Appendix E

The TANGLE processor

(Version 4.6)

Introduction ................................................................. 1 126
The character set ............................................................ 11 127
Input and output ............................................................ 19 128
Reporting errors to the user ................................................. 29 130
Data structures ............................................................... 37 131
Searching for identifiers ................................................. 50 132
Searching for module names ............................................ 65 135
Tokens ................................................................. 70 135
Stacks for output .......................................................... 77 135
Producing the output ...................................................... 94 138
The big output switch ..................................................... 112 139
Introduction to the input phase ......................................... 123 141
Inputting the next token ................................................ 143 141
Scanning a numeric definition ............................................ 156 141
Scanning a macro definition ............................................. 163 142
Scanning a module ........................................................ 171 143
Debugging ............................................................... 179 145
The main program ......................................................... 182 146
System-dependent changes .................................................. 188 147
Index ................................................................. 201 150

March 21, 2022 at 18:47
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.

\begin{verbatim}
define my_name ≡ 'tangle'
define banner ≡ 'This is TANGLE, Version 4.6'
\end{verbatim}

2* The program begins with a fairly normal header, made up of pieces that will mostly be filled in later. The WEB input comes from files web_file and change_file, 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.

\begin{verbatim}
⟨Compiler directives 4⟩
\end{verbatim}

\begin{verbatim}
program TANGLE(web_file, change_file, Pascal_file, pool);
const (Constants in the outer block 8*)
type (Types in the outer block 11)
var (Globals in the outer block 9)
  ⟨Define parse_arguments 188*⟩
procedure initialize;
  var (Local variables for initialization 16)
    begin kpse_set_program_name(argv[0], my_name); parse_arguments; ⟨Set initial values 10⟩
  end;
\end{verbatim}

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

\begin{verbatim}
⟨Constants in the outer block 8*⟩≡
  buf_size = 1000; { maximum length of input line }
  max_bytes = 65535; { 1/ww times the number of bytes in identifiers, strings, and module names; must be less than 65536 }
  max_toks = 65535;
    { 1/zz times the number of bytes in compressed Pascal code; must be less than 65536 }
  max_names = 10239; { number of identifiers, strings, module names; must be less than 10240 }
  max_texts = 10239; { number of replacement texts, must be less than 10240 }
  hash_size = 8501; { 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 = 100; { number of simultaneous levels of macro expansion }
  max_id_length = 50; { long identifiers are chopped to this length, which must not exceed line_length }
  def_unambig_length = 32; { identifiers must be unique if chopped to this length }
\end{verbatim}

This code is used in section 2*.
The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, so WEB assumes that it is being used with a Pascal whose character set contains at least the characters of standard ASCII as listed above. Some Pascal compilers use the original name `char` for the data type associated with the characters in text files, while other Pascals consider `char` to be a 64-element subrange of a larger data type that has some other name.

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

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

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

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

WEB’s character set is essentially identical to \TeX{}’s, even with respect to characters less than ‘\40. Changes to the present module will make WEB more friendly on computers that have an extended character set, so that one can type things like ≠ instead of <>. 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.)
Terminal output is done by writing on file `term_out`, which is assumed to consist of characters of type `text_char`:

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

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

⟨Set initial values 10⟩ +≡
{ Nothing need be done for C. }

The `update_terminal` procedure is called when we want to make sure that everything we have output to the terminal so far has actually left the computer’s internal buffers and been sent.

```plaintext
define update_terminal ≡ fflush(term_out)  { empty the terminal output buffer}
```

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.

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

The following code opens `Pascal_file`. Opening `pool` will be deferred until section 64. 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⟩ +≡
rewriting(Pascal_file, pascal_name);
The `input ln` procedure brings the next line of input from the specified file into the `buffer` array and returns the value `true`, unless the file has already been entirely read, in which case it returns `false`. The conventions of TeX are followed; i.e., `ASCII_code` numbers representing the next line of the file are input into `buffer[0]`, `buffer[1]`, ..., `buffer[limit − 1]`; trailing blanks are ignored; and the global variable `limit` is set to the length of the line. The value of `limit` must be strictly less than `buf size`.

We assume that none of the `ASCII_code` values of `buffer[j]` for `0 ≤ j < limit` is equal to `0`, `177`, `line_feed`, `form_feed`, or `carriage_return`.

```plaintext
function input ln(var f: text_file): boolean; { inputs a line or returns false }
  var final_limit: 0 .. buf size; { limit without trailing blanks }
  begin
    limit ← 0; final_limit ← 0;
    if eof(f) then input ln ← false
    else begin
      while ¬eoln(f) do
        begin
          buffer[limit] ← xord[getc(f)]; incr(limit);
          if buffer[limit − 1] ≠ " " then final_limit ← limit;
          if limit = buf size then
            begin
              while ¬eoln(f) do vgetc(f);
              decr(limit); { keep buffer[buf size] empty }
              if final_limit > limit then final_limit ← limit;
              print nl(´\texttt{Input line too long}´); loc ← 0; error;
            end;
            end;
          read ln(f); limit ← final_limit; input ln ← true;
        end;
      end;
    end;
```
The `jump_out` procedure just cuts across all active procedure levels and jumps out of the program.

```plaintext
define jump_out ≡ uexit(1)
define fatal_error(#) ≡
  begin new_line; write(stderr,#); error; mark_fatal; jump_out;
  end
```
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; T\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 $zz = 1$.)

\begin{verbatim}
define $ww = 3$  \{ we multiply the byte capacity by approximately this amount \}
define $zz = 5$  \{ we multiply the token capacity by approximately this amount \}
\end{verbatim}

\begin{verbatim}
<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 integer;  \{ info corresponding to names \}
<text\_link>: array [0..max\_texts] of sixteen\_bits;  \{ relates replacement texts \}
\end{verbatim}

Four types of identifiers are distinguished by their \ ilk:\n
\begin{verbatim}
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 \}
define parametric2 = 4  \{ second type of parametric macros have this \ ilk \}
\end{verbatim}
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) +≡

\begin{align*}
\text{id_first} & : 0 \ldots \text{buf_size}; \quad \{ \text{where the current identifier begins in the buffer} \} \\
\text{id_loc} & : 0 \ldots \text{buf_size}; \quad \{ \text{just after the current identifier in the buffer} \} \\
\text{double_chars} & : 0 \ldots \text{buf_size}; \quad \{ \text{correction to length in case of strings} \}
\end{align*}

\begin{align*}
\text{hash}, \text{chop_hash} & : \text{array}[0 \ldots \text{hash_size}] \text{ of sixteen_bits}; \quad \{ \text{heads of hash lists} \} \\
\text{chopped_id} & : \text{array}[0 \ldots \text{max_id_length}] \text{ of ASCII code}; \quad \{ \text{chopped identifier} \}
\end{align*}

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*, *parametric*, or *parametric2*.

