The TANGLE processor

(Version 4.6)

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␣4.6´

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

$$\begin{aligned}
&\textbf{case } x \textbf{ of}\\
&1: \langle \text{code for } x = 1 \rangle;\\
&3: \langle \text{code for } x = 3 \rangle;\\
&\textbf{othercases } \langle \text{code for } x \neq 1 \text{ and } x \neq 3 \rangle\\
&\textbf{endcases}
\end{aligned}$$

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 }

$max\_toks = 65000;$

   { $1/zz$ 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 $mark\_harmless \equiv$ **if** $history = spotless$ **then** $history \leftarrow harmless\_message$

define $mark\_error \equiv history \leftarrow error\_message$

define $mark\_fatal \equiv history \leftarrow fatal\_message$

⟨ 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 ⟩ ≡

$history \leftarrow spotless;$

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

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

Here is a table of the standard visible ASCII codes:

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

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

⟨ Types in the outer block 11 ⟩ ≡
    $ASCII\_code = 0 \mathinner{\ldotp\ldotp} 255$;   { eight-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_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* = 255   { ordinal number of the largest element of *text_char* }

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

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

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

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

⟨Set initial values 10⟩ +≡
  $xchr[´40] \leftarrow ´\sqcup´;\ xchr[´41] \leftarrow ´!´;\ xchr[´42] \leftarrow ´"´;\ xchr[´43] \leftarrow ´\#´;\ xchr[´44] \leftarrow ´\$´;$
  $xchr[´45] \leftarrow ´\%´;\ xchr[´46] \leftarrow ´\&´;\ xchr[´47] \leftarrow ´´´´;$
  $xchr[´50] \leftarrow ´(´;\ xchr[´51] \leftarrow ´)´;\ xchr[´52] \leftarrow ´*´;\ xchr[´53] \leftarrow ´+´;\ xchr[´54] \leftarrow ´,´;$
  $xchr[´55] \leftarrow ´-´;\ xchr[´56] \leftarrow ´.´;\ xchr[´57] \leftarrow ´/´;$
  $xchr[´60] \leftarrow ´0´;\ xchr[´61] \leftarrow ´1´;\ xchr[´62] \leftarrow ´2´;\ xchr[´63] \leftarrow ´3´;\ xchr[´64] \leftarrow ´4´;$
  $xchr[´65] \leftarrow ´5´;\ xchr[´66] \leftarrow ´6´;\ xchr[´67] \leftarrow ´7´;$
  $xchr[´70] \leftarrow ´8´;\ xchr[´71] \leftarrow ´9´;\ xchr[´72] \leftarrow ´:´;\ xchr[´73] \leftarrow ´;´;\ xchr[´74] \leftarrow ´<´;$
  $xchr[´75] \leftarrow ´=´;\ xchr[´76] \leftarrow ´>´;\ xchr[´77] \leftarrow ´?´;$
  $xchr[´100] \leftarrow ´@´;\ xchr[´101] \leftarrow ´A´;\ xchr[´102] \leftarrow ´B´;\ xchr[´103] \leftarrow ´C´;\ xchr[´104] \leftarrow ´D´;$
  $xchr[´105] \leftarrow ´E´;\ xchr[´106] \leftarrow ´F´;\ xchr[´107] \leftarrow ´G´;$
  $xchr[´110] \leftarrow ´H´;\ xchr[´111] \leftarrow ´I´;\ xchr[´112] \leftarrow ´J´;\ xchr[´113] \leftarrow ´K´;\ xchr[´114] \leftarrow ´L´;$
  $xchr[´115] \leftarrow ´M´;\ xchr[´116] \leftarrow ´N´;\ xchr[´117] \leftarrow ´O´;$
  $xchr[´120] \leftarrow ´P´;\ xchr[´121] \leftarrow ´Q´;\ xchr[´122] \leftarrow ´R´;\ xchr[´123] \leftarrow ´S´;\ xchr[´124] \leftarrow ´T´;$
  $xchr[´125] \leftarrow ´U´;\ xchr[´126] \leftarrow ´V´;\ xchr[´127] \leftarrow ´W´;$
  $xchr[´130] \leftarrow ´X´;\ xchr[´131] \leftarrow ´Y´;\ xchr[´132] \leftarrow ´Z´;\ xchr[´133] \leftarrow ´[´;\ xchr[´134] \leftarrow ´\backslash´;$
  $xchr[´135] \leftarrow ´]´;\ xchr[´136] \leftarrow ´\char94´;\ xchr[´137] \leftarrow ´\_´;$
  $xchr[´140] \leftarrow ´\char96´;\ xchr[´141] \leftarrow ´a´;\ xchr[´142] \leftarrow ´b´;\ xchr[´143] \leftarrow ´c´;\ xchr[´144] \leftarrow ´d´;$
  $xchr[´145] \leftarrow ´e´;\ xchr[´146] \leftarrow ´f´;\ xchr[´147] \leftarrow ´g´;$
  $xchr[´150] \leftarrow ´h´;\ xchr[´151] \leftarrow ´i´;\ xchr[´152] \leftarrow ´j´;\ xchr[´153] \leftarrow ´k´;\ xchr[´154] \leftarrow ´l´;$
  $xchr[´155] \leftarrow ´m´;\ xchr[´156] \leftarrow ´n´;\ xchr[´157] \leftarrow ´o´;$
  $xchr[´160] \leftarrow ´p´;\ xchr[´161] \leftarrow ´q´;\ xchr[´162] \leftarrow ´r´;\ xchr[´163] \leftarrow ´s´;\ xchr[´164] \leftarrow ´t´;$
  $xchr[´165] \leftarrow ´u´;\ xchr[´166] \leftarrow ´v´;\ xchr[´167] \leftarrow ´w´;$
  $xchr[´170] \leftarrow ´x´;\ xchr[´171] \leftarrow ´y´;\ xchr[´172] \leftarrow ´z´;\ xchr[´173] \leftarrow ´\{´;\ xchr[´174] \leftarrow ´|´;$
  $xchr[´175] \leftarrow ´\}´;\ xchr[´176] \leftarrow ´\sim´;$
  $xchr[0] \leftarrow ´\sqcup´;\ xchr[´177] \leftarrow ´\sqcup´;$ { these ASCII codes are not used }

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

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

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

⟨ Local variables for initialization 16 ⟩ ≡
i: 0 . . 255;

See also sections 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 if you have, for example, an extended character set like the one in Appendix C of *The T_EXbook*, the first line of code in this module should be changed to

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

WEB's character set is essentially identical to T_EX'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* ← 1 **to** ´37´ **do** *xchr*[*i*] ← ´␣´;
    for *i* ← ´200´ **to** ´377´ **do** *xchr*[*i*] ← ´␣´;

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

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

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 $print(\#) \equiv write(term\_out, \#)$ {'*print*' means write on the terminal}
  define $print\_ln(\#) \equiv write\_ln(term\_out, \#)$ {'*print*' and then start new line}
  define $new\_line \equiv write\_ln(term\_out)$ {start new line}
  define $print\_nl(\#) \equiv$ {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 $update\_terminal \equiv break(term\_out)$ {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.

