "0";
repeat if next-control ≥ "A" then next-control ← next-control + "0" + 10 − "A";
  val ← 16 * val + next-control − "0"; next_control ← get_next;
until (next-control > "F") ∨ (next-control < "0") ∨ ((next-control > "9") A (next-control < "A"))
```

This code is used in section 158.

163. Scanning a macro definition. The rules for generating the replacement texts corresponding to simple macros, parametric macros, and **PASCAL** texts of a module are almost identical, so a single procedure is used for all three cases. The differences are that

a) The sign **#** denotes a parameter only when it appears outside of strings in a parametric macro; otherwise it stands for the ASCII character #. (This is not used in standard **PASCAL,** but some PASCALs **allow,** for example, '/#' after a certain kind of file name.)

b) Module names are not allowed in simple macros or parametric macros; in fact, the appearance of a module name terminates such macros and denotes the name of the current module.

c) The symbols @d and Of and @p are not allowed after module names, while they terminate macro definitions.

164. Therefore there is a procedure ***scan_repl*** whose parameter ***t*** specifies either **simple or *parametric* or *module-name.*** After ***scan_repl*** has acted, ***cur-repl-text*** will point to the replacement text just generated, and ***next-control*** will contain the control code that terminated the activity.

⟨ Globals in the outer block 9 ⟩ +≡
cur-repl-text : text-pointer ; { replacement text formed by ***scan_repl*** }

165.

```
procedure scan_repl (t : eight-bits); { creates a replacement text }
  label continue, done, found;
  var a: sixteen-bits; { the current token }
    b: ASCII-code ; { a character from the buffer }
    bal: eight-bits ; { left parentheses minus right parentheses }
  begin bal ← 0;
  loop begin continue: a ← get-next;
    case a of
    "(": incr (bal);
    ")": if bul = 0 then err-print ( . !␣Extra␣) ')
      else decr (bal);
    "'": ⟨ copy a string from the buffer-to tok-mem 168⟩;
    "#": if t = parametric then a ← param;
    ⟨ In cases that a is a non-ASCII token (identifier, module-name, etc.), either process it and change a to
          a byte that should be stored, or goto continue if a should be ignored, or goto done if a signals
          the end of this replacement text 167 ⟩
    othercases do-nothing
    endcases;
    app_repl (a); {store a in tok-mem }
    end;
 done: next-control ← a; ⟨Make sure the parentheses balance 166⟩;
  if text-ptr > max_texts − zz then overflow ('text ');
  cur_repl_text ← text-ptr ; tok_start [text-ptr + zz] ← tok_ptr [z]; incr (textqtr);
  if z = zz − 1 then z ← 0 else incr (z);
  end:
```

166. ⟨Make sure the parentheses balance 166⟩ ≡
```
if bal > 0 then
  begin if bal = 1 then err_print('!␣Missing␣)')
  else err_print('!␣Missing␣', bal:1, '␣)''s');
  while bal > 0 do
    begin app_repl(")"); decr(bal);
    end;
  end
```
This code is used in section 165.

167. ⟨In cases that *a* is a non-ASCII token (*identifier*, *module_name*, etc.), either process it and change *a* to a byte that should be stored, or **goto** *continue* if *a* should be ignored, or **goto** *done* if *a* signals the end of this replacement text 167⟩ ≡
```
identifier: begin a ← id_lookup(normal); app_repl((a div '400) + '200); a ← a mod '400;
  end;
module_name: if t ≠ module_name then goto done
  else begin app_repl((cur_module div '400) + '250); a ← cur_module mod '400;
    end;
verbatim: ⟨Copy verbatim string from the buffer to tok_mem 169⟩;
definition, format, begin_pascal: if t ≠ module_name then goto done
  else begin err_print('!␣@', xchr[buffer[loc − 1]], '␣is␣ignored␣in␣PASCAL␣text'); goto continue;
    end;
new_module: goto done;
```
This code is used in section 165.

168. ⟨Copy a string from the buffer to *tok_mem* 168⟩ ≡
```
begin b ← "'";
  loop begin app_repl(b);
    if b = "@" then
      if buffer[loc] = "@" then incr(loc) {store only one @}
      else err_print('!␣You␣should␣double␣@␣signs␣in␣strings');
    if loc = limit then
      begin err_print('!␣String␣didn''t␣end'); buffer[loc] ← "'"; buffer[loc + 1] ← 0;
      end;
    b ← buffer[loc]; incr(loc);
    if b = "'" then
      begin if buffer[loc] ≠ "'" then goto found
      else begin incr(loc); app_repl("'");
        end;
      end;
    end;
found: end {now a holds the final "'" that will be stored}
```
This code is used in section 165.

169. ⟨ Copy verbatim string from the buffer to *tok_mem* 169 ⟩ ≡

```
begin app_repl(verbatim); buffer[limit + 1] ← "@";
while buffer[loc] ≠ "@" do
  begin app_repl(buffer[loc]); incr(loc);
  if loc < limit then
    if (buffer[loc] = "@") ∧ (buffer[loc + 1] = "@") then
      begin app_repl("@"); loc ← loc + 2;
      end;
  end;
if loc ≥ limit then err_print('!␣Verbatim␣string␣didn''t␣end')
else if buffer[loc + 1] ≠ ">" then err_print('!␣You␣should␣double␣@␣signs␣in␣verbatim␣strings');
loc ← loc + 2;
end {another verbatim byte will be stored, since a = verbatim}
```

This code is used in section 167.

170. The following procedure is used to define a simple or parametric macro, just after the '==' of its definition has been scanned.