function *id_lookup*(t : eight_bits): name_pointer; \{ finds current identifier \}

\begin{verbatim}
  label found, not_found;
  var c: eight_bits; \{ byte being chopped \}
    i: 0 .. \text{buf_size}; \{ index into \text{buffer} \}
    h: 0 .. \text{hash_size}; \{ hash code \}
    k: 0 .. \text{max_bytes}; \{ index into \text{byte_mem} \}
    w: 0 .. \text{ww} - 1; \{ segment of \text{byte_mem} \}
    l: 0 .. \text{buf_size}; \{ length of the given identifier \}
    p, q: name_pointer; \{ where the identifier is being sought \}
    s: 0 .. \text{max_id_length}; \{ index into \text{chopped_id} \}

  begin l ← id_loc – id_first; \{ compute the length \}
    \langle Compute the hash code h 54; \rangle
    \langle Compute the name location p 55; \rangle
    if (p = name_ptr) ∨ (t ≠ normal) then \langle Update the tables and check for possible errors 57; \rangle
      id_lookup ← p;
    end;
\end{verbatim}
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 \(\text{chopped} \_\text{id}\) to serve as a convenient sentinel.

\[
\langle \text{Compute the secondary hash code } h \text{ and put the first characters into the auxiliary array } \text{chopped} \_\text{id} \rangle \equiv \\
\text{begin } i \leftarrow \text{id} \_\text{first}; \ s \leftarrow 0; \ h \leftarrow 0; \text{ while } (i < \text{id} \_\text{loc}) \land (s < \text{unambig} \_\text{length}) \text{ do } \\
\text{ begin if } (\text{buffer}[i] \neq "\_") \lor (\text{allow} \_\text{underlines} \land \neg \text{strict} \_\text{mode}) \text{ then } \\
\quad \text{ begin if } (\text{strict} \_\text{mode} \lor \text{force} \_\text{uppercase}) \land (\text{buffer}[i] \geq "a") \text{ then } \text{chopped} \_\text{id}[s] \leftarrow \text{buffer}[i] - 40 \\
\quad \text{ else if } (\neg \text{strict} \_\text{mode} \land \text{force} \_\text{lowercase}) \land (\text{buffer}[i] \geq "a") \land (\text{buffer}[i] \leq "z") \text{ then } \\
\qquad \text{chopped} \_\text{id}[s] \leftarrow \text{buffer}[i] + 40 \\
\text{ else } \text{chopped} \_\text{id}[s] \leftarrow \text{buffer}[i]; \\
\quad h \leftarrow (h + h + \text{chopped} \_\text{id}[s]) \mod \text{hash} \_\text{size}; \text{ incr}(s); \\
\text{ end; } \\
\text{ incr}(i); \\
\text{ end; } \\
\text{chopped} \_\text{id}[s] \leftarrow 0; \text{ end }
\]

This code is used in section 57.

\[
\langle \text{Check if } q \text{ conflicts with } p \rangle \equiv \\
\text{begin } k \leftarrow \text{byte} \_\text{start}[q]; \ s \leftarrow 0; \ w \leftarrow q \mod \text{ww}; \text{ while } (k < \text{byte} \_\text{start}[q + \text{ww}]) \land (s < \text{unambig} \_\text{length}) \text{ do } \\
\text{ begin } c \leftarrow \text{byte} \_\text{mem}[w, k]; \\
\text{ if } c \neq "\_" \lor (\text{allow} \_\text{underlines} \land \neg \text{strict} \_\text{mode}) \text{ then } \\
\quad \text{ begin if } (\text{strict} \_\text{mode} \lor \text{force} \_\text{uppercase}) \land (c \geq "a") \text{ then } c \leftarrow c - 40 \\
\quad \text{ else if } (\neg \text{strict} \_\text{mode} \land \text{force} \_\text{lowercase}) \land (c \geq "a") \land (c \leq "z") \text{ then } c \leftarrow c + 40; \\
\quad \text{ if } \text{chopped} \_\text{id}[s] \neq c \text{ then goto not_found; } \\
\quad \text{ incr}(s); \\
\quad \text{ incr}(k); \\
\text{ end; } \\
\text{ end; } \\
\text{ if } (k = \text{byte} \_\text{start}[q + \text{ww}]) \land (\text{chopped} \_\text{id}[s] \neq 0) \text{ then goto not_found; } \\
\text{ print}_\text{nl}("\_\_\_\text{Identifier} \_\text{conflict} \_\text{with} \_\_\_\text{conflict}"); \\
\text{ for } k \leftarrow \text{byte} \_\text{start}[q] \text{ to } \text{byte} \_\text{start}[q + \text{ww}] - 1 \text{ do } \text{print}(\text{chr}[\text{byte} \_\text{mem}[w, k]]); \\
\text{ error; } q \leftarrow 0; \text{ only one conflict will be printed, since equiv[0] = 0 } \\
\text{ not_found: end }
\]

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.

\[ \text{define check\_sum\_prime} \equiv 3777777667 \ \{ 2^{29} - 73 \} \]

\[
\text{begin}\ \text{ilk}[p] \leftarrow \text{numeric}; \ \{ \text{strings are like numeric macros} \}
\]

\[
\text{if } l - \text{double\_chars} = 2 \ \{ \text{this string is for a single character} \}
\]

\[
equiv[p] \leftarrow \text{buffer}[\text{id\_first} + 1] + \text{`10000000000}
\]

\[
\text{else begin} \ \{ \text{Avoid creating empty pool files.} \}
\]

\[
\text{if } \text{string\_ptr} = 256 \ \text{then } \text{rewritebin(pool, pool\_name)};
\]

\[
equiv[p] \leftarrow \text{string\_ptr} + \text{`10000000000}; \ l \leftarrow l - \text{double\_chars} - 1;
\]

\[
\text{if } l > 99 \ \text{then } \text{err\_print(\text{`![Preprocessed\_string\_is\_too\_long']})};
\]

\[
\text{incr(string\_ptr); write(pool, xchr[\text{"0" + l div 10}], xchr[\text{"0" + l mod 10}]);} \ \{ \text{output the length} \}
\]

\[
\text{pool\_check\_sum} \leftarrow \text{pool\_check\_sum} + \text{pool\_check\_sum} + l;
\]

\[
\text{while pool\_check\_sum > check\_sum\_prime do } \text{pool\_check\_sum} \leftarrow \text{pool\_check\_sum} - \text{check\_sum\_prime};
\]

\[
i \leftarrow \text{id\_first} + 1;
\]

\[
\text{while } i < \text{id\_loc} \text{ do }
\]

\[
\text{begin write(pool, xchr[buffer[i]]);} \ \{ \text{output characters of string} \}
\]

\[
\text{pool\_check\_sum} \leftarrow \text{pool\_check\_sum} + \text{pool\_check\_sum} + \text{buffer}[i];
\]

\[
\text{while pool\_check\_sum > check\_sum\_prime do } \text{pool\_check\_sum} \leftarrow \text{pool\_check\_sum} - \text{check\_sum\_prime};
\]

\[
\text{if } (\text{buffer}[i] = \text{"\"\"\") \lor (\text{buffer}[i] = \text{"@\") then } i \leftarrow i + 2
\]

\[
\{ \text{omit second appearance of doubled character} \}
\]

\[
\text{else incr}(i);
\]

\[
\text{end};
\]

\[
\text{write\_ln(pool)};
\]

\[
\text{end};
\]

\[
\text{end}
\]

This code is used in section 61.
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.

```plaintext
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) \( \lor \) (ilk[cur_name] = parametric2) then
        ⟨ Remove a parameter from the parameter stack ⟩;
      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 }
      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;
```

This code is used in section 87.

