Appendix E

The TANGLE processor

(Version 4.6)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>The character set</td>
<td>11</td>
</tr>
<tr>
<td>Input and output</td>
<td>19</td>
</tr>
<tr>
<td>Reporting errors to the user</td>
<td>29</td>
</tr>
<tr>
<td>Data structures</td>
<td>37</td>
</tr>
<tr>
<td>Searching for identifiers</td>
<td>50</td>
</tr>
<tr>
<td>Searching for module names</td>
<td>65</td>
</tr>
<tr>
<td>Tokens</td>
<td>70</td>
</tr>
<tr>
<td>Stacks for output</td>
<td>77</td>
</tr>
<tr>
<td>Producing the output</td>
<td>94</td>
</tr>
<tr>
<td>The big output switch</td>
<td>112</td>
</tr>
<tr>
<td>Introduction to the input phase</td>
<td>123</td>
</tr>
<tr>
<td>Inputting the next token</td>
<td>143</td>
</tr>
<tr>
<td>Scanning a numeric definition</td>
<td>156</td>
</tr>
<tr>
<td>Scanning a macro definition</td>
<td>163</td>
</tr>
<tr>
<td>Scanning a module</td>
<td>171</td>
</tr>
<tr>
<td>Debugging</td>
<td>179</td>
</tr>
<tr>
<td>The main program</td>
<td>182</td>
</tr>
<tr>
<td>System-dependent changes</td>
<td>188</td>
</tr>
<tr>
<td>Index</td>
<td>201</td>
</tr>
</tbody>
</table>
**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 T EX, 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.

```plaintext
define my_name ≡ 'tangle'
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.

```plaintext
⟨Compiler directives 4⟩
```

```plaintext
program TANGLE(web_file, change_file, Pascal_file, pool);
```

```plaintext
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;
    begin kpse_set_program_name(argv[0], my_name); parse_arguments; ⟨Set initial values 10⟩
    end;
```

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

```plaintext
⟨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 }
```

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 \texttt{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 \texttt{char} for the data type associated with the characters in text files, while other Pascals consider \texttt{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 \texttt{text\_char} to stand for the data type of the characters in the input and output files. We shall also assume that \texttt{text\_char} consists of the elements $\texttt{chr(first\_text\_char)}$ through $\texttt{chr(last\_text\_char)}$, inclusive. The following definitions should be adjusted if necessary.

\begin{verbatim}
define text\_char \equiv ASCII\_code \{ the data type of characters in text files \}
define first\_text\_char = 0 \{ ordinal number of the smallest element of text\_char \}
define last\_text\_char = 255 \{ ordinal number of the largest element of text\_char \}
\end{verbatim}

\begin{verbatim}
⟨Types in the outer block 11⟩ +≡

for $i \leftarrow 1$ to $'37$ do $xchr[i] \leftarrow \texttt{chr}(i)$;
\end{verbatim}

\texttt{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 \texttt{WEB} more friendly on computers that have an extended character set, so that one can type things like $\neq$ 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 \texttt{WEB} are allowed to have in their input files, provided that unsuitable characters do not correspond to special codes like \texttt{carriage\_return} that are listed above.

(The present file \texttt{TANGLE.WEB} does not contain any of the non-ASCII characters, because it is intended to be used with all implementations of \texttt{WEB}. It was originally created on a Stanford system that has a convenient extended character set, then “sanitized” by applying another program that transliterated all of the non-standard characters into standard equivalents.)

\begin{verbatim}
⟨Set initial values 10⟩ +≡

for $i \leftarrow 1$ to $'37$ do $xchr[i] \leftarrow \texttt{chr}(i)$;
for $i \leftarrow '200$ to $'377$ do $xchr[i] \leftarrow \texttt{chr}(i)$;
\end{verbatim}
20* Terminal output is done by writing on file `term_out`, which is assumed to consist of characters of type `text_char`:

```c
#define term_out ≡ stdout
#define print(#) ≡ write(term_out, #)  {'print' means write on the terminal}
#define print_ln(#) ≡ writeLn(term_out, #)  {'print' and then start new line}
#define new_line ≡ writeLn(term_out)  {start new line}
#define print_nl(#) ≡ {
  print information starting on a new line
}
```

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

⟨Set initial values 10⟩ +=

{ Nothing need be done for C. }

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.

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

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.

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

26* The following code opens `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⟩ +=

```pascal
rewrite(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] ≠ " \n" then final_limit ← limit;
    if limit = buf_size then
      begin while ¬eoln (f) do vgetc(f);
      decr(limit);  { keep buffer[buf_size] empty }\n      if final_limit > limit then final_limit ← limit;
      println(´!\nInput\nline\ntoo\nlong´); loc ← 0; error;
    end;
    end;
  end;
  read ln(f); limit ← final_limit; input ln ← true;
  end;
end;
```
The `jump_out` procedure just cuts across all active procedure levels and jumps out of the program.