```
procedure define_macro(t : eight_bits);
  var p: name_pointer; { the identifier being defined }
  begin p ← id_lookup(t); scan_repl(t);
  equiv[p] ← cur_repl_text; text_link[cur_repl_text] ← 0;
  end;
```

171. Scanning a module. The *scan_module* procedure starts when '@␣' or '@*' has been sensed in the input, and it proceeds until the end of that module. It uses *module_count* to keep track of the current module number; with luck, WEAVE and TANGLE will both assign the same numbers to modules.

```
⟨ Globals in the outer block 9 ⟩ +≡
module_count : 0 . . 27777;  { the current module number }
```

172. The top level of *scan_module* is trivial.

```
procedure scan_module;
  label continue, done, exit;
  var p: name_pointer ;  { module name for the current module }
  begin incr (module_count); ⟨ Scan the definition part of the current module 173 ⟩;
  ⟨ Scan the PASCAL part of the current module 175 ⟩;
exit: end;
```

173. ⟨ Scan the definition part of the current module 173 ⟩ ≡

```
  next_control ← 0;
  loop begin continue: while next_control ≤ format do
      begin next_control ← skip_ahead;
      if next_control = module_name then
        begin  {we want to scan the module name too}
        loc ← loc − 2; next_control ← get_next;
        end;
      end;
    if next_control ≠ definition then goto done;
    next_control ← get_next ;  { get identifier name }
    if next_control ≠ identifier then
      begin err_print ('!␣Definition␣flushed,␣must␣start␣with␣', 'identifier␣of␣length␣>␣1');
      goto continue ;
      end;
    next_control ← get_next;  { get token after the identifier }
    if next_control = "=" then
      begin scan_numeric (id_lookup (numeric)); goto continue ;
      end
    else if next_control = equivalence_sign then
        begin define_macro (simple); goto continue ;
        end
      else ⟨ If the next text is '(#) ≡', call define_macro and goto continue 174 ⟩;
    err_print ('!␣Definition␣flushed␣since␣it␣starts␣badly');
    end;
done:
```

This code is used in section 172.

174. ⟨ If the next text is '(#) ≡', call *define-macro* and **goto** *continue* 174 ⟩ ≡

```
if next-control = "(" then
  begin next-control ← get-next;
  if next-control = "#" then
    begin next-control ← get-next;
    if next-control = ")" then
      begin next-control ← get-next;
      if next-control = "=" then
        begin err-print('!␣Use␣==␣for␣macros'); next-control ← equivalence-sign;
        end;
      if next-control = equivalence-sign then
        begin define-macro(parametric); goto continue;
        end;
      end;
    end;
  end;
```

This code is used in section 173.

175. ⟨ Scan the **PASCAL** part, of the current module 175 ⟩ ≡

```
case next-control of
begin-Pascal: p ← 0;
module_name: begin p ← cur-module;
  ⟨ Check that = or ≡ follows this module name, otherwise return 176 ⟩;
  end;
othercases return
endcases;
⟨Insert the module number into tok-mem 177⟩;
scan-repl(module-name); { now cur_repl_text points to the replacement text }
⟨Update the data structure so that the replacement text is accessible 178⟩;
```

This code is used in section 172.

176. ⟨Check that = or ≡ follows this module name, otherwise **return** 176⟩ ≡

```
repeat next-control ← get-next;
until next-control ≠ "+"; {allow optional '+='}
if (next-control ≠ "=") A (next-control ≠ equivalence-sign) then
  begin err-print('!␣PASCAL␣text␣flushed,␣=␣sign␣is␣missing');
  repeat next-control ← skip-ahead;
  until next-control = new-module;
  return;
  end
```

This code is used in section 175.

177. ⟨ Insert the module number into *tok_mem* 177 ⟩ ≡

```
store_two_bytes('150000 + module-count); { '150000 = 320 * '400 }
```

This code is used in section 175.

178. ⟨ Update the data structure so that the replacement text is accessible 178 ⟩ ≡

```
if p = 0 then { unnamed module}
  begin text-link [lust-unnamed] ← cur-repl-text ; lust-unnamed ← cur-repl-text ;
  end
else if equiv [p] = 0 then equiu [p] ← cur-repl-text   { first module of this name }
  else begin p ← equiv [p];
    while text-link [p] < module_flag do p ← text-link [p];   { find end of list }
    text-link [p] ← cur-repl-text ;
    end;
text-link [cur-repl-text] ← module-flag;   { mark this replacement text as a nonmacro }
```

This code is used in section 175.

182. The main program. We have defined plenty of procedures, and it is time to put the last pieces of the puzzle in place. Here is where TANGLE starts, and where it ends.