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 $0 \leq j < limit$ 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 }
        if *final_limit* > *limit* **then** *final_limit* ← *limit*;
        print_nl(´!␣Input␣line␣too␣long´); *loc* ← 0; *error*;
        end;
      end;
    read_ln(f); *limit* ← *final_limit*; *input_ln* ← *true*;
    end;
  end;

29. Reporting errors to the user. The `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

$$\text{`} err\_print(\text{´}!_{\sqcup}\texttt{Error}_{\sqcup}\texttt{message}\text{´})\text{'},$$

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_skipped* ← *debug_cycle*; *debug_help*; **gubed**
  end;

32. The error locations during Phase I 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 }
  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*(´.␣(l.´, *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 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_start* 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 or 2. (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_start: **array** $[0 . . max\_names]$ **of** *sixteen_bits*;   { directory into *byte_mem* }
tok_start: **array** $[0 . . max\_texts]$ **of** *sixteen_bits*;   { directory into *tok_mem* }
link: **array** $[0 . . max\_names]$ **of** *sixteen_bits*;   { hash table or tree links }
ilk: **array** $[0 . . max\_names]$ **of** *sixteen_bits*;   { type codes or tree links }
equiv: **array** $[0 . . max\_names]$ **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_start*[$p + ww$] − 1, inclusive, in the segment of *byte_mem* whose first index is $p \bmod ww$. Thus, when $ww = 2$ the even-numbered name bytes appear in *byte_mem*[0, ∗] and the odd-numbered ones appear in *byte_mem*[1, ∗]. The pointer 0 is used for undefined module names; we don't want to use it for the names of identifiers, since 0 stands for a null pointer in a linked list.

Strings are 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_start*[*name_ptr* + w] = *byte_ptr*[(*name_ptr* + w) $\bmod ww$] for $0 \le w < ww$, since these are the starting positions for the next ww names to be stored in *byte_mem*.

  define *length*(#) ≡ *byte_start*[# + ww] − *byte_start*[#]   { the length of a name }
⟨ Types in the outer block 11 ⟩ +≡
  name_pointer = 0 . . *max_names*;   { identifies a name }

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 ≠ 1 }

byte_ptr: **array** [0 .. *ww* − 1] **of** 0 .. *max_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 ← 1; *string_ptr* ← 256; *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 $p \bmod zz$. 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 *zz* replacement texts to be stored in *tok_mem*.

⟨Types in the outer block 11⟩ +≡

  text_pointer = 0 .. *max_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 .. *max_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 *equiv* 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 }

48. 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 search tree is used to retrieve module names, with fields called *llink* and *rlink*
in place of *link* and *ilk*. The root of this tree is *rlink*[0]. 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* $\equiv$ *link*   { left link in binary search tree for module names }
define *rlink* $\equiv$ *ilk*   { right link in binary search tree for module names }
⟨ Set initial values 10 ⟩ +≡
   rlink[0] $\leftarrow 0$;   { the binary search tree starts out with nothing in it }
   equiv[0] $\leftarrow 0$;   { 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_mem* }
   begin if $p \geq name\_ptr$ **then** *print*(´IMPOSSIBLE´)
   else begin $w \leftarrow p \bmod ww$;
      for $k \leftarrow byte\_start[p]$ **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 *unambig_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_hash*[*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 *unambig_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 }

hash, *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** *hash_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: $0 \ldots buf\_size$;   {index into $buffer$}
    h: $0 \ldots hash\_size$;   {hash code}
    k: $0 \ldots max\_bytes$;   {index into $byte\_mem$}
    w: $0 \ldots ww - 1$;   {segment of $byte\_mem$}
    l: $0 \ldots buf\_size$;   {length of the given identifier}
    p, q: $name\_pointer$;   {where the identifier is being sought}
    s: $0 \ldots unambig\_length$;   {index into $chopped\_id$}
  begin $l \leftarrow id\_loc - id\_first$;   {compute the length}
  ⟨Compute the hash code h 54⟩;
  ⟨Compute the name location p 55⟩;
  if $(p = name\_ptr) \vee (t \neq normal)$ **then** ⟨Update the tables and check for possible errors 57⟩;
  $id\_lookup \leftarrow p$;
  end;

54.   A simple hash code is used: If the sequence of ASCII codes is $c_1 c_2 \ldots c_n$, its hash value will be

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

⟨Compute the hash code h 54⟩ ≡
  $h \leftarrow buffer[id\_first]$; $i \leftarrow id\_first + 1$;
  while $i < id\_loc$ **do**
    begin $h \leftarrow (h + h + buffer[i]) \bmod hash\_size$; $incr(i)$;
    end
This code is used in section 53.

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

⟨Compute the name location p 55⟩ ≡
  $p \leftarrow hash[h]$;
  while $p \neq 0$ **do**
    begin if $length(p) = l$ **then** ⟨Compare name p with current identifier, **goto** *found* if equal 56⟩;
    $p \leftarrow link[p]$;
    end;
  $p \leftarrow name\_ptr$;   {the current identifier is new}
  $link[p] \leftarrow hash[h]$; $hash[h] \leftarrow p$;   {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 $i \leftarrow id\_first$; $k \leftarrow byte\_start[p]$; $w \leftarrow p \bmod ww$;
  while $(i < id\_loc) \wedge (buffer[i] = byte\_mem[w, k])$ **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 $((p \neq name\_ptr) \wedge (t \neq normal) \wedge (ilk[p] = normal)) \vee ((p = name\_ptr) \wedge (t = normal) \wedge (buffer[id\_first] \neq """"))$ **then** ⟨Compute the secondary hash code h and put the first characters into the auxiliary array *chopped_id* 58⟩;
  if $p \neq name\_ptr$ **then** ⟨Give double-definition error, if necessary, 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 $i \leftarrow id\_first$; $s \leftarrow 0$; $h \leftarrow 0$;
  while $(i < id\_loc) \wedge (s < unambig\_length)$ **do**
    begin if $buffer[i] \neq$ "_" **then**
      begin if $buffer[i] \geq$ "a" **then** $chopped\_id[s] \leftarrow buffer[i] - \text{ } ´40$
      else $chopped\_id[s] \leftarrow buffer[i]$;
      $h \leftarrow (h + h + chopped\_id[s]) \bmod hash\_size$; $incr(s)$;
      end;
    $incr(i)$;
    end;
  $chopped\_id[s] \leftarrow 0$;
  end

This code is used in section 57.

59. If a nonnumeric 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 numeric 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, if necessary, and change p to type t 59⟩ ≡
    { now $p \neq name\_ptr$ and $t \neq normal$ }
  begin if $ilk[p] = normal$ **then**
    begin if $t = numeric$ **then** $err\_print(´!\textvisiblespace This\textvisiblespace identifier\textvisiblespace has\textvisiblespace already\textvisiblespace appeared´)$;
    ⟨Remove p from secondary hash table 60⟩;
    end
  else $err\_print(´!\textvisiblespace This\textvisiblespace identifier\textvisiblespace was\textvisiblespace defined\textvisiblespace before´)$;
  $ilk[p] \leftarrow t$;
  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⟩ ≡
  $q \leftarrow chop\_hash[h]$;
  if $q = p$ **then** $chop\_hash[h] \leftarrow equiv[p]$
  else begin while $equiv[q] \neq p$ **do** $q \leftarrow equiv[q]$;
    $equiv[q] \leftarrow equiv[p]$;
    end

This code is 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 $(t = normal) \wedge (buffer[id\_first] \neq """")$ **then**
　　　⟨Check for ambiguity and update secondary hash 62⟩;
　　$w \leftarrow name\_ptr$ **mod** ww; $k \leftarrow byte\_ptr[w]$;
　　if $k + l > max\_bytes$ **then** $overflow(\text{´byte}_\sqcup\text{memory´})$;
　　if $name\_ptr > max\_names - ww$ **then** $overflow(\text{´name´})$;
　　$i \leftarrow id\_first$; 　{ get ready to move the identifier into $byte\_mem$ }
　　while $i < id\_loc$ **do**
　　　begin $byte\_mem[w, k] \leftarrow buffer[i]$; $incr(k)$; $incr(i)$;
　　　end;
　　$byte\_ptr[w] \leftarrow k$; $byte\_start[name\_ptr + ww] \leftarrow k$; $incr(name\_ptr)$;
　　if $buffer[id\_first] \neq """"$ **then** $ilk[p] \leftarrow t$
　　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 $q \leftarrow chop\_hash[h]$;
　　while $q \neq 0$ **do**
　　　begin ⟨Check if q conflicts with p 63⟩;
　　　$q \leftarrow equiv[q]$;
　　　end;
　　$equiv[p] \leftarrow chop\_hash[h]$; $chop\_hash[h] \leftarrow p$; 　{ put p at front of secondary list }
　　end

This code is used in section 61.