```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; 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 $zz = 1$.)

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

Globals in the outer block 9) +≡

```plaintext
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 \}
```

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 $2^{30}$. 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 and parametric2 identifiers have been defined to be parametric macros; like simple identifiers, their equiv value points to the replacement text.

```plaintext
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 \}
```
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 .. `max_id_length`] of `ASCII_code`; { chopped identifier }

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`.

```plaintext
function id_lookup(t : `eight_bits`): `name_pointer`; { finds current identifier }

    label found, not_found;

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

    begin l ← `id_loc − id_first`; { compute the length }

    (Compute the hash code `h` 54);
    (Compute the name location `p` 55);

    if `(p = name.ptr) ∨ (t ≠ `normal`)` then { Update the tables and check for possible errors 57 };
    `id_lookup` ← `p`;
    end;
```
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`⟩ ≡

begin
  \texttt{i} \leftarrow \texttt{id\_first}; \texttt{s} \leftarrow 0; \texttt{h} \leftarrow 0;
  \text{while } (\texttt{i} < \texttt{id\_loc}) \land (\texttt{s} < \texttt{unambig\_length}) \text{ do}
  \begin{align*}
    &\text{begin if } (\texttt{buffer}[\texttt{i}] \neq \_\_\texttt{)} \lor (\texttt{allow\_underlines} \land \neg \texttt{strict\_mode}) \text{ then} \\
    &\text{begin if } (\texttt{strict\_mode} \lor \texttt{force\_uppercase}) \land (\texttt{buffer}[\texttt{i}] \geq \texttt{a}) \text{ then } \texttt{chopped\_id}[\texttt{s}] \leftarrow \texttt{buffer}[\texttt{i}] - \_40 \\
    &\text{else if } (\neg \texttt{strict\_mode} \land \texttt{force\_lowercase}) \land (\texttt{buffer}[\texttt{i}] \geq \texttt{A}) \land (\texttt{buffer}[\texttt{i}] \leq \texttt{Z}) \text{ then} \\
    &\text{chopped\_id}[\texttt{s}] \leftarrow \texttt{buffer}[\texttt{i}] + \_40 \\
    &\text{else } \texttt{chopped\_id}[\texttt{s}] \leftarrow \texttt{buffer}[\texttt{i}]; \\
    &\texttt{h} \leftarrow (\texttt{h} + \texttt{h} + \texttt{chopped\_id}[\texttt{s}]) \mod \texttt{hash\_size}; \texttt{incr}(\texttt{s}); \\
    &\text{end}; \\
    &\text{incr}(\texttt{i}); \\
    &\text{end}; \\
    &\texttt{chopped\_id}[\texttt{s}] \leftarrow 0;
  \end{align*}
end
This code is used in section \texttt{57}.