```
  begin initialize; ⟨ Initialize the input system 134 ⟩;
  print_ln (banner);   { print a “banner line” }
  ⟨Phase I: Read all the user's text and compress it into tok-mem 183 ⟩;
  stat for zo ← 0 to zz − 1 do max_tok_ptr [zo] ← tok_ptr [zo];
  tats
  ⟨Phase II: Output the contents of the compressed tables 112 ⟩;
end_of_TANGLE: if string-ptr > 128 then ⟨ Finish off the string pool file 184⟩;
  stat ⟨Print statistics about memory usage 180⟩; tats
{here files should be closed if the operating system requires it }
  ⟨Print the job history 187⟩;
  end.
```

183. ⟨Phase I: Read all the user's text and compress it into *tok-mem* 183⟩ ≡

```
  phase-one ← true; module-count ← 0;
  repeat next-control ← skip-ahead;
  until next-control = new-module;
  while ¬input_has_ended do scan-module;
  ⟨ Check that all changes have been read 138 ⟩;
  phase-one ← false;
```

This code is used in section 182.

184. ⟨ Finish off the string pool file 184 ⟩ ≡

```
  begin print_nl (string-ptr − 128 : 1, '␣strings␣written␣to␣string␣pool␣file.'); write(pool, '*');
  for string-ptr ← 1 to 9 do
    begin out-buf [string-ptr] t pool-check-sum mod 10; pool-check-sum ← pool-check-sum div 10;
    end;
  for string-ptr t 9 downto 1 do write (pool, xchr ["0" + out-buf [string-ptr]]);
  write-Zn(pool);
  end
```

This code is used in section 182.

185. ⟨ Globals in the outer block 9⟩ +≡

```
  stat wo: 0 .. ww − 1;   { segment of memory for which statistics are being printed }
  tats
```

186. ⟨ Print statistics about memory usage 186 ⟩ ≡

```
  print-nZ('Memory␣usage␣statistics:');
  print_nl (name-ptr : 1, '␣names,␣', text-ptr : 1, '␣replacement␣texts;'); print_nl (byte-ptr [0] : 1);
  for wo ← 1 to ww −− 1 do print ('+ ●    , byte-ptr [wo] : 1);
  print ('␣bytes,␣', max-tok-ptr [0] : 1);
  for zo ← 1 to zz − 1 do print ('+ ●    , max-tok-ptr [zo] : 1);
  print ('␣tokens.');
```

This code is used in section 182.

```
181.  define breakpoint = 888  {place where a breakpoint is desirable }
  debug procedure debug-help;  {routine to display various things }
  label breakpoint, exit;
  var k: sixteen-bits ;  { index into various arrays }
  begin incr (debug-skipped);
  if debug-skipped < debug-cycle then return;
  debug-skipped ← 0;
  loop begin write (term-out, '#'); update-terminal;  { prompt }
    read (term-in, ddt ); . {read a list of integers }
    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_repl (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 print (xchr [out-contrib [k]]);
        othercases print ( '? ')
        endcases;
        end;
    end;
exit: end;
  gubed
```

179. Debugging. The PASCAL debugger with which TANGLE 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* ≤ 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 you 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.

⟨ Globals in the outer block 9 ⟩ +≡

```
  debug trouble-shooting: boolean ;  { is debug-help wanted? }
ddt : sixteen-bits ;  {operation code for the debug-help routine }
dd: sixteen-bits ;  { 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 }
term-in: text_file ;  { the user's terminal as an input file }
  gubed
```

180. 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;
  trouble-shooting ← false; debug-cycle ← 99999;  { use these when it almost works }
  reset (term-in, 'TTY :', '/I ');  { open term-in as the terminal, don't do a get }
  gubed
```

187. 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 187⟩ ≡
  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 182.

188. System-dependent changes. This module should be replaced, if necessary, by changes to the program that are necessary to make TANGLE work at a particular installation. It is usually best to design your change file so that all changes to previous modules preserve the module numbering; then everybody's version will be consistent with the printed program. More extensive changes, which introduce new modules, can be inserted here; then only the index itself will get a new module number.

189. Index. Here is a cross-reference table for the TANGLE processor. All modules in which an identifier is used are listed with that identifier, except that reserved words arc 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 and a few other things like "ASCII code" are indexed here too.