63. ⟨Check if q conflicts with p 63⟩ ≡
　　begin $k \leftarrow byte\_start[q]$; $s \leftarrow 0$; $w \leftarrow q$ **mod** ww;
　　while $(k < byte\_start[q + ww]) \wedge (s < unambig\_length)$ **do**
　　　begin $c \leftarrow byte\_mem[w, k]$;
　　　if $c \neq "\_"$ **then**
　　　　begin if $c \geq "a"$ **then** $c \leftarrow c - ´40$; 　{ merge lowercase with uppercase }
　　　　if $chopped\_id[s] \neq c$ **then goto** not_found;
　　　　$incr(s)$;
　　　　end;
　　　$incr(k)$;
　　　end;
　　if $(k = byte\_start[q + ww]) \wedge (chopped\_id[s] \neq 0)$ **then goto** not_found;
　　$print\_nl(\text{´!}_\sqcup\text{Identifier}_\sqcup\text{conflict}_\sqcup\text{with}_\sqcup\text{´})$;
　　for $k \leftarrow byte\_start[q]$ **to** $byte\_start[q + ww] - 1$ **do** $print(xchr[byte\_mem[w, k]])$;
　　$error$; $q \leftarrow 0$; 　{ 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 $check\_sum\_prime \equiv \text{´}3777777667$ $\{2^{29} - 73\}$

$\langle$ Define and output a new string of the pool $64\rangle \equiv$
  begin $ilk[p] \leftarrow numeric;$ $\{$ strings are like numeric macros $\}$
  if $l - double\_chars = 2$ **then** $\{$ this string is for a single character $\}$
    $equiv[p] \leftarrow buffer[id\_first + 1] + \text{´}100000$
  else begin $equiv[p] \leftarrow string\_ptr + \text{´}100000;$ $l \leftarrow l - double\_chars - 1;$
    if $l > 99$ **then** $err\_print(\text{´}!\sqcup\text{Preprocessed}\sqcup\text{string}\sqcup\text{is}\sqcup\text{too}\sqcup\text{long}\text{´});$
    $incr(string\_ptr);$ $write(pool, xchr[\text{"0"} + l \textbf{ div } 10], xchr[\text{"0"} + l \textbf{ mod } 10]);$ $\{$ output the length $\}$
    $pool\_check\_sum \leftarrow pool\_check\_sum + pool\_check\_sum + l;$
    while $pool\_check\_sum > check\_sum\_prime$ **do** $pool\_check\_sum \leftarrow pool\_check\_sum - check\_sum\_prime;$
    $i \leftarrow id\_first + 1;$
    while $i < id\_loc$ **do**
      begin $write(pool, xchr[buffer[i]]);$ $\{$ output characters of string $\}$
      $pool\_check\_sum \leftarrow pool\_check\_sum + pool\_check\_sum + buffer[i];$
      while $pool\_check\_sum > check\_sum\_prime$ **do** $pool\_check\_sum \leftarrow pool\_check\_sum - check\_sum\_prime;$
      if $(buffer[i] = \text{""""}) \vee (buffer[i] = \text{"@"})$ **then** $i \leftarrow i + 2$
          $\{$ 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 $mod\_text[1 \mathbin{.\,.} l]$ 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** $[0 \mathbin{.\,.} longest\_name]$ **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: $less \mathbin{.\,.} extension$;   { comparison between two names }
      j: $0 \mathbin{.\,.} longest\_name$;   { index into mod_text }
      k: $0 \mathbin{.\,.} max\_bytes$;   { index into $byte\_mem$ }
      w: $0 \mathbin{.\,.} ww - 1$;   { segment of $byte\_mem$ }
      p: $name\_pointer$;   { current node of the search tree }
      q: $name\_pointer$;   { father of node p }
   begin $c \leftarrow greater$; $q \leftarrow 0$; $p \leftarrow rlink[0]$;   { $rlink[0]$ is the root of the tree }
   while $p \neq 0$ **do**
      begin ⟨ Set c to the result of comparing the given name to name p 68 ⟩;
      $q \leftarrow p$;
      if $c = less$ **then** $p \leftarrow llink[q]$
      else if $c = greater$ **then** $p \leftarrow rlink[q]$
         else goto $found$;
      end;
   ⟨ Enter a new module name into the tree 67 ⟩;
$found$: **if** $c \neq equal$ **then**
      begin $err\_print(\texttt{'!\_Incompatible\_section\_names'})$; $p \leftarrow 0$;
      end;
   $mod\_lookup \leftarrow p$;
   end;

67.   ⟨ Enter a new module name into the tree 67 ⟩ ≡
   $w \leftarrow name\_ptr$ **mod** ww; $k \leftarrow byte\_ptr[w]$;
   if $k + l > max\_bytes$ **then** $overflow(\texttt{'byte\_memory'})$;
   if $name\_ptr > max\_names - ww$ **then** $overflow(\texttt{'name'})$;
   $p \leftarrow name\_ptr$;
   if $c = less$ **then** $llink[q] \leftarrow p$
   else $rlink[q] \leftarrow p$;
   $llink[p] \leftarrow 0$; $rlink[p] \leftarrow 0$; $c \leftarrow equal$; $equiv[p] \leftarrow 0$;
   for $j \leftarrow 1$ **to** l **do** $byte\_mem[w, k + j - 1] \leftarrow mod\_text[j]$;
   $byte\_ptr[w] \leftarrow k + l$; $byte\_start[name\_ptr + ww] \leftarrow k + l$; $incr(name\_ptr)$;
This code is used in section 66.