⟨Check if \texttt{q} conflicts with \texttt{p}⟩ ≡

begin
  \texttt{k} \leftarrow \texttt{byte\_start}[\texttt{q}]; \texttt{s} \leftarrow 0; \texttt{w} \leftarrow \texttt{q} \mod \texttt{ww};
  \text{while } (\texttt{k} < \texttt{byte\_start}[\texttt{q} + \texttt{ww}]) \land (\texttt{s} < \texttt{unambig\_length}) \text{ do}
  \begin{align*}
    &\text{begin \texttt{c} \leftarrow \texttt{byte\_mem}[\texttt{w}, \texttt{k}]; \\
    &\text{if } \texttt{c} \neq \_\_\texttt{)} \lor (\texttt{allow\_underlines} \land \neg \texttt{strict\_mode}) \text{ then} \\
    &\text{begin if } (\texttt{strict\_mode} \lor \texttt{force\_uppercase}) \land (\texttt{c} \geq \texttt{a}) \text{ then } \texttt{c} \leftarrow \texttt{c} - \_40 \\
    &\text{else if } (\neg \texttt{strict\_mode} \land \texttt{force\_lowercase}) \land (\texttt{c} \geq \texttt{A}) \land (\texttt{c} \leq \texttt{Z}) \text{ then} \\
    &\text{c} \leftarrow \texttt{c} + \_40; \\
    &\text{if } \texttt{chopped\_id}[\texttt{s}] \neq \texttt{c} \text{ then } \texttt{goto} \ not\_found; \\
    &\texttt{incr}(\texttt{s}); \\
    &\text{end}; \\
    &\text{incr}(\texttt{k}); \\
    &\text{end}; \\
    &\text{if } (\texttt{k} = \texttt{byte\_start}[\texttt{q} + \texttt{ww}]) \land (\texttt{chopped\_id}[\texttt{s}] \neq 0) \text{ then } \texttt{goto} \ not\_found; \\
    &\texttt{print\_nl}(\texttt{\_\_Identifier\_conflicts\_with\_}); \\
    &\text{for } \texttt{k} \leftarrow \texttt{byte\_start}[\texttt{q}] \text{ to } \texttt{byte\_start}[\texttt{q} + \texttt{ww}] - 1 \text{ do } \texttt{print}(\texttt{xchr}[\texttt{byte\_mem}[\texttt{w}, \texttt{k}]]){; \\
    &\text{error}; \texttt{q} \leftarrow 0; \texttt{\{ only one conflict will be printed, since equiv[0] = 0 \}}
  \end{align*}
end
\texttt{not\_found}: \texttt{end}
This code is used in section \texttt{62}.
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 ≡ '3777777667 { 2^{29} - 73 }

( Define and output a new string of the pool 64*)
begin ilk[p] ← numeric; { strings are like numeric macros }
if l − double_chars = 2 then { this string is for a single character }
equiv[p] ← buffer[id.first + 1] + '10000000000
else begin { Avoid creating empty pool files. }
if string_ptr = 256 then rewritebin(pool, pool_name);
equiv[p] ← string_ptr + '10000000000; l ← l − double_chars − 1;
if l > 99 then err.print('!Preprocessed_string is too long');
incr(string_ptr); write(pool, xchr["0" + l div 10], xchr["0" + l mod 10]); { output the length }
pool_check_sum ← pool_check_sum + pool_check_sum + l;
while pool_check_sum > check_sum_prime do pool_check_sum ← pool_check_sum − check_sum_prime;
i ← id.first + 1;
while i < id.loc do
    begin write(pool, xchr[buffer[i]]); { output characters of string }
pool_check_sum ← pool_check_sum + pool_check_sum + buffer[i];
    while pool_check_sum > check_sum_prime do pool_check_sum ← pool_check_sum − check_sum_prime;
    if (buffer[i] = "\n") ∨ (buffer[i] = "@") then i ← i + 2
    { omit second appearance of doubled character }
else incr(i);
end;
write_ln(pool);
end;
end
```

This code is used in section 61.
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) ∨ (ilk[cur_name] = parametric2) then
         ⟨ Remove a parameter from the parameter stack 91 ⟩;
      end
   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;
```

(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.

\[
\text{Put a parameter on the parameter stack, or } \textbf{goto restart} \text{ if error occurs } 90^* \equiv \\
\text{while } (\text{cur}_y = \text{cur}_e) \land (\text{stack}_y > 0) \text{ do } \text{pop}_l \text{;}
\]
\[
\text{if } (\text{stack}_y = 0) \lor (\text{ilk}_y = \text{parametric}) \land (\text{tok}_m[z]_y, \\
\text{cur}_y) \neq "()" \lor (\text{ilk}_y = \text{parametric2}) \land (\text{tok}_m[z]_y, \text{cur}_y) \neq "["), \text{then}
\]
\[
\text{begin print}_n_l(\text{No parameter given for } \text{id}_a); \text{print}_i_d(a); \text{error; goto restart;}
\]
\[
\text{end;}
\]
\[
\text{(Copy the parameter into } \text{tok}_m \text{ 93*;}
\]
\[
\text{equiv}[\text{name}_y] \leftarrow \text{text}_y; \text{ilk}[\text{name}_y] \leftarrow \text{simple}; \text{w} \leftarrow \text{name}_y \text{mod } \text{ww}; \text{k} \leftarrow \text{byte}_y[w];
\]
\[
\text{debug if } \text{k} = \text{max}_y \text{ then } \text{overflow}(\text{byte}_y; \text{memory});
\]
\[
\text{byte}_m[w, k] \leftarrow "#"; \text{incr}(k); \text{byte}_y[w] \leftarrow k;
\]
\[
\text{gubed } \{ \text{this code has set the parameter identifier for debugging printouts} \}
\]
\[
\text{if } \text{name}_y > \text{max}_y \text{ names } - \text{ww} \text{ then } \text{overflow}(\text{name});
\]
\[
\text{byte}_s[t]_y \leftarrow \text{k}; \text{incr}(\text{name}_y);
\]
\[
\text{if } \text{text}_y > \text{max}_y \text{ texts } - \text{zz} \text{ then } \text{overflow}(\text{text});
\]
\[
\text{text}_l[k]_y \leftarrow 0; \text{tok}_s[t]_y + \text{zz} \leftarrow \text{tok}_s[z]; \text{incr}(\text{text}_y); \text{z} \leftarrow \text{text}_y \text{mod } \text{zz}
\]
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.

```plaintext
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 + 77777)
  else begin if b ≥ 200 then
              begin app_repl(b); b ← tok_mem[zo, cur_byte]; incr(cur_byte);
              end
    else case b of
      "(": 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 = ";
      othercases do nothing
  endcases;
  app_repl(b);
end;
end
```