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

```plaintext
begin case ilk[a] of
  normal: begin cur_val ← a; a ← identifier;
          end;
  numeric: begin cur_val ← equiv[a] − '10000000000; a ← number;
          end;
  simple: begin push_level(a); goto restart;
          end;
  parametric, parametric2: 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.
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.

\[
\text{⟨Put a parameter on the parameter stack, or goto restart if error occurs 90*⟩ ≡}
\]

\[
\text{while (cur-byte = cur-end) \& (stack.ptr > 0) do pop.level;}
\]

\[
\text{if (stack.ptr = 0) \lor ((ilk[a] = parametric) \& (tok_mem[zo,}
\text{ cur-byte] \neq "(")) \lor ((ilk[a] = parametric2) \& (tok_mem[zo, cur-byte] \neq "[")) then}
\]

\[
\text{begin print.nl(‘!No parameter given for‘); print_id(a); error; goto restart;}
\]

\[
\text{end;}
\]

\[
\text{⟨Copy the parameter into tok_mem 93*⟩;}
\]

\[
equiv[name.ptr] \leftarrow text.ptr; \ ilk[name.ptr] \leftarrow \text{simple; } w \leftarrow name.ptr \ mod \ ww; k \leftarrow byte.ptr[w];
\]

\[
\text{debug if } k = \text{max.bytes then overflow(‘byte.memory‘);}
\]

\[
\text{byte_mem[w,k] \leftarrow "#"; incr(k); byte.ptr[w] \leftarrow k;}
\]

\[
\text{gubed \{ this code has set the parameter identifier for debugging printouts \}}
\]

\[
\text{if name.ptr > max.names - ww then overflow(‘name‘);}
\]

\[
\text{byte_start[name.ptr + ww] \leftarrow k; incr(name.ptr);}
\]

\[
\text{if text.ptr > max.texts - zz then overflow(‘text‘);}
\]

\[
\text{text_link[text.ptr] \leftarrow 0; tok_start[text.ptr + zz] \leftarrow tok_ptr[z]; incr(text.ptr); z \leftarrow text.ptr \ mod \ zz}
\]

This code is used in section 89*.
Similarly, a `param` token encountered as we copy a parameter is converted into a simple macro call for `name_ptr − 1`. Some care is needed to handle cases like `macro(#; print(´#´));` the # token will have been changed to `param` outside of strings, but we still must distinguish ‘real’ parentheses from those in strings.

```define app_repl(#) ≡
   begin if tok_ptr[z] = max_toks then overflow(´token´);
   tok_mem[z, tok_ptr[z]] ← #; incr(tok_ptr[z]);
   end

   { Copy the parameter into tok_mem 93* } ≡
bal ← 1; incr(cur_byte); { skip the opening ′(′ or ′[′ }
loop begin b ← tok_mem[zo, cur_byte]; incr(cur_byte);
   if b = param then store_two_bytes(name_ptr + 7777)
   else begin if b ≥ ′200 then
      begin app_repl(b); b ← tok_mem[zo, cur_byte]; incr(cur_byte);
      end
   else case b of
      ′(′: if ilk[a] = parametric then incr(bal);
      ′) ′: if ilk[a] = parametric then
         begin decr(bal);
         if bal = 0 then goto done;
         end;
      ′[′: if ilk[a] = parametric2 then incr(bal);
      ′] ′: if ilk[a] = parametric2 then
         begin decr(bal);
         if bal = 0 then goto done;
         end;
      ′´ ′: repeat app_repl(b); b ← tok_mem[zo, cur_byte]; incr(cur_byte);
         until b = ′´; { copy string, don’t change bal }
      othercases do nothing
   endcases
   app_repl(b);
end;
end;
done:
```

This code is used in section 90*.
(Contribution is * or / or DIV or MOD 105*) \(\equiv\)

\[
(t = \text{ident}) \land (v = 3) \land ((\text{out}\_\text{contrib}[1] = "D") \land (\text{out}\_\text{contrib}[2] = "I") \land (\text{out}\_\text{contrib}[3] = "V").\]

\[
((\text{out}\_\text{contrib}[1] = "d") \land (\text{out}\_\text{contrib}[2] = "i") \land (\text{out}\_\text{contrib}[3] = "v").\]

\[
((\text{out}\_\text{contrib}[1] = "M") \land (\text{out}\_\text{contrib}[2] = "O") \land (\text{out}\_\text{contrib}[3] = "D").\]

\[
((\text{out}\_\text{contrib}[1] = "m") \land (\text{out}\_\text{contrib}[2] = "o") \land (\text{out}\_\text{contrib}[3] = "d").\]

\[
(t = \text{misc}) \land ((v = "*") \lor (v = "/")\]

This code is used in section 104.

(If previous output was DIV or MOD, goto bad case 110) \(\equiv\)

\[
\text{if } (\text{out}_\text{ptr} = \text{break}_\text{ptr} + 3) \lor ((\text{out}_\text{ptr} = \text{break}_\text{ptr} + 4) \land (\text{out}_\text{buf}[\text{break}_\text{ptr}] = ".")\text{ then}
\]

\[
\text{if } ((\text{out}_\text{buf}[\text{out}_\text{ptr} - 3] = "D") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 2] = "I") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 1] = "V")\lor
\]

\[
((\text{out}_\text{buf}[\text{out}_\text{ptr} - 3] = "d") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 2] = "i") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 1] = "v")\lor
\]

\[
((\text{out}_\text{buf}[\text{out}_\text{ptr} - 3] = "M") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 2] = "O") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 1] = "D")\lor
\]

\[
((\text{out}_\text{buf}[\text{out}_\text{ptr} - 3] = "m") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 2] = "o") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 1] = "d")\]

\text{then goto bad case}

This code is used in section 107.
114* (Cases like $\ll$ and $\equiv$ 114*)
    end;
    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.

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

define up_to(#) ≡ # − 24, # − 23, # − 22, # − 21, # − 20, # − 19, # − 18, # − 17, # − 16, # − 15, # − 14, # − 13,
    # − 12, # − 11, # − 10, # − 9, # − 8, # − 7, # − 6, # − 5, # − 4, # − 3, # − 2, # − 1, #

(Cases related to identifiers 116*)

"A", up_to("Z"): begin if force_lowercase then out_contrib[1] ← cur_char + '40
      else out_contrib[1] ← cur_char;
      send_out(ident, 1);
    end;
"a", up_to("z"): begin if force_uppercase then out_contrib[1] ← cur_char − '40
      else out_contrib[1] ← cur_char;
      send_out(ident, 1);
    end;
identifier: begin k ← 0; j ← byte_start[cur_val]; w ← cur_val mod ww;
      while (k < max_id_length) ∧ (j < byte_start[cur_val + ww]) do
        begin incr(k); out_contrib[k] ← byte_mem[w, j]; incr(j);
        if force_uppercase ∧ (out_contrib[k] ≥ "a") then out_contrib[k] ← out_contrib[k] − '40
        else if force_lowercase ∧ (out_contrib[k] ≤ "z") then out_contrib[k] ← out_contrib[k] + '40
          else if ¬allow_underlines ∧ (out_contrib[k] = ".") then decr(k);
        end;
        send_out(ident, k);
      end;
This code is used in section 113.
In order to encourage portable software, TANGLE complains if the constants get dangerously close to the largest value representable on a 32-bit computer ($2^{31} - 1$).

\[\text{define digits} \equiv \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\]

(Cases related to constants, possibly leading to \textit{get\_fraction} or \textit{reswitch} 119*)

\[\text{define digits} \equiv \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\]

\begin{verbatim}
\texttt{define digits} \equiv \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
\end{verbatim}

\begin{verbatim}
〈Cases related to constants, possibly leading to \textit{get\_fraction} or \textit{reswitch} 119*〉 \equiv
\end{verbatim}

\begin{verbatim}
digits: begin
  n ← 0;
  repeat cur\_char ← cur\_char − "0";
    if n ≥ 1463146314 then err\_print("!\_Constant\_too\_big")
    else n ← 10 * n + cur\_char;
    cur\_char ← get\_output;
  until (cur\_char > "9") ∨ (cur\_char < "0");
  send\_val(n); k ← 0;
  if cur\_char = "e" then cur\_char ← "E";
  if cur\_char = "E" then goto get\_fraction
  else goto reswitch;
end;

check\_sum: send\_val(pool\_check\_sum);

octal: begin
  n ← 0; cur\_char ← "0";
  repeat cur\_char ← cur\_char − "0";
    if n ≥ 10000000000 then err\_print("!\_Constant\_too\_big")
    else n ← 8 * n + cur\_char;
    cur\_char ← get\_output;
  until (cur\_char > "7") ∨ (cur\_char < "0");
  send\_val(n); goto reswitch;
end;

hex: begin
  n ← 0; cur\_char ← "0";
  repeat if cur\_char ≥ "A" then cur\_char ← cur\_char + 10 − "A"
    else cur\_char ← cur\_char − "0";
    if n ≥ 40000000 then err\_print("!\_Constant\_too\_big")
    else n ← 16 * n + cur\_char;
    cur\_char ← get\_output;
  until (cur\_char > "F") ∨ (cur\_char < "0") ∨ ((cur\_char > "9") ∧ (cur\_char < "A"));
  send\_val(n); goto reswitch;
end;

number: send\_val(cur\_val);

"." : begin
  k ← 1; out\_contrib[1] ← "."; cur\_char ← get\_output;
  if cur\_char = "." then
    begin out\_contrib[2] ← "."; send\_out(str, 2);
      end
  else if (cur\_char ≥ "0") ∧ (cur\_char ≤ "9") then goto get\_fraction
    else begin send\_out(misc, "."); goto reswitch;
      end;
end;
\end{verbatim}

This code is used in section 113.
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.