68. $\langle$ Set c to the result of comparing the given name to name p 68 $\rangle \equiv$
 begin $k \leftarrow byte\_start[p]$; $w \leftarrow p \bmod ww$; $c \leftarrow equal$; $j \leftarrow 1$;
 while $(k < byte\_start[p + ww]) \wedge (j \leq l) \wedge (mod\_text[j] = byte\_mem[w, k])$ **do**
  begin $incr(k)$; $incr(j)$;
  end;
 if $k = byte\_start[p + ww]$ **then**
  if $j > l$ **then** $c \leftarrow equal$
  else $c \leftarrow extension$
 else if $j > l$ **then** $c \leftarrow prefix$
  else if $mod\_text[j] < byte\_mem[w, k]$ **then** $c \leftarrow less$
   else $c \leftarrow greater$;
 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 $q \leftarrow 0$; $p \leftarrow rlink[0]$; $count \leftarrow 0$; $r \leftarrow 0$; { begin search at root of tree }
 while $p \neq 0$ **do**
  begin $\langle$ Set c to the result of comparing the given name to name p 68 $\rangle$;
  if $c = less$ **then** $p \leftarrow llink[p]$
  else if $c = greater$ **then** $p \leftarrow rlink[p]$
   else begin $r \leftarrow p$; $incr(count)$; $q \leftarrow rlink[p]$; $p \leftarrow llink[p]$;
    end;
  if $p = 0$ **then**
   begin $p \leftarrow q$; $q \leftarrow 0$;
   end;
  end;
 if $count \neq 1$ **then**
  if $count = 0$ **then** $err\_print(´!\_Name\_does\_not\_match´)$
  else $err\_print(´!\_Ambiguous\_prefix´)$;
 $prefix\_lookup \leftarrow r$; { 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* ≡ *max_texts*   { final *text_link* in module replacement texts }

⟨ Globals in the outer block 9 ⟩ +≡
last_unnamed: *text_pointer*;   { most recent replacement text of unnamed module }

71.   ⟨ Set initial values 10 ⟩ +≡
    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 ≤ a < ´250, then $(a - ´200) \times 2^8 + b$ points to an identifier; if ´250 ≤ a < ´320, then $(a - ´250) \times 2^8 + b$ points to a module name; otherwise, i.e., if ´320 ≤ 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. The @> at the end of such a string is also denoted by *verbatim*.

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* : *sixteen_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] ← *x* **mod** ´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* ← *p* **mod** *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 + *tok_mem*[*zp*, *k*]; *print_id*(*a*);
    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* ← (*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**

 $begin\_comment$: $print(\text{´@´}, xchr[\text{"\{"}])$;

 $end\_comment$: $print(\text{´@´}, xchr[\text{"\}"}])$; { can't use right brace between **debug** and **gubed** }

 $octal$: $print(\text{´@´´´})$;

 hex: $print(\text{´@"´})$;

 $check\_sum$: $print(\text{´@\$´})$;

 $param$: $print(\text{´#´})$;

 "@": $print(\text{´@@´})$;

 $verbatim$: $print(\text{´@=´})$;

 $force\_line$: $print(\text{´@\\´})$;

 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 *stack* 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 *stack_ptr* = 0.

⟨ Types in the outer block 11 ⟩ +≡
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* ≡ *cur_state.end_field*   { current ending location in *tok_mem* }
  define *cur_byte* ≡ *cur_state.byte_field*   { location of next output byte in *tok_mem* }
  define *cur_name* ≡ *cur_state.name_field*   { pointer to current name being expanded }
  define *cur_repl* ≡ *cur_state.repl_field*   { pointer to current replacement text }
  define *cur_mod* ≡ *cur_state.mod_field*   { current module number being expanded }

⟨ Globals in the outer block 9 ⟩ +≡
cur_state: *output_state*;   { *cur_end*, *cur_byte*, *cur_name*, *cur_repl*, *cur_mod* }
stack: **array** [1 .. *stack_size*] **of** *output_state*;   { info for non-current levels }
stack_ptr: 0 .. *stack_size*;   { first unused location in the output state stack }

80.   It is convenient to keep a global variable *zo* equal to *cur_repl* **mod** *zz*.

⟨ Globals in the outer block 9 ⟩ +≡
zo: 0 .. *zz* − 1;   { the segment of *tok_mem* from which output is coming }

81.   Parameters must also be stacked. They are 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 problem.

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 ⟩ +≡
brace_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** $cur\_val \leftarrow a$; $a \leftarrow identifier$;
      end;
   $numeric$: **begin** $cur\_val \leftarrow equiv[a] - ´100000$; $a \leftarrow number$;
      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 $(cur\_byte = cur\_end) \wedge (stack\_ptr > 0)$ **do** pop_level;
   if $(stack\_ptr = 0) \vee (tok\_mem[zo, cur\_byte] \neq$ "(") **then**
      begin $print\_nl(´!\_No\_parameter\_given\_for\_´)$; $print\_id(a)$; $error$; **goto** $restart$;
      end;
   ⟨Copy the parameter into tok_mem 93⟩;
   $equiv[name\_ptr] \leftarrow text\_ptr$; $ilk[name\_ptr] \leftarrow simple$; $w \leftarrow name\_ptr$ **mod** ww; $k \leftarrow byte\_ptr[w]$;
   debug if $k = max\_bytes$ **then** $overflow(´byte\_memory´)$;
   $byte\_mem[w, k] \leftarrow$ "#"; $incr(k)$; $byte\_ptr[w] \leftarrow k$;
   gubed   { this code has set the parameter identifier for debugging printouts }
   if $name\_ptr > max\_names - ww$ **then** $overflow(´name´)$;
   $byte\_start[name\_ptr + ww] \leftarrow k$; $incr(name\_ptr)$;
   if $text\_ptr > max\_texts - zz$ **then** $overflow(´text´)$;
   $text\_link[text\_ptr] \leftarrow 0$; $tok\_start[text\_ptr + zz] \leftarrow tok\_ptr[z]$; $incr(text\_ptr)$; $z \leftarrow text\_ptr$ **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 $decr(name\_ptr)$; $decr(text\_ptr)$; $z \leftarrow text\_ptr$ **mod** zz;
   stat if $tok\_ptr[z] > max\_tok\_ptr[z]$ **then** $max\_tok\_ptr[z] \leftarrow tok\_ptr[z]$;
   tats   { the maximum value of tok_ptr occurs just before parameter popping }
   $tok\_ptr[z] \leftarrow tok\_start[text\_ptr]$;
   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 $(name\_ptr - 1)$:

$\langle$ Start scanning current macro parameter, **goto** $restart$ 92 $\rangle \equiv$
  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 $macro(\texttt{\#}; print(\texttt{`\#)`}))$; the $\texttt{\#}$ token will have been changed to $param$ outside of strings, but we still must distinguish 'real' parentheses from those in strings.

  define $app\_repl(\texttt{\#}) \equiv$
       begin if $tok\_ptr[z] = max\_toks$ **then** $overflow(\texttt{`token`})$;
       $tok\_mem[z, tok\_ptr[z]] \leftarrow \texttt{\#}$; $incr(tok\_ptr[z])$;
       end

$\langle$ Copy the parameter into tok_mem 93 $\rangle \equiv$
  $bal \leftarrow 1$; $incr(cur\_byte)$;   { skip the opening '(' }
  loop begin $b \leftarrow tok\_mem[zo, cur\_byte]$; $incr(cur\_byte)$;
    if $b = param$ **then** $store\_two\_bytes(name\_ptr + \texttt{´77777})$
    else begin if $b \geq \texttt{´200}$ **then**
      begin $app\_repl(b)$; $b \leftarrow tok\_mem[zo, cur\_byte]$; $incr(cur\_byte)$;
      end
    else case b **of**
      "(": $incr(bal)$;
      ")": **begin** $decr(bal)$;
        if $bal = 0$ **then goto** $done$;
        end;
      "´": **repeat** $app\_repl(b)$; $b \leftarrow tok\_mem[zo, cur\_byte]$; $incr(cur\_byte)$;
        until $b = \texttt{"´"}$;   { 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 are 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 $1 \leq out\_ptr \leq line\_length$ most of the
time. The second is *break_ptr*, which is the largest value $\leq out\_ptr$ such that we are definitely entitled to end
a line by outputting the characters *out_buf* $[1 .. (break\_ptr - 1)]$; we will always have $break\_ptr \leq line\_length$.
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 $semi\_ptr \leq break\_ptr$.