⟨ Initialize the input system 134 ⟩ used in section 182.
⟨ Initialize the output buffer 96 ⟩ Used in section 112.
⟨Initialize the output stacks 83 ⟩ Used in section 112.
⟨ Insert the module number into *tok_mem* 177 ⟩ Used in section 175.
⟨Local variables for initialization 16, **41**, 45, **51** ⟩ Used in section 2.
⟨ Make sure the parentheses balance 166⟩ Used in section 165.
⟨Move *buffer* and ***limit*** to *change_buffer* and ***change-limit*** 131⟩ Used in sections 128 and 132.
⟨ Other printable characters **115** ⟩ Used in section 113.
⟨Phase I: Read all the user's text and compress it into tok-mem 183 ⟩ Used in section 182.
⟨Phase II: Output the contents of the compressed tables 112 ⟩ Used in section 182.
⟨Print error location based on input buffer 32 ⟩ Used in section 31.
⟨Print error location based on output buffer 33 ⟩ Used in section 31.
⟨Print statistics about memory usage 186⟩ Used in section 182.
⟨Print the job ***history*** 187 ⟩ Used in section 182.
⟨Put a parameter on the parameter stack, or **goto** ***restart*** if error occurs **90**⟩ Used in section 89.
⟨ Put module name into ***mod-text*** [1 . . *k*] 153 ⟩ Used in section 151.
⟨Read from ***change-file*** and maybe turn ***off changing*** 137⟩ Used in section 135.
⟨ Read from *web_file* and maybe turn on ***changing*** 136 ⟩ Used in section 135.
⟨Reduce *sign_val_val* to *sign_val* and **goto** ***restart*** 104⟩ Used in section 102.
⟨Remove a parameter from the parameter stack 91⟩ Used in section 85.
⟨ Remove ***p*** from secondary hash table 60 ⟩ Used in section 59.
⟨ Scan the definition part of the current module **173** ⟩ Used in section 172.
⟨ Scan the module name and make ***cur-module*** point to it **151**⟩ used in section 150.
⟨ Scan the PASCAL part of the current module 175⟩ Used in section 172.
⟨Send a string, **goto** *reswitch* 117⟩ Used in section 113.
⟨ Send verbatim string 118⟩ Used in section 113.
⟨ Set ***accumulator*** to the value of the right-hand side 158⟩ Used in section 157.
⟨ Set c to the result of comparing the given name to name ***p*** 68 ⟩ Used in sections 66 and 69.
⟨ Set initial values 10, 14, 17, 18, 21, 26, 42, **46**, 48, 52, 71, 144, 152, 180 ⟩ Used in section 2.
⟨ Set *val* to value of decimal constant, and set ***next-control*** to the following token 160⟩ Used in section 158.
⟨ Set *val* to value of hexadecimal constant, and set ***next-control*** to the following token 162⟩ Used in section 158.
⟨ Set *val* to value of octal constant, and set ***next-control*** to the following token 161⟩ Used in section 158.
⟨ Signal error, flush rest of the definition **159** ⟩ Used in section 158.
⟨ Skip over comment lines in the change file; return if end of file 129 ⟩ Used in section 128.
⟨ Skip to the next nonblank line; return if end of file 130 ⟩ Used in section 128.
⟨ Special code to finish real constants **120** ⟩ Used in section **113.**
⟨ Start scanning current macro parameter, **goto** ***restart*** 92⟩ Used in section 87.
⟨ Types in the outer block 11, 12, 37, 39, 43, 78 ⟩ Used in section 2.
⟨Update the data structure so that the replacement text is accessible 178⟩ Used in section 175.
⟨Update the tables and check for possible errors 57 ⟩ Used in section 53.