This code is used in section 90*.
105. Contribution is * or / or DIV or MOD 105. \( \equiv \)
\[
(t = \text{ident}) \land (v = 3) \land ((\text{out}_\text{contrib}[1] = "D") \land (\text{out}_\text{contrib}[2] = "I") \land (\text{out}_\text{contrib}[3] = "V") \lor \\
((\text{out}_\text{contrib}[1] = "d") \land (\text{out}_\text{contrib}[2] = "i") \land (\text{out}_\text{contrib}[3] = "v") \lor \\
((\text{out}_\text{contrib}[1] = "M") \land (\text{out}_\text{contrib}[2] = "O") \land (\text{out}_\text{contrib}[3] = "D") \lor \\
((\text{out}_\text{contrib}[1] = "m") \land (\text{out}_\text{contrib}[2] = "o") \land (\text{out}_\text{contrib}[3] = "d") ))\lor \\
((t = \text{misc}) \land ((v = \ast) \lor (v = /)))
\]
This code is used in section 104.

110. If previous output was DIV or MOD, goto bad_case 110. \( \equiv \)
\[
\text{if } (\text{out}_\text{ptr} = \text{break}_\text{ptr} + 3) \lor ((\text{out}_\text{ptr} = \text{break}_\text{ptr} + 4) \land (\text{out}_\text{buf}[\text{break}_\text{ptr}] = \ast)) \text{ then} \\
\quad \text{if } ((\text{out}_\text{buf}[\text{out}_\text{ptr} - 3] = "D") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 2] = "I") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 1] = "V") \lor \\
((\text{out}_\text{buf}[\text{out}_\text{ptr} - 3] = "d") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 2] = "i") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 1] = "v") \lor \\
((\text{out}_\text{buf}[\text{out}_\text{ptr} - 3] = "M") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 2] = "O") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 1] = "D") \lor \\
((\text{out}_\text{buf}[\text{out}_\text{ptr} - 3] = "m") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 2] = "o") \land (\text{out}_\text{buf}[\text{out}_\text{ptr} - 1] = "d") ) \text{ then} \\
\quad \text{goto bad_case}
\]
This code is used in section 107.
114* (Cases like $\ll$ and $\equiv 114*$) ≡

\[
\begin{align*}
\text{and_sign: begin} & \text{ out_contrib}[1] \leftarrow "a" ; \text{ out_contrib}[2] \leftarrow "n" ; \text{ out_contrib}[3] \leftarrow "d" ; \text{ send_out}(ident, 3) ; \\
\text{end} ; \\
\text{not_sign: begin} & \text{ out_contrib}[1] \leftarrow "n" ; \text{ out_contrib}[2] \leftarrow "o" ; \text{ out_contrib}[3] \leftarrow "t" ; \text{ send_out}(ident, 3) ; \\
\text{end} ; \\
\text{set_element_sign: begin} & \text{ out_contrib}[1] \leftarrow "i" ; \text{ out_contrib}[2] \leftarrow "n" ; \text{ send_out}(ident, 2) ; \\
\text{end} ; \\
\text{or_sign: begin} & \text{ out_contrib}[1] \leftarrow "o" ; \text{ out_contrib}[2] \leftarrow "r" ; \text{ send_out}(ident, 2) ; \\
\text{end} ; \\
\text{left_arrow: begin} & \text{ out_contrib}[1] \leftarrow ";" ; \text{ out_contrib}[2] \leftarrow "=" ; \text{ send_out}(str, 2) ; \\
\text{end} ; \\
\text{not_equal: begin} & \text{ out_contrib}[1] \leftarrow "<" ; \text{ out_contrib}[2] \leftarrow ">" ; \text{ send_out}(str, 2) ; \\
\text{end} ; \\
\text{less_or_equal: begin} & \text{ out_contrib}[1] \leftarrow "<" ; \text{ out_contrib}[2] \leftarrow "=" ; \text{ send_out}(str, 2) ; \\
\text{end} ; \\
\text{greater_or_equal: begin} & \text{ out_contrib}[1] \leftarrow ">" ; \text{ out_contrib}[2] \leftarrow "=" ; \text{ send_out}(str, 2) ; \\
\text{end} ; \\
\text{equivalence_sign: begin} & \text{ out_contrib}[1] \leftarrow "=" ; \text{ out_contrib}[2] \leftarrow "=" ; \text{ send_out}(str, 2) ; \\
\text{end} ; \\
\text{double_dot: begin} & \text{ out_contrib}[1] \leftarrow "." ; \text{ out_contrib}[2] \leftarrow "." ; \text{ send_out}(str, 2) ; \\
\text{end} ; \\
\end{align*}
\]