⟨ 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		−1
+	(x	*sign_val_sign*	"+"	−15	+1	+1
19	(x	*sign_val_val*	"+"	−15	+19	+1
−	(x	*sign_val_sign*	"+"	+4	−1	−1
2	(x	*sign_val_val*	"+"	+4	−2	−1
)	(x+2)	*misc*				

At each stage we have 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 $sign = 2$ { state 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 ⟩ ≡
  break_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 the output buffer.

> **define** $app(\#) \equiv$
> > **begin** $out\_buf[out\_ptr] \leftarrow \#$; $incr(out\_ptr)$;   { append a single character }
> > **end**

procedure $app\_val(v : integer)$;   { puts v into buffer, assumes $v \geq 0$ }
  var k: $0 \mathinner{.\,.} out\_buf\_size$;   { index into out_buf }
  begin $k \leftarrow out\_buf\_size$;   { first we put the digits at the very end of out_buf }
  repeat $out\_buf[k] \leftarrow v \bmod 10$; $v \leftarrow v \mathbf{div} 10$; $decr(k)$;
  until $v = 0$;
  repeat $incr(k)$; $app(out\_buf[k] + \texttt{"0"})$;
  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 be 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 }

$\langle$ Globals in the outer block 9 $\rangle$ $+\equiv$
$out\_contrib$: **array** $[1 \mathinner{.\,.} line\_length]$ **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: $0 \mathinner{.\,.} line\_length$;   { index into $out\_contrib$ }
  begin $\langle$ Get the buffer ready for appending the new information 102 $\rangle$;
  if $t \neq misc$ **then**
    for $k \leftarrow 1$ **to** v **do** $app(out\_contrib[k])$
  else $app(v)$;
  $check\_break$;
  if $(t = misc) \wedge ((v = \texttt{";"}) \vee (v = \texttt{"\}"}))$ **then**
    begin $semi\_ptr \leftarrow out\_ptr$; $break\_ptr \leftarrow out\_ptr$;
    end;
  if $t \geq ident$ **then** $out\_state \leftarrow num\_or\_id$ { $t = ident$ or $frac$ }
  else $out\_state \leftarrow misc$ { $t = str$ 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** $t \neq frac$ **then**
      begin $break\_ptr \leftarrow out\_ptr$;
      if $t = ident$ **then** $app(\text{"}\sqcup\text{"})$;
      end;
  $sign$: **begin** $app("," - out\_app)$; $check\_break$; $break\_ptr \leftarrow out\_ptr$;
    end;
  $sign\_val$, $sign\_val\_sign$: **begin** ⟨ Append out_val to buffer 103 ⟩;
    $out\_state \leftarrow out\_state - 2$; **goto** $restart$;
    end;
  $sign\_val\_val$: ⟨ Reduce $sign\_val\_val$ to $sign\_val$ and **goto** $restart$ 104 ⟩;
  $misc$: **if** $t \neq frac$ **then** $break\_ptr \leftarrow out\_ptr$;
    othercases $do\_nothing$ { this is for $unbreakable$ state }
  endcases

This code is used in section 101.

103.   ⟨ Append out_val to buffer 103 ⟩ ≡
  if $(out\_val < 0) \vee ((out\_val = 0) \wedge (last\_sign < 0))$ **then** $app("-")$
  else if $out\_sign > 0$ **then** $app(out\_sign)$;
  $app\_val(abs(out\_val))$; $check\_break$;

This code is used in sections 102 and 104.

104.   ⟨ Reduce $sign\_val\_val$ to $sign\_val$ and **goto** $restart$ 104 ⟩ ≡
  begin if $(t = frac) \vee (⟨$ Contribution is $*$ or $/$ or DIV or MOD 105 ⟩) **then**
    begin ⟨ Append out_val to buffer 103 ⟩;
    $out\_sign \leftarrow "+"$; $out\_val \leftarrow out\_app$;
    end
  else $out\_val \leftarrow out\_val + out\_app$;
  $out\_state \leftarrow sign\_val$; **goto** $restart$;
  end

This code is used in section 102.

105.   ⟨ Contribution is $*$ or $/$ or DIV or MOD 105 ⟩ ≡
  $((t = ident) \wedge (v = 3) \wedge (((out\_contrib[1] = "D") \wedge (out\_contrib[2] = "I") \wedge (out\_contrib[3] = "V")) \vee$
    $((out\_contrib[1] = "M") \wedge (out\_contrib[2] = "O") \wedge (out\_contrib[3] = "D")))) \vee$
    $((t = misc) \wedge ((v = "*") \vee (v = "/")))$

This code is used in section 104.

106.   The following routine is called with $v = \pm 1$ 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**
  $sign, sign\_val\_sign$: $out\_app \leftarrow out\_app * v$;
  $sign\_val$: **begin** $out\_app \leftarrow v$; $out\_state \leftarrow sign\_val\_sign$;
    end;
  $sign\_val\_val$: **begin** $out\_val \leftarrow out\_val + out\_app$; $out\_app \leftarrow v$; $out\_state \leftarrow sign\_val\_sign$;
    end;
  othercases begin $break\_ptr \leftarrow out\_ptr$; $out\_app \leftarrow v$; $out\_state \leftarrow sign$;
    end
  endcases;
  $last\_sign \leftarrow out\_app$;
  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⟩;
    $out\_sign \leftarrow "\sqcup"$; $out\_state \leftarrow sign\_val$; $out\_val \leftarrow v$; $break\_ptr \leftarrow out\_ptr$; $last\_sign \leftarrow +1$;
    end;
  $misc$: **begin** ⟨If previous output was * or /, **goto** bad_case 109⟩;
    $out\_sign \leftarrow 0$; $out\_state \leftarrow sign\_val$; $out\_val \leftarrow v$; $break\_ptr \leftarrow out\_ptr$; $last\_sign \leftarrow +1$;
    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** $out\_sign \leftarrow "+"$; $out\_state \leftarrow sign\_val$; $out\_val \leftarrow out\_app * v$;
  end;
$sign\_val$: **begin** $out\_state \leftarrow sign\_val\_val$; $out\_app \leftarrow v$;
  $err\_print($ ´!␣Two␣numbers␣occurred␣without␣a␣sign␣between␣them´$)$;
  end;
$sign\_val\_sign$: **begin** $out\_state \leftarrow sign\_val\_val$; $out\_app \leftarrow out\_app * v$;
  end;
$sign\_val\_val$: **begin** $out\_val \leftarrow out\_val + out\_app$; $out\_app \leftarrow v$;
  $err\_print($ ´!␣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* ← *misc*;
    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*(*str*, 2);
  end;
not_equal: **begin** *out_contrib*[1] ← "<"; *out_contrib*[2] ← ">"; *send_out*(*str*, 2);
  end;
less_or_equal: **begin** *out_contrib*[1] ← "<"; *out_contrib*[2] ← "="; *send_out*(*str*, 2);
  end;
greater_or_equal: **begin** *out_contrib*[1] ← ">"; *out_contrib*[2] ← "="; *send_out*(*str*, 2);
  end;
equivalence_sign: **begin** *out_contrib*[1] ← "="; *out_contrib*[2] ← "="; *send_out*(*str*, 2);
  end;
double_dot: **begin** *out_contrib*[1] ← "."; *out_contrib*[2] ← "."; *send_out*(*str*, 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 $up\_to(\#) \equiv \# - 24, \# - 23, \# - 22, \# - 21, \# - 20, \# - 19, \# - 18, \# - 17, \# - 16, \# - 15, \# - 14, \# - 13,$
$\# - 12, \# - 11, \# - 10, \# - 9, \# - 8, \# - 7, \# - 6, \# - 5, \# - 4, \# - 3, \# - 2, \# - 1, \#$