```pascal
define add_in(#) ≡
    begin accumulator ← accumulator + next_sign * (#); next_sign ← +1;
end

procedure scan_numeric(p : name_pointer); { defines numeric macros }
    label reswitch, done;
    var accumulator: integer; { accumulates sums }
        next_sign: −1..+1; { sign to attach to next value }
        q: name_pointer; { points to identifiers being evaluated }
        val: integer; { constants being evaluated }
    begin ⟨Set accumulator to the value of the right-hand side 158*⟩;
        if abs(accumulator) ≥ '100000000000 then
            begin err_print(´!_Value_too_big;_´, accumulator : 1); accumulator ← 0;
                end;
            equiv[p] ← accumulator + '100000000000; { 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] − '100000000000);
        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*. 

---

The page contains an introduction to the input phase, focusing on the evaluation of numeric expressions in a programming context. Variables such as `accumulator` and `next_sign` are used to accumulate values and handle signs, respectively. The code snippet provides a Pascal-like definition for handling numeric expressions, including error checking for values too big and formatting errors. The procedure `scan_numeric` is defined to scan and handle different types of numeric inputs, including decimal, octal, and hexadecimal constants, as well as identifiers. The code snippet also includes error handling for cases where the value is too big or when semicolons are omitted in numeric definitions.
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 the right parentheses }
  begin bal ← 0;
  loop begin continue: a ← get_next;
    case a of
      "(": if t = parametric then incr(bal);
      ")": if t = parametric then
        if bal = 0 then err_print(´!\_Extra\_]´)
        else decr(bal);
      "]": if t = parametric2 then incr(bal);
      "[": if t = parametric2 then
        if bal = 0 then err_print(´!\_Extra\_]´)
        else decr(bal);
      ": ( Copy a string from the buffer to tok_mem 168 );
      ": if (t = parametric) ∨ (t = parametric2) 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
    if t = parametric 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
    else 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
  end

This code is used in section 165*.
\begin{verbatim}
173*  (Scan the definition part of the current module 173*) \equiv
next_control \leftarrow 0;
loop begin continue: while next_control \leq format do
  begin next_control \leftarrow \text{skip\_ahead};
  if next_control = \text{module\_name} then
    begin  \{ we want to scan the module name too \}
      loc \leftarrow loc - 2; next_control \leftarrow \text{get\_next};
    end;
  end;
if next_control \neq \text{definition} then goto done;
next_control \leftarrow \text{get\_next};  \{ get identifier name \}
if next_control \neq \text{identifier} then
  begin
    \text{err\_print}(`Definition flushed, must start with \text{identifier} of length \geq 1');
go to continue;
  end;
next_control \leftarrow \text{get\_next};  \{ get token after the identifier \}
if next_control = '=' then
  begin
    scan\_numeric(id\_lookup(numeric)); goto continue;
  end
else if next_control = \text{equivalence\_sign} then
  begin
    define\_macro(simple); goto continue;
  end
else \{ If the next text is `(#)==' or `[#]==', call define\_macro and goto continue 174*\};
\text{err\_print}(`Definition flushed since it starts badly');
end:
done:
\end{verbatim}
This code is used in section 172.
174* (If the next text is ‘(#)==‘ or ‘[#]==‘, 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;
else 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(parametric2); goto continue;
        end;
    end;
  end;
end
This code is used in section 173*.
179° Debugging. The Pascal debugger with which TANGLE was developed allows breakpoints to be set, and variables can be read and changed, but procedures cannot be executed. Therefore a 'debug_help' procedure has been inserted in the main loops of each phase of the program; when ddt and dd are set to appropriate values, symbolic printouts of various tables will appear.