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.

\[
\begin{align*}
\text{define } & \text{ up_to}(\#) \equiv \# - 24, \# - 23, \# - 22, \# - 21, \# - 20, \# - 19, \# - 18, \# - 17, \# - 16, \# - 15, \# - 14, \# - 13, \\
& \quad \# - 12, \# - 11, \# - 10, \# - 9, \# - 8, \# - 7, \# - 6, \# - 5, \# - 4, \# - 3, \# - 2, \# - 1, \# \\
\text{end} ; \\
\text{define } & \text{ up_to("Z")} : \text{ begin if force_lowercase then } \text{ out_contrib}[1] \leftarrow \text{ cur_char} + '40 \\
& \text{ else } \text{ out_contrib}[1] \leftarrow \text{ cur_char} ; \\
& \text{ send_out}(ident, 1) ; \\
& \text{end} ; \\
\text{define } & \text{ up_to("z")} : \text{ begin if force_uppercase then } \text{ out_contrib}[1] \leftarrow \text{ cur_char} - '40 \\
& \text{ else } \text{ out_contrib}[1] \leftarrow \text{ cur_char} ; \\
& \text{ send_out}(ident, 1) ; \\
& \text{end} ; \\
\text{identifier: begin} & k \leftarrow 0 ; j \leftarrow \text{ byte_start}[\text{cur_val}] ; w \leftarrow \text{ cur_val} \mod ww ; \\
\text{while } & (k < \text{ max_id_length}) \land (j < \text{ byte_start}[\text{cur_val} + ww]) \text{ do} \\
& \text{ begin incr}(k) ; \text{ out_contrib}[k] \leftarrow \text{ byte_mem}[w, j] ; \text{ incr}(j) ; \\
& \text{ if force_uppercase } \land (\text{ out_contrib}[k] \geq "a") \text{ then } \text{ out_contrib}[k] \leftarrow \text{ out_contrib}[k] - '40 \\
& \text{ else if force_lowercase } \land (\text{ out_contrib}[k] \leq "z") \text{ then } \text{ out_contrib}[k] \leftarrow \text{ out_contrib}[k] + '40 \\
& \quad \text{ else if } \text{ allow_underlines } \land (\text{ out_contrib}[k] = ".") \text{ then } \text{ decr}(k) ; \\
& \text{ send_out}(ident, k) ; \\
& \text{end} ; \\
\end{align*}
\]

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 ($2^{31} - 1$).

```c
#define digits "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"

Cases related to constants, possibly leading to `get_fraction` or `reswitch`:

```c
define digits "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
⟨\text{Cases related to constants, possibly leading to `get_fraction` or `reswitch`}⟩
\equiv\begin{align*}
\text{digits:} & \quad \text{begin } n \leftarrow 0; \\
& \quad \text{repeat cur_char }\leftarrow \text{cur_char }- \text{"0"}; \\
& \quad \quad \text{if } n \geq 1463146314 \text{ then err_print(\text{"!\text{Constant too big}"})} \\
& \quad \quad \text{else } n \leftarrow 10 \times n + \text{cur_char}; \\
& \quad \quad \text{cur_char }\leftarrow \text{get_output}; \\
& \quad \quad \text{until (cur_char } > \text{"9"}) \lor (\text{cur_char } < \text{"0"}); \\
& \quad \text{send_val(n); } k \leftarrow 0; \\
& \quad \text{if cur_char }= \text{"e"} \text{ then } \text{cur_char }\leftarrow \text{"E"}; \\
& \quad \text{if cur_char }= \text{"E"} \text{ then goto `get_fraction"; } \\
& \quad \text{else goto `reswitch"; } \end{align*}
\end{align*}
```