⟨ 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** $k \leftarrow 0$; $j \leftarrow byte\_start[cur\_val]$; $w \leftarrow cur\_val$ **mod** ww;
  while $(k < max\_id\_length) \wedge (j < byte\_start[cur\_val + ww])$ **do**
    begin $incr(k)$; $out\_contrib[k] \leftarrow byte\_mem[w, j]$; $incr(j)$;
    if $out\_contrib[k] \geq$ "a" **then** $out\_contrib[k] \leftarrow out\_contrib[k] - ´40$
    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 $k \leftarrow 1$; $out\_contrib[1] \leftarrow$ "´";
  repeat if $k < line\_length$ **then** $incr(k)$;
    $out\_contrib[k] \leftarrow get\_output$;
  until $(out\_contrib[k] =$ "´"$) \vee (stack\_ptr = 0)$;
  if $k = line\_length$ **then** $err\_print($´!␣String␣too␣long´$)$;
  send_out(*str*, *k*); $cur\_char \leftarrow get\_output$;
  if $cur\_char =$ "´" **then** $out\_state \leftarrow unbreakable$;
  goto *reswitch*;
  end

This code is used in section 113.

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

⟨ Send verbatim string 118 ⟩ ≡
  begin $k \leftarrow 0$;
  repeat if $k < line\_length$ **then** $incr(k)$;
    $out\_contrib[k] \leftarrow get\_output$;
  until $(out\_contrib[k] = verbatim) \vee (stack\_ptr = 0)$;
  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** $n \leftarrow 0$;
  repeat *cur_char* ← *cur_char* − "0";
    if $n \geq \text{´}1463146314$ **then** *err_print*(´!␣Constant␣too␣big´)
    else $n \leftarrow 10 * n + \text{\textit{cur\_char}}$;
    cur_char ← *get_output*;
  until (*cur_char* > "9") ∨ (*cur_char* < "0");
  send_val(*n*); $k \leftarrow 0$;
  if *cur_char* = "e" **then** *cur_char* ← "E";
  if *cur_char* = "E" **then goto** *get_fraction*
  else goto *reswitch*;
  end;
check_sum: *send_val*(*pool_check_sum*);
octal: **begin** $n \leftarrow 0$; *cur_char* ← "0";
  repeat *cur_char* ← *cur_char* − "0";
    if $n \geq \text{´}2000000000$ **then** *err_print*(´!␣Constant␣too␣big´)
    else $n \leftarrow 8 * n + \text{\textit{cur\_char}}$;
    cur_char ← *get_output*;
  until (*cur_char* > "7") ∨ (*cur_char* < "0");
  send_val(*n*); **goto** *reswitch*;
  end;
hex: **begin** $n \leftarrow 0$; *cur_char* ← "0";
  repeat if *cur_char* ≥ "A" **then** *cur_char* ← *cur_char* + 10 − "A"
    else *cur_char* ← *cur_char* − "0";
    if $n \geq \text{″}8000000$ **then** *err_print*(´!␣Constant␣too␣big´)
    else $n \leftarrow 16 * n + \text{\textit{cur\_char}}$;
    cur_char ← *get_output*;
  until (*cur_char* > "F") ∨ (*cur_char* < "0") ∨ ((*cur_char* > "9") ∧ (*cur_char* < "A"));
  send_val(*n*); **goto** *reswitch*;
  end;
number: *send_val*(*cur_val*);
".": **begin** $k \leftarrow 1$; *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") ∧ (*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** $incr(k)$;
> $out\_contrib[k] \leftarrow cur\_char$; $cur\_char \leftarrow get\_output$;
>   **if** $(out\_contrib[k] = \text{"E"}) \wedge ((cur\_char = \text{"+"}) \vee (cur\_char = \text{"-"}))$ **then**
>   **begin if** $k < line\_length$ **then** $incr(k)$;
> $out\_contrib[k] \leftarrow cur\_char$; $cur\_char \leftarrow get\_output$;
>   **end**
>   **else if** $cur\_char = \text{"e"}$ **then** $cur\_char \leftarrow \text{"E"}$;
> **until** $(cur\_char \neq \text{"E"}) \wedge ((cur\_char < \text{"0"}) \vee (cur\_char > \text{"9"}))$;
> **if** $k = line\_length$ **then** $err\_print(\text{´!}_\sqcup\text{Fraction}_\sqcup\text{too}_\sqcup\text{long´})$;
> $send\_out(frac, k)$; **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 '$out\_contrib[1] \leftarrow \text{"{"}$' that appears here should be replaced by '**begin** $out\_contrib[1] \leftarrow \text{"("}$; $out\_contrib[2] \leftarrow \text{"*"}$; $incr(k)$; **end**', and a similar change should be made to '$out\_contrib[k] \leftarrow \text{"}\text{"}$'.

⟨ Cases involving @{ and @} 121 ⟩ ≡
begin_comment: **begin if** $brace\_level = 0$ **then** $send\_out(misc, \text{"{"})$
> **else** $send\_out(misc, \text{"["})$;
> $incr(brace\_level)$;
> **end**;
end_comment: **if** $brace\_level > 0$ **then**
>   **begin** $decr(brace\_level)$;
>   **if** $brace\_level = 0$ **then** $send\_out(misc, \text{"}\text{"})$
>   **else** $send\_out(misc, \text{"]"})$;
>   **end**
> **else** $err\_print(\text{´!}_\sqcup\text{Extra}_\sqcup\text{@}\text{´})$;
module_number: **begin** $k \leftarrow 2$;
> **if** $brace\_level = 0$ **then** $out\_contrib[1] \leftarrow \text{"{"}$
> **else** $out\_contrib[1] \leftarrow \text{"["}$;
> **if** $cur\_val < 0$ **then**
>   **begin** $out\_contrib[k] \leftarrow \text{":"}$; $cur\_val \leftarrow -cur\_val$; $incr(k)$;
>   **end**;
> $n \leftarrow 10$;
> **while** $cur\_val \geq n$ **do** $n \leftarrow 10 * n$;
> **repeat** $n \leftarrow n$ **div** 10; $out\_contrib[k] \leftarrow \text{"0"} + (cur\_val$ **div** $n)$; $cur\_val \leftarrow cur\_val$ **mod** n; $incr(k)$;
> **until** $n = 1$;
> **if** $out\_contrib[2] \neq \text{":"}$ **then**
>   **begin** $out\_contrib[k] \leftarrow \text{":"}$; $incr(k)$;
>   **end**;
> **if** $brace\_level = 0$ **then** $out\_contrib[k] \leftarrow \text{"}\text{"}$
> **else** $out\_contrib[k] \leftarrow \text{"]"}$;
> $send\_out(str, k)$;
> **end**;

This code is used in section 113.