Appendix F: The webmac . tex file. This is the file that extends "plain TEX" format in order to support the features needed by the output of WEAVE.

```
% standard macros for WEB listings (in addition to PLAIN.TEX)
\parskip Opt % no stretch between paragraphs
\parindent lem % for paragraphs and for the first line of PASCAL text

\font\eightrm=cmr8
\let\sc=\eightrm \let\mainfont=\tenrm
\font\titlefont=cmr7 scaled\magstep4 % title on the contents page
\font\ttitlefont=cmtt10 scaled\magstep2 % typewriter type in title
\font\tentex=cmtex10 % TeX extended character set (used in strings)

\def\\#1{\hbox{\it#1\/\kern.05em}} % italic type for identifiers
\def\|#1{\hbox{$#1$}} % one-letter identifiers look a bit better this way
\def\&#1{\hbox{\bf#1\/}} % boldface type for reserved words
\def\.#1{\hbox{\tentex % typewriter type for strings
  \let\\=\BS % backslash in a string
  \let\'=\RQ % right quote in a string
  \let\`=\LQ % left quote in a string
  \let\{=\LB % left brace in a string
  \let\}=\RB % right brace in a string
  \let\~=\TL % tilde in a string
  \let\ =\SP % space in a string
  \let\_=\UL % underline in a string
  \let\&=\AM % ampersand in a string
 .#1}}
\def\#{\hbox{\tt\char`\#}} % parameter sign
\def\${\hbox{\tt\char`\$}} % dollar sign
\def\%{\hbox{\tt\char`\%}} % percent sign
\def\^{\ifmmode\mathchar"222 \else\char`^ \fi} % pointer or hat
% circumflex accents can be obtained from \^^D instead of \^
\def\AT!{@} % at sign for control text

\chardef\AM=`\& % ampersand character in a string
\chardef\BS=`\\ % backslash in a string
\chardef\LB=`\{ % left brace in a string
\def\LQ{{\tt\char'22}} % left quote in a string
\chardef\RB=`\} % right brace in a string
\def\RQ{{\tt\char'23}} % right quote in a string
\def\SP{{\tt\char`\ }} % (visible) space in a string
\chardef\TL=`\~ % tilde in a string
\chardef\UL=`\_ % underline character in a string

\newbox\bak \setbox\bak=\hbox to -1em{} % backspace one em
\newbox\bakk\setbox\bakk=\hbox to -2em{} % backspace two ems

\newcount\ind % current indentation in ems
\def\l{\global\advance\ind by1\hangindent\ind em) % indent one more notch
\def\2{\global\advance\ind by-l) % indent one less notch
\def\3#1{\hfil\penalty#10\hfilneg} % optional break within a statement
\def\4{\copy\bak} % backspace one notch
\def\5{\hfil\penalty-1\hfilneg\kern2.5em\copy\bakk\ignorespaces}%optionalbreak
```

```
\def\6{\ifmmode\else\par % forced break
  \hangindent\ind em\noindent\kern\ind em\copy\bakk\ignorespaces\fi}
\def\7{\Y\6} % forced break and a little extra space

\let\yskip=\smallskip
\def\to{\mathrel{.\,.}} % double dot, used only in math mode
\def\note#1#2.{\Y\noindent{\hangindent2em\baselineskip10pt\eightrm#1 #2.\par}}
\def\lapstar{\rlap{*}}
\def\startsection{\Q\noindent{\let\*=\lapstar\bf\modstar.\quad}}
\def\defin#1{\global\advance\ind by 2 \1\&{#1 }} % begin 'define' or 'format'
\def\A{\note{See also}} % cross-reference for multiply defined section names
\def\B{\mathopen{\.{@\{}}} % begin controlled comment
\def\C#1{\ifmmode\gdef\XX{\null$\null}\else\gdef\XX{}\fi % PASCAL comments
  \XX\hfil\penalty-1\hfilneg\quad$\{\,$#1$\,\}$\XX}
\def\D{\defin{define}} % macro definition
\def\E{\cdot10^} % exponent in floating point constant
\def\F{\defin{format}} % format definition
\let\G=\ge % greater than or equal sign
\def\H#1{\hbox{\rm\char"7D\tt#1}} % hexadecimal constant
\let\I=\ne % unequal sign
\def\J{\.{@\&}} % TANGLE's join operation
\let\K=\gets % left arrow
\let\L=\le % less than or equal sign
\outer\def\M#1.{\MN#1.\ifon\vfil\penalty-100\vfilneg % beginning of section
  \vskip12ptminus3pt\startsection\ignorespaces}
\outer\def\N#1.#2.{\MN#1.\vfil\eject % beginning of starred section
  \def\rhead{\uppercase{\ignorespaces#2}} % define running headline
  \message{*\modno} % progress report
  \edef\next{\write\cont{\Z{#2}{\modno}{\the\pageno}}}\next % to contents file
  \ifon\startsection{\bf\ignorespaces#2.\quad}\ignorespaces}
\def\MN#1.{\par % common code for \M, \N
  {\xdef\modstar{#1}\let\*=\empty\xdef\modno{#1}}
  \ifx\modno\modstar \onmaybe \else\ontrue \fi \mark{\modno}}
\def\O#1{\hbox{\rm\char'23\kern-.2em\it#1\/\kern.05em}} % octal constant
\def\P{\rightskip=0pt plus 100pt minus 10pt % go into PASCAL mode
  \sfcode`;=3000
  \pretolerance 10000
  \hyphenpenalty 10000 \exhyphenpenalty 10000
  \global\ind=2 \1\ \unskip}
\def\Q{\rightskip=0pt % get out of PASCAL mode
  \sfcode`;=1500 \pretolerance 200 \hyphenpenalty 50 \exhyphenpenalty 50 }
\let\R=\lnot % logical not
\let\S=\equiv % equivalence sign
\def\T{\mathclose{\.{@\}}}} % terminate controlled comment
\def\U{\note{This code is used in}} % cross-reference for uses of sections
\let\V=\lor % logical or
\let\W=\land % logical and
\def\X#1:#2\X{\ifmmode\gdef\XX{\null$\null}\else\gdef\XX{}\fi % section name