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

\[
\begin{align*}
\text{define } & \text{term_in } \equiv \text{stdin} \\
\langle \text{Globals in the outer block 9} \rangle & + \equiv \\
\text{debug } & \text{trouble_shooting: boolean; } \{ \text{is debug_help wanted?} \} \\
\text{ddt: integer; } & \{ \text{operation code for the debug_help routine} \} \\
\text{dd: integer; } & \{ \text{operand in procedures performed by debug_help} \} \\
\text{debug_cycle: integer; } & \{ \text{threshold for debug_help stopping} \} \\
\text{debug_skipped: integer; } & \{ \text{we have skipped this many debug_help calls} \} \\
& \text{gubed}
\end{align*}
\]

180° The debugging routine needs to read from the user’s terminal.

\[
\langle \text{Set initial values 10} \rangle + \equiv \\
\text{debug } & \text{trouble_shooting } \leftarrow \text{true}; \text{debug_cycle } \leftarrow 1; \text{debug_skipped } \leftarrow 0; \\
& \text{trouble_shooting } \leftarrow \text{false}; \text{debug_cycle } \leftarrow 99999; \{ \text{use these when it almost works} \} \\
& \text{gubed}
\]
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; \langle Initialize the input system 134 \rangle;  
print(banner); \{ print a \textquotedblright banner line\}  
print\_ln(version\_string); \langle \text{Phase I: Read all the user\textquotesingle s text and compress it into tok\_mem 183} \rangle;  
stat for ii ← 0 to zz − 1 do max\_tok\_ptr[ii] ← tok\_ptr[ii];  
tats \langle \text{Phase II: Output the contents of the compressed tables 112} \rangle;  
if string\_ptr > 256 then \langle \text{Finish off the string pool file 184} \rangle;  
stat \langle \text{Print statistics about memory usage 186} \rangle; tats  
\{ here files should be closed if the operating system requires it \}  
\langle \text{Print the job history 187} \};  
new\_line;  
if (history \neq spotless) \land (history \neq harmless\_message) \textbf{then} uexit(1)  
else uexit(0);  
end.
188* System-dependent changes. Parse a Unix-style command line.

define argument_is(#) ≡ (strcmp(long_options[option_index].name, #) = 0)
(Define parse_arguments 188*)

procedure parse_arguments;
const n_options = 10;  { Pascal won’t count array lengths for us. }
var long_options: array [0 .. n_options] of getopt_struct;
getopt_return_val: integer; option_index: c_int_type; current_option: 0 .. n_options; len: integer;
begin {Define the option table 190*};
unambig_length ← def_unambig_length;
repeat getopt_return_val ← getopt_long_only(argc, argv, "", long_options, address_of (option_index));
  if getopt_return_val = −1 then
    begin do nothing;  { End of arguments; we exit the loop below. }
    end
  else if getopt_return_val = "?" then
    begin usage(my_name);
    end
  else if argument_is(´help´) then
    begin usage_help(TANGLE_HELP, nil);
    end
  else if argument_is(´version´) then
    begin print_version_and_exit(banner, nil, ´D.E.Knuth´, nil);
    end
  else if argument_is(´mixedcase´) then
    begin force_uppercase ← false; force_lowercase ← false;
    end
  else if argument_is(´uppercase´) then
    begin force_uppercase ← true; force_lowercase ← false;
    end
  else if argument_is(´lowercase´) then
    begin force_uppercase ← false; force_lowercase ← true;
    end
  else if argument_is(´underlines´) then
    begin allow_underlines ← true;
    end
  else if argument_is(´strict´) then
    begin strict_mode ← true;
    end
  else if argument_is(´loose´) then
    begin strict_mode ← false;
    end
  else if argument_is(´length´) then
    begin len ← atoi(optarg);
    if (len ≤ 0) ∨ (len > max_id_length) then len ← max_id_length;
    unambig_length ← len;
    end;  { Else it was a flag; getopt has already done the assignment. }
until getopt_return_val = −1;
(Handle file name arguments 189*)
end;
This code is used in section 2*. 
Now *optind* is the index of first non-option on the command line.

```pascal
if (optind + 1 > argc) \lor (optind + 3 < argc) then
  begin writeLn(stderr, my_name, `.\i\Need\_one\_to\_three\_file\_arguments.\`); usage(my_name);
  end;  \{ Supply ".web" and ".ch" extensions if necessary. \}
web_name ← extend_filename(cmdline(optind), `.web`);
if optind + 2 ≤ argc then
  begin \{ `-` is shortcut for an empty changefile. \}
    if strcmp(char_to_string(`-`), cmdline(optind + 1)) \neq 0 then
      chg_name ← extend_filename(cmdline(optind + 1), `ch`);
    end;
if optind + 3 = argc then
  begin \{ User has provided an explicit Pascal output file, possibly with path. \}
    pascal_name ← extend_filename(cmdline(optind + 2), char_to_string(`p`));
    pool_name ← extend_filename(remove_suffix(pascal_name), `pool`);
  end \\
else begin \{ Change ".web" to ".p" and ".pool" and use the current directory. \}
    pascal_name ← basename_change_suffix(web_name, ".web", ".p");
    pool_name ← basename_change_suffix(web_name, ".web", ".pool");
  end;
```

This code is used in section 188*.

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

```pascal
\begin{align*}
\text{(Define the option table 190*)} & \iff
  \text{current_option} ← 0; \text{long_options[current_option].name} ← `help`; \\
  \text{long_options[current_option].has_arg} ← 0; \text{long_options[current_option].flag} ← 0; \\
  \text{long_options[current_option].val} ← 0; \text{incr(current_option)};
\end{align*}
```

This code is used in section 188*.

Another of the standard options.

```pascal
\begin{align*}
\text{(Define the option table 190*)} & \iff
  \text{long_options[current_option].name} ← `version`; \text{long_options[current_option].has_arg} ← 0; \\
  \text{long_options[current_option].flag} ← 0; \text{long_options[current_option].val} ← 0; \text{incr(current_option)};
\end{align*}
```

Use all mixed case.

```pascal
\begin{align*}
\text{(Define the option table 190*)} & \iff
  \text{long_options[current_option].name} ← `mixedcase`; \text{long_options[current_option].has_arg} ← 0; \\
  \text{long_options[current_option].flag} ← 0; \text{long_options[current_option].val} ← 0; \text{incr(current_option)};
\end{align*}
```

Use all uppercase.

```pascal
\begin{align*}
\text{(Define the option table 190*)} & \iff
  \text{long_options[current_option].name} ← `uppercase`; \text{long_options[current_option].has_arg} ← 0; \\
  \text{long_options[current_option].flag} ← 0; \text{long_options[current_option].val} ← 0; \text{incr(current_option)};
\end{align*}
```

Use all lowercase.

```pascal
\begin{align*}
\text{(Define the option table 190*)} & \iff
  \text{long_options[current_option].name} ← `lowercase`; \text{long_options[current_option].has_arg} ← 0; \\
  \text{long_options[current_option].flag} ← 0; \text{long_options[current_option].val} ← 0; \text{incr(current_option)};
\end{align*}
```
§195* Allow underlines.
\[\text{Define the option table 190*} + \equiv \]
\[\text{long_options}[\text{current_option}].\text{name} \leftarrow \text{underlines}; \text{long_options}[\text{current_option}].\text{has_arg} \leftarrow 0;\]
\[\text{long_options}[\text{current_option}].\text{flag} \leftarrow 0; \text{long_options}[\text{current_option}].\text{val} \leftarrow 0; \text{incr(\text{current_option});}\]