122.   ⟨Force a line break 122⟩ ≡
  begin *send_out*(*str*, 0);   {normalize the buffer}
  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 ⟩ +≡

ii: *integer*;   { general purpose **for** loop variable in the outer block }

line: *integer*;   { the number of the current line in the current file }

other_line: *integer*;   { the number of the current line in the input file that is not currently being read }

temp_line: *integer*;   { used when interchanging *line* with *other_line* }

limit: 0 .. *buf_size*;   { the last character position occupied in the buffer }

loc: 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 $(change\_limit = 0) \wedge \neg changing$ if and only if the change file is exhausted. This procedure is called only when *changing* is true; hence error messages will be reported correctly.

procedure *prime_the_change_buffer*;
    label *continue*, *done*, *exit*;
    var k: $0 \mathrel{..} buf\_size$;   { index into the buffers }
    begin *change_limit* $\leftarrow 0$;   { 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] $\neq$ "@" **then goto** *continue*;
        if (*buffer*[1] $\geq$ "X") $\wedge$ (*buffer*[1] $\leq$ "Z") **then** *buffer*[1] $\leftarrow$ *buffer*[1] + "z" − "Z";   { lowercasify }
        if *buffer*[1] = "x" **then goto** *done*;
        if (*buffer*[1] = "y") $\vee$ (*buffer*[1] = "z") **then**
            begin *loc* $\leftarrow 2$; *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* $\leftarrow$ *limit*;
    if *limit* > 0 **then**
        for $k \leftarrow 0$ **to** *limit* $- 1$ **do** *change_buffer*[k] $\leftarrow$ *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 *change_changing*;   { now it's *true* }
    incr(*line*);
    if ¬*input_ln*(*change_file*) **then**
      begin *err_print*(´!␣Change␣file␣ended␣before␣@y´); *change_limit* ← 0; *change_changing*;
          { *false* again }
      return;
      end;
    ⟨ If the current line starts with @y, report any discrepancies and **return** 133 ⟩;
    ⟨ Move *buffer* and *limit* to *change_buffer* and *change_limit* 131 ⟩;
    change_changing;   { now it's *false* }
    incr(*line*);
    if ¬*input_ln*(*web_file*) **then**
      begin *err_print*(´!␣WEB␣file␣ended␣during␣a␣change´); *input_has_ended* ← *true*; **return**;
      end;
    if *lines_dont_match* **then** *incr*(*n*);
    end;
exit: **end**;

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*; *change_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 *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 *ii* ← 0 **to** *change_limit* − 1 **do** *buffer* [*ii*] ← *change_buffer* [*ii*];
    limit ← *change_limit*; *changing* ← *true*; *line* ← *other_line*; *loc* ← *change_limit*;
    err_print (´!␣Change␣file␣entry␣did␣not␣match´);
    end

This code is used in section 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 *loc* > *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* ← *control_code*(*buffer*[*loc* − 1]);
      if (*c* ≠ *ignore*) ∨ (*buffer*[*loc* − 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*[*loc*]; *incr*(*loc*); ⟨ 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 $c \leftarrow buffer[loc]$;
    if $(c \neq$ "␣"$) \wedge (c \neq tab\_mark) \wedge (c \neq$ "*"$)$ **then** $incr(loc)$
    else begin $err\_print($ ´!␣Section␣ended␣in␣mid-comment´$)$; $decr(loc)$; **return**;
      end
    end
  else if $(c = $ "\"$) \wedge (buffer[loc] \neq$ "@"$)$ **then** $incr(loc)$
    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_next* 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_hex* **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_mark*: **goto** *restart*;   { ignore spaces and tabs }
  "{": **begin** *skip_comment*; **goto** *restart*;
    end;
  "}": **begin** *err_print*(´!␣Extra␣}´); **goto** *restart*;
    end;
  othercases if *c* ≥ 128 **then goto** *restart*   { ignore nonstandard characters }
    else *do_nothing*
  endcases;
found: **debug if** *trouble_shooting* **then** *debug_help*; **gubed**
  get_next ← *c*;
  end;

146. ⟨Go to *found* if *c* is a hexadecimal digit, otherwise set *scanning_hex* ← *false* 146⟩ ≡
  if $((c \geq "0") \land (c \leq "9")) \lor ((c \geq "A") \land (c \leq "F"))$ **then goto** *found*
  else *scanning_hex* ← *false*

This code is 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 if $loc \leq limit$ **then**
         begin $c \leftarrow$ #; *incr*(*loc*);
         end;
       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 $((c = "e") \lor (c = "E")) \land (loc > 1)$ **then**
    if $(buffer[loc - 2] \leq "9") \land (buffer[loc - 2] \geq "0")$ **then** $c \leftarrow 0$;
  if $c \neq 0$ **then**
    begin *decr*(*loc*); *id_first* ← *loc*;
    repeat *incr*(*loc*); $d \leftarrow buffer[loc]$;
    until $((d < "0") \lor ((d > "9") \land (d < "A")) \lor ((d > "Z") \land (d < "a")) \lor (d > "z")) \land (d \neq "_")$;
    if $loc > id\_first + 1$ **then**
      begin $c \leftarrow identifier$; *id_loc* ← *loc*;
      end;
    end
  else $c \leftarrow "E"$;   {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 $double\_chars \leftarrow 0$; $id\_first \leftarrow loc - 1$;
  repeat $d \leftarrow buffer[loc]$; $incr(loc)$;
    if $(d = \texttt{""""}) \vee (d = \texttt{"@"})$ **then**
      if $buffer[loc] = d$ **then**
        begin $incr(loc)$; $d \leftarrow 0$; $incr(double\_chars)$;
        end
      else begin if $d = \texttt{"@"}$ **then** $err\_print(´!_⊔\texttt{Double}_⊔\texttt{@}_⊔\texttt{sign}_⊔\texttt{missing}´)$
        end
    else if $loc > limit$ **then**
        begin $err\_print(´!_⊔\texttt{String}_⊔\texttt{constant}_⊔\texttt{didn}´´\texttt{t}_⊔\texttt{end}´)$; $d \leftarrow \texttt{""""}$;
        end;
  until $d = \texttt{""""}$;
  $id\_loc \leftarrow loc - 1$; $c \leftarrow identifier$;
  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 $c \leftarrow control\_code(buffer[loc])$; $incr(loc)$;
  if $c = ignore$ **then goto** $restart$
  else if $c = hex$ **then** $scanning\_hex \leftarrow true$
    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 $c \leftarrow skip\_ahead$;
          until $c \neq \texttt{"@"}$;
          if $buffer[loc - 1] \neq \texttt{">"}$ **then** $err\_print(´!_⊔\texttt{Improper}_⊔\texttt{@}_⊔\texttt{within}_⊔\texttt{control}_⊔\texttt{text}´)$;
          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 $(mod\_text[k] = \texttt{"."}) \wedge (mod\_text[k-1] = \texttt{"."}) \wedge (mod\_text[k-2] = \texttt{"."})$ **then**
    $cur\_module \leftarrow prefix\_lookup(k - 3)$
    else $cur\_module \leftarrow mod\_lookup(k)$;
    end
  else $cur\_module \leftarrow mod\_lookup(k)$;
  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 $mod\_text[0] \leftarrow \texttt{"}_⊔\texttt{"}$ to facilitate this, since the mod_lookup routine uses $mod\_text[1]$ as the first character of the name.)

⟨ Set initial values 10 ⟩ +≡
  $mod\_text[0] \leftarrow \texttt{"}_⊔\texttt{"}$;

153.   ⟨Put module name into $mod\_text[1 .. k]$ 153⟩ ≡
  $k \leftarrow 0$;
  loop begin if $loc > limit$ **then**
      begin get_line;
      if $input\_has\_ended$ **then**
        begin $err\_print(\text{'}!_{\sqcup}\text{Input}_{\sqcup}\text{ended}_{\sqcup}\text{in}_{\sqcup}\text{section}_{\sqcup}\text{name'})$; **goto** $done$;
        end;
      end;
    $d \leftarrow buffer[loc]$; ⟨If end of name, **goto** $done$ 154⟩;
    $incr(loc)$;
    if $k < longest\_name - 1$ **then** $incr(k)$;
    if $(d = "_{\sqcup}") \vee (d = tab\_mark)$ **then**
      begin $d \leftarrow "_{\sqcup}"$;
      if $mod\_text[k-1] = "_{\sqcup}"$ **then** $decr(k)$;
      end;
    $mod\_text[k] \leftarrow d$;
    end;
$done$: ⟨Check for overlong name 155⟩;
  if $(mod\_text[k] = "_{\sqcup}") \wedge (k > 0)$ **then** $decr(k)$;
This code is used in section 151.