  \XX$\langle\,$#2{\eightrm\kern.5em#1}$\,\rangle$\XX}
\def\Y{\par\yskip}
\let\Z=\let % now you can \send the control sequence \Z
\def\){\hbox{\.{@\$}}} % sign for string pool check sum
```

```
\def\]{\hbox{\.{@\\}}} % sign for forced line break
\def\=#1{\kern2pt\hbox{\vrule\vtop{\vbox{\hrule
        \hbox{\strut\kern2pt\.{#1}\kern2pt}}
      \hrule}\vrule}\kern2pt} % verbatim string
\let\~=\ignorespaces
\let\*=*

\def\onmaybe{\let\ifon=\maybe} \let\maybe=\iftrue
\newif\ifon  \newif\iftitle  \newif\ifpagesaved
\def\lheader{\mainfont\the\pageno\eightrm\qquad\rhead\hfill\title\qquad
  \tensy x\mainfont\topmark} % top line on left-hand pages
\def\rheader{\tensy x\mainfont\topmark\eightrm\qquad\title\hfill\rhead
  \qquad\mainfont\the\pageno} % top line on right-hand pages
\def\page{\box255 }
\def\normaloutput#1#2#3{\shipout\vbox{
  \ifodd\pageno\hoffset=\pageshift\fi
  \vbox to\fullpageheight{
  \iftitle\global\titlefalse
  \else\hbox to\pagewidth{\vbox to10pt{}\ifodd\pageno #3\else#2\fi}\fi
  \vfill#1}} % parameter #1 is the page itself
  \global\advance\pageno by1}

\def\rhead{\.{WEB} OUTPUT} % this running head is reset by starred sections
\def\title{} % an optional title can be set by the user
\def\topofcontents{\centerline{\titlefont\title}
  \vfill} % this material will start the table of contents page
\def\botofcontents{\vfill} % this material will end the table of contents page
\def\contentspagenumber{0} % default page number for table of contents
\newdimen\pagewidth \pagewidth=6.5in % the width of each page
\newdimen\pageheight \pageheight=8.7in % the height of each page
\newdimen\fullpageheight \fullpageheight=9in % page height including headlines
\newdimen\pageshift \pageshift=0in % shift righthand pages wrt lefthand ones
\def\magnify#1{\mag=#1\pagewidth=6.5truein\pageheight=8.7truein
  \fullpageheight=9truein}
\def\setpage{\hsize\pagewidth\vsize\pageheight} % use after changing page size
\def\contentsfile{CONTENTS} % file that gets table of contents info
\def\readcontents{\input CONTENTS}

\newwrite\cont
\output{\setbox0=\page % the first page is garbage
  \openout\cont=\contentsfile
  \global\output{\normaloutput\page\lheader\rheader}}
\setpage
\vbox to \vsize{} % the first \topmark won't be null

\def\ch{\note{The following sections were changed by the change file:}
  \let\*=\relax}
\newbox\sbox % saved box preceding the index
\newbox\lbox % lefthand column in the index
\def\inx{\par\vskip6pt plus 1fil % we are beginning the index
  \write\cont{} % ensure that the contents file isn't empty
  \closeout\cont % the contents information has been fully gathered
  \output{\ifpagesaved\normaloutput{\box\sbox}\lheader\rheader\fi
    \global\setbox\sbox=\page   \global\pagesavedtrue}
  \pagesavedfalse \eject % eject the page-so-far and predecessors
  \setbox\sbox\vbox{\unvbox\sbox} % take it out of its box
  \vsize=\pageheight \advance\vsize by -\ht\sbox % the remaining height
  \hsize=.5\pagewidth \advance\hsize by -10pt
    % column width for the index (20pt between cols)
  \parfillskip 0pt plus .6\hsize % try to avoid almost empty lines
  \def\lr{L} % this tells whether the left or right column is next
  \output{\if L\lr\global\setbox\lbox=\page \gdef\lr{R}
    \else\normaloutput{\vbox to\pageheight{\box\sbox\vss
        \hbox to\pagewidth{\box\lbox\hfil\page}}}\lheader\rheader
    \global\vsize\pageheight\gdef\lr{L}\global\pagesavedfalse\fi}
  \message{Index:}
  \parskip 0pt plus .5pt
  \outer\def\:##1, {\par\hangindent2em\noindent##1:\kern1em}% index entry
  \def\[##1]{$\underline{##1}$} % underlined index item
  \rm \rightskip0pt plus 2.5em \tolerance 10000 \let\*=\lapstar
  \hyphenpenalty 10000 \parindent0pt}
\def\fin{\par\vfill\eject % this is done when we are ending the index
  \ifpagesaved\null\vfill\eject\fi % output a null index column
  \if L\lr\else\null\vfill\eject\fi % finish the current page
  \parfillskip 0pt plus 1fil
  \def\rhead{NAMES OF THE SECTIONS}
  \message{Section names:}
  \output{\normaloutput\page\lheader\rheader}
  \setpage
  \def\note##1##2.{\quad{\eightrm##1 ##2.}}
  \def\U{\note{Used in}} % cross-reference for uses of sections
  \def\:{\par\hangindent 2em}\let\*=*}
\def\con{\par\vfill\eject % finish the section names
  \rightskip 0pt \hyphenpenalty 50 \tolerance 200
  \setpage
  \output{\normaloutput\page\lheader\rheader}
  \titletrue % prepare to output the table of contents
  \pageno=\contentspagenumber \def\rhead{TABLE OF CONTENTS}
  \message{Table of contents:}
  \topofcontents
  \line{\hfil Section\hbox to3em{\hss Page}}
  \def\Z##1##2##3{\line{\ignorespaces##1
    \leaders\hbox to .5em{.\hfil}\hfil\ ##2\hbox to3em{\hss##3}}}
  \readcontents\relax % read the contents info
  \botofcontents \end} % print the contents page(s) and terminate
```