§196* Strict comparisons.
\[\text{Define the option table 190*} + \equiv \]
\[\text{long_options}[\text{current_option}].\text{name} \leftarrow \text{strict}; \text{long_options}[\text{current_option}].\text{has_arg} \leftarrow 0;\]
\[\text{long_options}[\text{current_option}].\text{flag} \leftarrow 0; \text{long_options}[\text{current_option}].\text{val} \leftarrow 0; \text{incr(\text{current_option);}\]

§197* Loose comparisons.
\[\text{Define the option table 190*} + \equiv \]
\[\text{long_options}[\text{current_option}].\text{name} \leftarrow \text{loose}; \text{long_options}[\text{current_option}].\text{has_arg} \leftarrow 0;\]
\[\text{long_options}[\text{current_option}].\text{flag} \leftarrow 0; \text{long_options}[\text{current_option}].\text{val} \leftarrow 0; \text{incr(\text{current_option);}\]

§198* Loose comparisons.
\[\text{Define the option table 190*} + \equiv \]
\[\text{long_options}[\text{current_option}].\text{name} \leftarrow \text{length}; \text{long_options}[\text{current_option}].\text{has_arg} \leftarrow 1;\]
\[\text{long_options}[\text{current_option}].\text{flag} \leftarrow 0; \text{long_options}[\text{current_option}].\text{val} \leftarrow 0; \text{incr(\text{current_option);}\]

§199* An element with all zeros always ends the list.
\[\text{Define the option table 190*} + \equiv \]
\[\text{long_options}[\text{current_option}].\text{name} \leftarrow 0; \text{long_options}[\text{current_option}].\text{has_arg} \leftarrow 0;\]
\[\text{long_options}[\text{current_option}].\text{flag} \leftarrow 0; \text{long_options}[\text{current_option}].\text{val} \leftarrow 0;\]

§200* Global filenames.
\[\text{Globals in the outer block 9} + \equiv \]
\[\text{web_name, chg_name, pascal_name, pool_name: const c_string;}\]
\[\text{force_uppercase, force_lowercase, allow_underlines, strict_mode: boolean;}\]
\[\text{unambig_length: 0 . max_id_length;}\]
201* 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.

The following sections were changed by the change file: 1, 2, 8, 12, 17, 20, 21, 22, 24, 26, 28, 34, 38, 47, 50, 53, 58, 63, 64, 85, 89, 90, 93, 105, 110, 114, 116, 119, 157, 158, 165, 166, 173, 174, 179, 180, 182, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201.

- help: 190*
- length: 198*
- loose: 197*
- lowercase: 194*
- mixedcase: 192*
- strict: 196*
- underlines: 195*
- uppercase: 193*
- version: 191*

@d is ignored in Pascal text: 167.
@f is ignored in Pascal text: 167.
@p is ignored in Pascal text: 167.

a: 74, 87, 165*
abs: 103, 157*
accumulator: 157* 158* 159.
add_in: 157* 158*
address_of: 188*
allow_underlines: 58* 63* 116* 188* 200*
Ambiguous prefix: 69.
and_sign: 15, 114*
app: 99, 101, 102, 103, 111.
app_repl: 93* 165* 166* 167, 168, 169.
app_val: 99, 103, 111.
args: 188* 189*
argument_is: 188*
argv: 2* 188*
ASCII code: 11, 72.
ASCII_code: 11, 12* 13, 27, 28* 38* 50* 65, 94, 95, 100, 126, 139, 141, 165*
atoi: 188*
b: 87, 97, 165*
bad_case: 107, 109, 110*
bal: 87, 93* 141, 142, 165* 166*
banner: 1* 182* 188*
basename_change_suffix: 189*
begin: 3.

begin_comment: 72, 76, 121, 139, 147.
begin_Pascal: 139, 156, 158* 167, 175.
boolean: 28* 29, 124, 127, 143, 179* 200*
brace_level: 82, 83, 98, 121.
budget: 179* 181.

byte_field: 78, 79.
byte_mem: 37, 38* 39, 40, 41, 48, 49, 53* 56, 61, 63* 66, 67, 68, 69, 75, 87, 90* 113, 116*.
byte_ptr: 39, 40, 42, 61, 67, 90* 91, 186.
byte_start: 37, 38* 39, 40, 42, 49, 50* 56, 61, 63* 67, 68, 75, 78, 81, 90* 116* 143.
c: 53* 66, 69, 139, 140, 141, 145.
c_int_type: 188*

Can’t output ASCII code n: 113.
carrกะ_rate: 15, 17* 28*
Change file ended...: 130, 132, 137.
Change file entry did not match: 138.
change_changing: 125, 126, 134, 137.
change_file: 2* 23, 24* 32, 124, 126, 129, 130, 132, 137.
char: 12* 14.
char_to_string: 189*
check_break: 97, 101, 102, 103, 111.
check_change: 132, 136.
check_sum: 72, 76, 119* 139.
check_sum_prime: 64*
chg_name: 24* 189* 200*
chop_hash: 50* 52, 60, 62.
chopped_id: 50* 53* 58* 63*
chr: 12* 13, 17* 18.
char: 189*
compress: 147.
confusion: 35, 89*
const_c_string: 200*

Constant too big: 119*.
control_code: 139, 140, 143, 150.
control_text: 139, 150.
count: 69.
id_first: 50*53*54, 56, 57, 58*61, 64*143, 148, 149.
id_loc: 50*53*54, 56, 58*61, 64*143, 148, 149.
id_lookup: 50*53*54, 143, 158*167, 170, 173*.
ident: 100, 101, 102, 105*114*116*.
identifier: 86, 89*116*143, 148, 149, 158*167, 173*.

Identifier conflict...: 63*.
ignore: 139, 140, 150.
ii: 124, 138, 182*184, 186.
ilk: 37, 38*47*, 48, 50*57, 59, 60, 61, 64*, 85*, 89*, 90*, 93*158*.

Improper @ within control text: 150.
Improper numeric definition...: 159.

Incompatible module names: 66.
initialize: 2*182*.

Input ended in mid-comment: 141.
Input ended in section name: 153.
Input line too long: 28*.
input_len: 28*129, 130, 132, 136, 137.
join: 72, 101, 113, 139.
jump_out: 2*31, 34*.

kpse_open_file: 24*.
kpse_set_program_name: 2*.
kpse_web_format: 24*.
last_sign: 95, 103, 106, 107.
last_text_char: 12*18.
last_unnamed: 70, 71, 178.
left_arrow: 15, 114*147.
len: 188*.
length: 39, 55.
less: 66, 67, 68, 69.
less_or_equal: 15, 114*147.
line_feed: 15, 28*.
line_length: 8*94, 97, 100, 101, 113, 117, 118, 120, 122.
lines_dont_match: 127, 132.

link: 37, 38*39, 48, 50*55.
Long line must be truncated: 97.
longest_name: 8*, 65, 66, 69, 145, 153, 155.
loop: 6*.
mark_error: 9, 31.
mark_fatal: 9, 34*.
mark_harmless: 9, 112, 155.
max_id_length: 8*, 50*, 53*, 116*, 188*200*.
max_names: 8*, 38*, 39, 61, 67, 69, 90*.
max_texts: 8*, 38*43, 70, 90*165*.
max_tok_ptr: 44, 91, 182*, 186*.
max_toks: 8*, 38*, 44, 73, 74, 93*.
misc: 95, 96, 100, 101, 102, 105*, 107, 111, 113, 119*, 121, 122.
Missing n ) : 166*.
Missing n J : 166*.
mod: 94.
mod_field: 78, 79.
mod_lookup: 65, 66, 151, 152.
module_count: 139, 171, 172, 177, 183.
module_flag: 70, 85*, 178*.
module_name: 139, 143, 150, 156, 158*159, 164, 167, 173*175*.
module_number: 86, 87, 121.
my_name: 1*, 2*, 188*, 189*.
n: 113, 132.
n_options: 188*.
Name does not match: 69.
name_field: 78, 79.
new_line: 20*, 31, 32, 34*, 182*.
new_module: 139, 140, 145, 156, 158*167, 176, 183.
next_sign: 157*, 158*.
nil: 6.