154.   ⟨If end of name, **goto** $done$ 154⟩ ≡
  if $d = "@"$ **then**
    begin $d \leftarrow buffer[loc + 1]$;
    if $d = ">"$ **then**
      begin $loc \leftarrow loc + 2$; **goto** $done$;
      end;
    if $(d = "_{\sqcup}") \vee (d = tab\_mark) \vee (d = "*")$ **then**
      begin $err\_print(\text{'}!_{\sqcup}\text{Section}_{\sqcup}\text{name}_{\sqcup}\text{didn´´t}_{\sqcup}\text{end'})$; **goto** $done$;
      end;
    $incr(k)$; $mod\_text[k] \leftarrow "@"$; $incr(loc)$;   { now $d = buffer[loc]$ again }
    end
This code is used in section 153.

155.   ⟨Check for overlong name 155⟩ ≡
  if $k \geq longest\_name - 2$ **then**
    begin $print\_nl(\text{'}!_{\sqcup}\text{Section}_{\sqcup}\text{name}_{\sqcup}\text{too}_{\sqcup}\text{long:}_{\sqcup}\text{'})$;
    for $j \leftarrow 1$ **to** 25 **do** $print(xchr[mod\_text[j]])$;
    $print(\text{'}...\text{'})$; $mark\_harmless$;
    end
This code is used in section 153.

156. Scanning a numeric definition.   When `TANGLE` looks at the Pascal text following the '=' of a numeric macro definition, it calls on the procedure $scan\_numeric(p)$, 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** $end\_of\_definition(\#) \equiv (\# \geq format)$ { is # a control code ending a definition? }

⟨ Globals in the outer block 9 ⟩ +≡
$next\_control$: $eight\_bits$;   { 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 $next\_sign$ is reset to +1.

> **define** $add\_in(\#) \equiv$
> > **begin** $accumulator \leftarrow accumulator + next\_sign * (\#)$; $next\_sign \leftarrow +1$;
> > **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 }
    val: $integer$;   { constants being evaluated }
  begin ⟨ Set $accumulator$ to the value of the right-hand side 158 ⟩;
  if $abs(accumulator) \geq ´100000$ **then**
    begin $err\_print(´!_{\sqcup}Value_{\sqcup}too_{\sqcup}big:_{\sqcup}´, accumulator : 1)$; $accumulator \leftarrow 0$;
    end;
  $equiv[p] \leftarrow accumulator + ´100000$;   { name p now is defined to equal $accumulator$ }
  end;

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.

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") ∧ (*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 @f 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*, *reswitch*;
 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** *bal* = 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*(*text_ptr*);
 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** $a \leftarrow id\_lookup(normal);$ $app\_repl((a$ **div** $´400) + ´200);$ $a \leftarrow a$ **mod** $´400;$
  end;
module_name: **if** $t \neq module\_name$ **then goto** *done*
  else begin $app\_repl((cur\_module$ **div** $´400) + ´250);$ $a \leftarrow cur\_module$ **mod** $´400;$
    end;
verbatim: ⟨Copy verbatim string from the buffer to *tok_mem* 169⟩;
definition, *format*, *begin_Pascal*: **if** $t \neq module\_name$ **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 $b \leftarrow "´";$
  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 $err\_print(´!\_String\_didn´´t\_end´);$ $buffer[loc] \leftarrow "´";$ $buffer[loc + 1] \leftarrow 0;$
      end;
    $b \leftarrow buffer[loc];$ $incr(loc);$
    if $b = "´"$ **then**
      begin if $buffer[loc] \neq "´"$ **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] ← "@";
reswitch: **if** *buffer*[*loc*] = "@" **then**
    begin if *loc* < *limit* **then**
      if *buffer*[*loc* + 1] = "@" **then**
        begin *app_repl*("@"); *loc* ← *loc* + 2; **goto** *reswitch*;
        end;
    end
  else begin *app_repl*(*buffer*[*loc*]); *incr*(*loc*); **goto** *reswitch*;
    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* ≠ "=") ∧ (*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 $text\_link[last\_unnamed] \leftarrow cur\_repl\_text$; $last\_unnamed \leftarrow cur\_repl\_text$;

    end

  else if $equiv[p] = 0$ **then** $equiv[p] \leftarrow cur\_repl\_text$ { first module of this name }

    else begin $p \leftarrow equiv[p]$;

      while $text\_link[p] < module\_flag$ **do** $p \leftarrow text\_link[p]$;   { find end of list }

      $text\_link[p] \leftarrow cur\_repl\_text$;

      end;

  $text\_link[cur\_repl\_text] \leftarrow module\_flag$;   { mark this replacement text as a nonmacro }

This code is used in section 175.

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 $ddt \leq 0$; thus you type $2n + 1$ integers, ending with zero or a negative number. Then control either passes to the breakpoint, allowing you to look at and/or change variables (if you typed zero), or to exit the routine (if you typed a negative value).

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

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

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

   debug procedure *debug_help*;   { routine to display various things }
   label *breakpoint*, *exit*;
   var *k*: *integer*;   { index into various arrays }
   begin *incr*(*debug_skipped*);
   if *debug_skipped* < *debug_cycle* **then return**;
   debug_skipped ← 0;
   loop begin *print_nl*(´#´); *update_terminal*;   { prompt }
     read(*term_in*, *ddt*);   { read a debug-command code }
     if *ddt* < 0 **then return**
     else if *ddt* = 0 **then**
         begin goto *breakpoint*; @\   { go to every label at least once }
       breakpoint: *ddt* ← 0; @\
         end
       else begin *read*(*term_in*, *dd*);
         case *ddt* **of**
         1: *print_id*(*dd*);
         2: *print_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

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 *ii* ← 0 **to** *zz* − 1 **do** *max_tok_ptr*[*ii*] ← *tok_ptr*[*ii*];
 tats
 ⟨Phase II: Output the contents of the compressed tables 112⟩;
end_of_TANGLE: **if** *string_ptr* > 256 **then** ⟨Finish off the string pool file 184⟩;
 stat ⟨Print statistics about memory usage 186⟩; **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* − 256 : 1, ´␣strings␣written␣to␣string␣pool␣file.´); *write*(*pool*, ´*´);
 for *ii* ← 1 **to** 9 **do**
  begin *out_buf*[*ii*] ← *pool_check_sum* **mod** 10; *pool_check_sum* ← *pool_check_sum* **div** 10;
  end;
 for *ii* ← 9 **downto** 1 **do** *write*(*pool*, *xchr*["0" + *out_buf*[*ii*]]);
 write_ln(*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_nl(´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);
 if *phase_one* **then**
  for *ii* ← 0 **to** *zz* − 1 **do** *max_tok_ptr*[*ii*] ← *tok_ptr*[*ii*];
 print(´␣bytes,␣´, *max_tok_ptr*[0] : 1);
 for *ii* ← 1 **to** *zz* − 1 **do** *print*(´+´, *max_tok_ptr*[*ii*] : 1);
 print(´␣tokens.´);
This code is used in section 182.

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 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 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.