Appendix G: How to use WEB macros. The macros in webmac make it possible to produce a variety of formats without editing the output of WEAVE, and the purpose of this appendix is to explain some of the possibilities.

1. Three fonts have been declared in addition to the standard fonts of PLAIN format: You can say '{\sc stuff}' to get STUFF in small caps; and you can select the largish fonts \titlefont and \ttitlefont in the title of your document, where \ttitlefont is a typewriter style of type.

2. When you mention an identifier in TEX text, you normally call it '|identifier|'. But you can also say '\\{identifier}'. The output will look the same in both cases, but the second alternative doesn't put identifier into the index, since it bypasses WEAVE's translation from PASCAL mode.

3. To get typewriter-like type, as when referring to 'WEB', you can use the '\.' macro (e.g., '\.{WEB}'). In the argument to this macro you should insert an additional backslash before the symbols listed as 'special string characters' in the index to WEAVE, i.e., before backslashes and dollar signs and the like. A '\␣' here will result in the visible space symbol; to get an invisible space following a control sequence you can say '{␣}'.

4. The three control sequences \pagewidth, \pageheight, and \fullpageheight can be redefined in the limbo section at the beginning of your WEB file, to change the dimensions of each page. The standard settings

```
\pagewidth=6.5in
\pageheight=8.7in
\fullpageheight=9in
```

were used to prepare the present report; \fullpageheight is \pageheight plus room for the additional heading and page numbers at the top of each page. If you change any of these quantities, you should call the macro \setpage immediately after making the change.

5. The \pageshift macro defines an amount by which right-hand pages (i.e., odd-numbered pages) are shifted right with respect to left-hand (even-numbered) ones. By adjusting this amount you may be able to get two-sided output in which the page numbers line up on opposite sides of each sheet.

6. The \title macro will appear at the top of each page in small caps. For example, Appendix D was produced after saying '\def\title{WEAVE}'.

7. The first page usually is number 1; if you want some other starting page, just set \pageno to the desired number. For example, the initial limbo section for Appendix D included the command '\pageno=16'.

8. The macro \iftitle will suppress the header line if it is defined by '\titletrue'. The normal value is \titlefalse except for the table of contents; thus, the contents page is usually unnumbered. If your program is so long that the table of contents doesn't fit on a single page, or if you want a number to appear on the contents page, you should reset \pageno when you begin the table of contents.

Two macros are provided to give flexibility to the table of contents: \topofcontents is invoked just before the contents info is read, and \botofcontents is invoked just after. For example, Appendix D was produced with the following definitions:

```
\def\topofcontents{\null\vfill
  \titlefalse % include headline on the contents page
  \def\rheader{\mainfont Appendix D\hfil 15}
  \centerline{\titlefont The {\ttitlefont WEAVE} processor}
  \vskip 15pt \centerline{(Version 2)} \vfill}
```

Redefining \rheader, which is the headline for right-hand pages, suffices in this case to put the desired information at the top of the page.

9. Data for the table of contents is written to a file that is read after the indexes have been TEXed; there's one line of data for every starred module. For example, when Appendix D was being generated, a file CONTENTS.TEX containing

```
\Z { Introduction}{1}{16}
\Z { The character set}{11}{19}
```

and similar lines was created. The `\topofcontents` macro could redefine `\Z` so that the information appears in another format.

10. Sometimes it is necessary or desirable to divide the output of WEAVE into subfiles that can be processed separately. For example, the listing of TeX runs to more than 500 pages, and that is enough to exceed the capacity of many printing devices and/or their software. When an extremely large job isn't cut into smaller pieces, the entire process might be spoiled by a single error of some sort, making it necessary to start everything over.

Here's a safe way to break a woven file into three parts: Say the pieces are α, β, and γ, where each piece begins with a starred module. All macros should be defined in the opening limbo section of α, and copies of this TeX code should be placed at the beginning of β and of γ. In order to process the parts separately, we need to take care of two things: The starting page numbers of β and γ need to be set up properly, and the table of contents data from all three runs needs to be accumulated.

The webmac macros include two control sequences `\contentsfile` and `\readcontents` that facilitate the necessary processing. We include '`\def\contentsfile{CONT1}`' in the limbo section of α, and we include '`\def\contentsfile{CONT2}`' in the limbo section of β; this causes TeX to write the contents data for α and β into `CONT1.TEX` and `CONT2.TEX`. Now in γ we say

```
\def\readcontents{\input CONT1 \input CONT2 \input CONTENTS};
```

this brings in the data from all three pieces, in the proper order.

However, we still need to solve the page-numbering problem. One way to do it is to include the following in the limbo material for β:

```
\message{Please type the last page number of part 1: }
\read-1to\\ \pageno=\\ \advance\pageno by 1
```

Then you simply provide the necessary data when TeX requests it; a similar construction is used at the beginning of γ.

This method can, of course, be used to divide a woven file into any number of pieces.

11. Sometimes it is nice to include things in the index that are typeset in a special way. For example, we might want to have an index entry for 'TeX'. WEAVE provides only two standard ways to typeset an index entry (unless the entry is an identifier or a reserved word): '`@^`' gives roman type, and '`@.`' gives typewriter type. But if we try to typeset 'TeX' in roman type by saying, e.g., '`@^\TeX@>`', the backslash character gets in the way, and this entry wouldn't appear in the index with the T's.

The solution is to use the '`@:`' feature, declaring a macro that simply removes a sort key as follows:

```
\def\9#1{}
```

Now you can say, e.g., '`@:TeX}{\TeX@>`' in your WEB file; WEAVE puts it into the index alphabetically, based on the sort key, and produces the macro call '`\9{TeX}{\TeX}`' which will ensure that the sort key isn't printed.

A similar idea can be used to insert hidden material into module names so that they are alphabetized in whatever way you might wish. Some people call these tricks "special refinements"; others call them "kludges".