No output was specified: 112.
No parameter given for macro: 90*
Not present: <section name>: 88.
not_equal: 15, 114*, 147.
not_found: 5, 53*, 63*.
not_sign: 15, 114*.
norm_or_id: 95, 101, 102, 107, 111.
number: 86, 89*, 119*.
numeric: 47*, 53*, 59, 64*, 89*, 158*, 173*.
octal: 72, 76, 119*, 139, 158*.
Omit semicolon in numeric def...: 158*.
open_input: 24*, 134.
optary: 188*.
opind: 189*.
option_index: 188*.
or_sign: 15, 114*.
ord: 13.
other_line: 124, 125, 134, 138.
othecases: 7.
others: 7.
out_app: 95, 102, 104, 106, 108.
out_buf_size: 8*, 31, 94, 97, 99.
out_sign: 95, 103, 104, 107, 108.
out_val: 95, 103, 104, 106, 107, 108.
output_state: 78, 79.
overflow: 36, 61, 67, 73, 84, 90*, 93*, 165*.
pack: 61.
param: 72, 76, 87, 93*, 165*.
parse_arguments: 2*, 188*.
Pascal text flushed...: 176.
Pascal_file: 2*, 25, 26*, 97.
pascal_name: 20*, 189*, 200*.
phase_one: 29*, 31, 183, 186.
pool: 2*, 25, 26*, 64*, 184.
pool_check_sum: 40, 42, 64*, 119*, 184.
pool_name: 64*, 189*, 200*.
pop_level: 85*, 87, 90*, 91.
prefix: 66, 68.
prefix_lookup: 69, 151.
Preprocessed string is too long: 64*.
preprocessed strings: 64*, 149.
prime_the_change_buffer: 128, 134, 137.
print_id: 49, 75, 88, 90*, 181.
print_len: 20*, 32, 33, 182*.
print_repl: 74, 181.
print_version_and_exit: 188*.
Program ended at brace level n: 98.
push_level: 84, 88, 89*, 92.
s: 53*, 66, 69, 157*.
r: 69*.
read: 181.
read_in: 28*.
remove_suffix: 189*.
repl_field: 78, 79.
return: 5, 6.
rewrite: 26*.
rewritebin: 64*.
s: 53*.
scan_module: 171, 172, 183.
scan_numeric: 156, 157*, 173*.
scan_repl: 164, 165*, 170, 175.
scanning_hex: 143, 144, 145, 146, 150.
Section ended in mid-comment: 142.
Section name didn't end: 154.
Section name too long: 155.
send_the_output: 112, 113.
send_val: 100, 107, 112, 119*.
set_element_sign: 15, 114*.
sign: 95, 102, 106, 108.
sign_val: 95, 102, 104, 106, 107, 108.
sign_val_set: 95, 102, 106, 108.
sixteen_bits: 37, 38*, 50*, 66, 69, 73, 74, 78, 87, 101, 165*.
skip_ahead: 140, 150, 159, 173*, 176, 183.
skip_comment: 141, 145.
Sorry, x capacity exceeded: 36.
spotless: 9, 10, 182*, 187.
stack: 78, 79, 84, 85*.
stack_ptr: 78, 79, 83, 84, 85, 87, 90, 113, 117, 118.
stack_size: 8, 79, 84.
stat: 3.
stderr: 34, 189*
stdin: 179*
stdout: 20*
store_two_bytes: 73, 93, 177.
str: 100, 101, 114, 117, 118, 119, 121, 122.
strargc: 188, 189*
strcat: 58, 63, 188, 200*
String constant didn't end: 149.
String didn't end: 168.
String too long: 117.
string_ptr: 39, 40, 42, 64, 182, 184.
system dependencies: 1, 2, 4, 7, 12, 17, 20, 21, 22, 24, 26, 28, 32, 115, 116, 121, 180, 181, 182, 187.
t: 53, 101, 165.
tab_mark: 15, 32, 139, 142, 145, 153, 154.
TANGLE: 2*
TANGLE_HELP: 188*
tats: 3.
temp_line: 124, 125.
term_in: 179, 181.
term_out: 20, 22*
text_char: 12, 13, 20*
text_file: 12, 23, 25, 28*
text_link: 37, 38, 43, 70, 71, 83, 85, 90, 112, 170, 178.
text_pointer: 43, 44, 70, 74, 78, 164.
text_ptr: 43, 44, 46, 74, 81, 90, 91, 165, 186.
This can't happen: 35.
This identifier has already...: 59.
This identifier was defined...: 59.
tok_mem: 37, 38, 43, 44, 45, 70, 73, 74, 75, 78, 79, 80, 81, 87, 90, 93, 165.
tok_ptr: 43, 44, 46, 73, 81, 90, 91, 93, 165, 182, 186.
tok_start: 37, 38, 43, 44, 46, 70, 74, 78, 83, 84, 85, 90, 91, 165.
trouble_shooting: 87, 145, 179, 180.
true: 6, 28, 29, 124, 125, 127, 132, 134, 136, 138, 143, 150, 179, 180, 183, 188.
Two numbers occurred...: 108.
exit: 34, 182.
unambiguous_length: 47, 50, 58, 63, 188, 200.
unbreakable: 95, 102, 113, 117.
up_to: 116, 145.
uppercase: 119, 120.
usage: 188, 189.
usage_help: 188.

val: 157, 158* 160, 161, 162, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199.
Value too big: 157.
verbatim: 72, 76, 113, 118, 139, 167, 169.
Verbatim string didn't end: 169.
Verbatim string too long: 118.
version_string: 182*

WEB file ended...: 132.
web_name: 24, 189, 200.
Where is the match...: 129, 133, 137.
wi: 41, 42.
wo: 185, 186.
write: 20, 34, 64, 97, 184.
write_log: 20, 64, 97, 184, 189.
x: 73.
xchr: 13, 14, 16, 17, 18, 32, 33, 49, 63, 64, 75, 76, 97, 155, 167, 181, 184.
xclause: 6.
xor: 13, 16, 18, 28.
You should double @ signs: 168, 169.
z: 44.
zi: 45, 46.
zo: 80, 83, 84, 85, 87, 90, 93.
zp: 74, 75.
zz: 8, 38, 43, 44, 45, 46, 74, 80, 83, 84, 85, 90, 91, 165, 182, 186.
TANGLE changes for C