12. The control sequence `\modno` is set to the number of the module being typeset.

13. If you want to list only the modules that have changed, together with the index, put the command '`\let\maybe=\iffalse`' in the limbo section before the first module of your WEB file. It's customary to make this the first change in your change file.

Appendix H: Installing the WEB system. Suppose you want to use the WEB programs on your computer, and suppose that you can't simply borrow them from somebody else who has the same kind of machine. Here's what to do:

(1) Get a tape that contains the files WEAVE.WEB, TANGLE.WEB, TANGLE. PAS, and WEBMAC .TEX. The tape will probably also contain an example change file TANGLE. CH.

(2) Look at the sections of TANGLE that are listed under "system dependencies" in the index of Appendix E above, and figure out what changes (if any) will be needed for your system.

(3) Make a change file TANGLE. CH that contains the changes of (2); do not change your copy of TANGLE. WEB, leave it intact. (The rules for change files are explained at the end of the manual just before the appendices; you may want to look at the example change file that arrived with your copy of TANGLE. WEB. It's also a good idea to define all the "switches" like debug and gubed to be null in your first change files; then you can sure that your compiler will handle all of the code.)

(4) Make the changes of (2) in your copy of TANGLE. PAS. (If these changes are extensive, you might be better off finding some computer that that already has TANGLE running, and making the new TANGLE. PAS from TANGLE. WEB and your TANGLE. CH.)

(5) Use your PASCAL compiler to convert your copy of TANGLE. PAS to a running program TANGLE.

(6) Check your changes as follows: Run TANGLE on TANGLE. WEB and your TANGLE. CH, yielding TANGLE. PAS'; make a running program TANGLE' by applying PASCAL to TANGLE. PAS'; run TANGLE' on TANGLE. WEB and your TANGLE. CH, yielding TANGLE. PAS"; and check that TANGLE. PAS" is identical to TANGLE. PAS'. Once this test has been passed, you have got a working TANGLE program.

(7) Make a change file WEAVE. CH analogous to (3), but this time consider the system-dependent parts of WEAVE that are listed in the index to Appendix D.

(8) Run TANGLE on WEAVE. WEB and your WEAVE. CH, obtaining WEAVE. PAS.

(9) Use PASCAL on WEAVE.PAS to make a running WEAVE program.

(10) Run WEAVE on TANGLE. WEB and TANGLE. CH to produce TANGLE. TEX.

(11) Run TEX on TANGLE .TEX, obtaining a listing analogous to Appendix E. This listing will incorporate your changes.

(12) Run WEAVE on WEAVE. WEB and your WEAVE. CH to produce WEAVE. TEX.

(13) Run TEX on WEAVE. TEX, obtaining a listing analogous to Appendix D that incorporates your changes.

This description assumes that you already have a working TEX82 system. But what if you don't have TEX82? Then you start with a tape that also contains TEX . WEB and plain. tex, and you refer to a hardcopy listing of the TEX82 program corresponding to TEX . WEB. Between steps (10) and (11) you do the following:

(10.1) Make a change file TEX. CH to fix the system dependent portions of TEX. WEB, in a manner analogous to step (2). Since TEX is a much more complex program than WEAVE or TANGLE, there are more system-dependent features to think about, but by now you will be good at making such modifications. Do not make any changes to TEX . WEB.

(10.2) Make an almost-copy of your TEX. CH called INITEX. CH; this one will have the 'init' and 'tini' macros redefined in order to make the initialization version of TEX. It also might have smaller font memory and dynamic memory areas, since INITEX doesn't need as much memory for such things; by setting the memory smaller in INITEX, you guarantee that the production system will have a "cushion."

(10.3) Run TANGLE on TEX. WEB and INITEX. CH, obtaining INITEX . PAS and TEX . POOL.

(10.4) Run PASCAL on INITEX. PAS, obtaining INITEX.

(10.5) Run INITEX on TEX. POOL, during which run you type 'plain' and '\dump'. This will produce a file plain. fmt containing the data needed to initialize TEX's memory.

(10.6) Run TANGLE on TEX. WEB and the TEX. CH of (10.1), obtaining TEX. PAS.

(10.7) Run PASCAL on TEX. PAS, obtaining VIRTEX.

(10.8) If your operating system supports programs whose core images have been saved, run VIRTEX, type '&plain', then save the core image and call it TEX. Otherwise, VIRTEX will be your TEX, and it will read 'plain. fmt' (or some other fmt file) each time you run.

This 21-step process may seem long, hut it is actually an oversimplification, since you also need fonts and a way to print the device-independent files that TEX spews out. On the other hand, the total number of steps is not quite so large when you consider that TANGLE-followed-by-PASCAL and WEAVE-followed-by-TEX may be regarded as single operations.

If you have only the present report, not a tape, you will have to prepare files WEAVE. WEB and TANGLE. WEB by hand, typing them into the computer by following Appendices D and E. Then you have to simulate the behavior of TANGLE by converting TANGLE. WEB manually into TANGLE. PAS; with a good text editor this takes about six hours. Then you have to correct errors that were made in all this hand work: but still the whole project is not impossibly difficult, because in fact the entire development of WEAVE and TANGLE (including the writing of the programs and the manual) took less than two months of work.