STRUCTURES

Append `out_val` to buffer 103 Used in sections 102 and 104.
Append the decimal value of `v`, with parentheses if negative 111  Used in section 107.
Cases involving `@{` and `@}` 121  Used in section 113.
Cases like `<>` and `:=` 114*  Used in section 113.
Cases related to constants, possibly leading to `get_fraction` or `reswitch` 119*  Used in section 113.
Cases related to identifiers 116*  Used in section 113.
Check for ambiguity and update secondary hash 62  Used in section 61.
Check for overlong name 155  Used in section 153.
Check if `q` conflicts with `p` 63*  Used in section 62.
Check that all changes have been read 138  Used in section 183.
Check that `= or` follows this module name, otherwise return 176  Used in section 175.
Compare name `p` with current identifier, `goto found` if equal 56  Used in section 55.
Compiler directives 4  Used in section 2*.
Compress two-symbol combinations like `:` 147  Used in section 145.
Compute the hash code `h` 54  Used in section 53*.
Compute the name location `p` 55  Used in section 53*.
Compute the secondary hash code `h` and put the first characters into the auxiliary array `chopped_id` 58*  Used in section 57.

Constants in the outer block 8*  Used in section 2*.
Contribution is `*` or `/ or DIV or MOD 105*  Used in section 104.
Copy a string from the buffer to `tok_mem` 168  Used in section 165*.
Copy the parameter into `tok_mem` 93*  Used in section 90*.
Copy verbatim string from the buffer to `tok_mem` 169  Used in section 167.
Define and output a new string of the pool 64*  Used in section 61.
Define `parse_arguments` 188*  Used in section 2*.
Display one-byte token `a` 76  Used in section 74.
Display two-byte token starting with `a` 75  Used in section 74.
Do special things when `c = "\n", "\"", "\"", "\"", "\"", "\""; return at end 142  Used in section 141.
Empty the last line from the buffer 98  Used in section 112.
Enter a new identifier into the table at position `p` 61  Used in section 57.
Enter a new module name into the tree 67  Used in section 66.
Error handling procedures 30, 31  Used in section 2*.
Expand macro `a` and `goto found`, or `goto restart` if no output found 89*  Used in section 87.
Expand module `a`  24000, `goto restart` 88  Used in section 87.
Finish off the string pool file 184  Used in section 182*.
Force a line break 122  Used in section 113.
Get a preprocessed string 149  Used in section 145.
Get an identifier 148  Used in section 145.
Get control code and possible module name 150  Used in section 145.
Get the buffer ready for appending the new information 102  Used in section 101.
Give double-definition error, if necessary, and change `p` to type `?` 59  Used in section 57.
Globals in the outer block 9, 13, 23, 25, 27, 29, 38*, 40, 44, 50*, 65, 70, 79, 80, 82, 86, 94, 95, 100, 124, 126, 143, 156, 164, 171, 179*, 185, 200*  Used in section 2*.
Go to `found` if `c` is a hexadecimal digit, otherwise set `scanning_hex ← false` 146  Used in section 145.
Handle cases of `send_val` when `out_state` contains a sign 108  Used in section 107.
Handle file name arguments 189*  Used in section 188*.
If end of name, `goto done` 154  Used in section 153.
If previous output was `*` or `/`, `goto bad_case` 109  Used in section 107.
If previous output was `DIV` or `MOD`, `goto bad_case` 110*  Used in section 107.
If the current line starts with `@y`, report any discrepancies and return 133  Used in section 132.
If the next text is `'(#' ==' or `'[#]' =='`, call `define_macro` and `goto continue` 174*  Used in section 173*.
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 ) Used in section 165*.

</br>(Initialize the input system 134 ) Used in section 182*.

</br>(Initialize the output buffer 96 ) Used in section 112.

</br>(Initialize the output stacks 83 ) Used in section 112.

</br>(Insert the module number into tok_mem 177 ) Used in section 175.

</br>(Local variables for initialization 16, 41, 45, 51 ) Used in section 2*.

</br>(Make sure the parentheses balance 166* ) Used in section 165*.

</br>(Move buffer and limit to change_buffer and change_limit 131 ) Used in sections 128 and 132.

</br>(Other printable characters 115 ) Used in section 113.

</br>(Phase I: Read all the user’s text and compress it into tok_mem 183 ) Used in section 182*.

</br>(Phase II: Output the contents of the compressed tables 112 ) Used in section 182*.

</br>(Print error location based on input buffer 32 ) Used in section 31.

</br>(Print error location based on output buffer 33 ) Used in section 31.

</br>(Print statistics about memory usage 186 ) Used in section 182*.

</br>(Print the job history 187 ) Used in section 182*.

</br>(Put a parameter on the parameter stack, or goto restart if error occurs 90* ) Used in section 89*.

</br>(Put module name into mod_text[1 . . k] 153 ) Used in section 151.

</br>(Read from change_file and maybe turn off changing 137 ) Used in section 135.

</br>(Read from web_file and maybe turn on changing 136 ) Used in section 135.

</br>(Reduce sign_val_val to sign_val and goto restart 104 ) Used in section 102.

</br>(Remove a parameter from the parameter stack 91 ) Used in section 85*.

</br>(Remove p from secondary hash table 60 ) Used in section 59.

</br>(Scan the definition part of the current module 173* ) Used in section 172.

</br>(Scan the module name and make cur_module point to it 151 ) Used in section 150.

</br>(Scan the Pascal part of the current module 175 ) Used in section 172.

</br>(Send a string, goto reswitch 117 ) Used in section 113.

</br>(Send verbatim string 118 ) Used in section 113.

</br>(Set accumulator to the value of the right-hand side 158* ) Used in section 157*.

</br>(Set c to the result of comparing the given name to name p 68 ) Used in sections 66 and 69.

</br>(Set initial values 10, 14, 17*, 18, 21*, 26*, 42, 46, 48, 52, 71, 144, 152, 180* ) Used in section 2*.

</br>(Set val to value of decimal constant, and set next_control to the following token 160 ) Used in section 158*.

</br>(Set val to value of hexadecimal constant, and set next_control to the following token 162 ) Used in
section 158*.

</br>(Set val to value of octal constant, and set next_control to the following token 161 ) Used in section 158*.

</br>(Signal error, flush rest of the definition 159 ) Used in section 158*.

</br>(Skip over comment lines in the change file; return if end of file 129 ) Used in section 128.

</br>(Skip to the next nonblank line; return if end of file 130 ) Used in section 128.

</br>(Special code to finish real constants 120 ) Used in section 113.

</br>(Start scanning current macro parameter, goto restart 92 ) Used in section 87.

</br>(Types in the outer block 11, 12*, 37, 39, 43, 78 ) Used in section 2*.

</br>(Update the data structure so that the replacement text is accessible 178 ) Used in section 175.

</br>(Update the tables and check for possible errors 57 ) Used in section 53*. 