```
\begin{align*}
\text{check_sum: } & \quad \text{send_val(pool_check_sum);} \\
\text{octal:} & \quad \text{begin } n \leftarrow 0; \text{cur_char }\leftarrow \text{"0"}; \\
& \quad \text{repeat cur_char }\leftarrow \text{cur_char }- \text{"0"}; \\
& \quad \quad \text{if } n \geq \text{100000000000 \ then err_print(\text{"!\text{Constant too big}"})} \\
& \quad \quad \text{else } n \leftarrow 8 \times n + \text{cur_char}; \\
& \quad \quad \text{cur_char }\leftarrow \text{get_output}; \\
& \quad \quad \text{until (cur_char } > \text{"7"}) \lor (\text{cur_char } < \text{"0"}); \\
& \quad \text{send_val(n); goto `reswitch"; } \end{align*}
\end{align*}
```

```
\begin{align*}
\text{hex:} & \quad \text{begin } n \leftarrow 0; \text{cur_char }\leftarrow \text{"0"}; \\
& \quad \text{repeat if cur_char }\geq \text{"A"} \text{ then } \text{cur_char }\leftarrow \text{cur_char }+ 10 - \text{"A"} \\
& \quad \quad \text{else cur_char }\leftarrow \text{cur_char }- \text{"0"}; \\
& \quad \quad \text{if } n \geq \text{40000000 \ then err_print(\text{"!\text{Constant too big}"})} \\
& \quad \quad \text{else } n \leftarrow 16 \times n + \text{cur_char}; \\
& \quad \quad \text{cur_char }\leftarrow \text{get_output}; \\
& \quad \quad \text{until (cur_char } > \text{"F"}) \lor (\text{cur_char } < \text{"0"}) \lor ((\text{cur_char } > \text{"9"}) \land (\text{cur_char } < \text{"A"}); \\
& \quad \text{send_val(n); goto `reswitch"; } \end{align*}
\end{align*}
```

```
\begin{align*}
\text{number: } & \quad \text{send_val(cur_val);} \\
\text{.}: & \quad \text{begin } k \leftarrow 1; \text{out_contribution[1] }\leftarrow \text{.} \text{;} \text{cur_char }\leftarrow \text{get_output}; \\
& \quad \text{if cur_char }= \text{.} \text{ then } \\
& \quad \quad \text{begin out_contribution[2] }\leftarrow \text{.} \text{;} \text{send_out(str,2);} \\
& \quad \text{end} \\
& \quad \text{else if (cur_char }\geq \text{"0"}) \land (\text{cur_char }\leq \text{"9"}) \text{ then goto `get_fraction";} \\
& \quad \text{else begin } \text{send_out(misc,\text{.}); goto `reswitch";} \\
& \quad \text{end} \\
& \text{end}; \\
\end{align*}
```

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) ≥ '10000000000 then
    begin err_print(´!_Value_too_big:`.accumulator : 1); accumulator ← 0;
      end;
  equiv[p] ← accumulator + ´10000000000; { 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] − ´10000000000);
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*.
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
        "(": 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; }
        ⟨ ⟨ Copy a string from the buffer to tok_mem 168; ⟩ ⟩
        ⟨ ⟨ Copy a string from the buffer to tok_mem 168; ⟩ ⟩
        "": if t 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
This code is used in section 165*. 
(Scan the definition part of the current module 173*)

\[ \text{next\_control} \leftarrow 0; \]
\[ \text{loop begin continue: while next\_control} \leq \text{format do} \]
\[ \begin{align*}
& \quad \text{begin next\_control} \leftarrow \text{skip\_ahead}; \\
& \quad \text{if next\_control} = \text{module\_name then} \\
& \qquad \text{begin \{we want to scan the module name too\}} \\
& \qquad \quad \text{loc} \leftarrow \text{loc} - 2; \text{next\_control} \leftarrow \text{get\_next}; \\
& \qquad \text{end;} \\
& \quad \text{end;} \\
& \quad \text{if next\_control} \neq \text{definition \then} \text{goto done;} \\
& \quad \text{next\_control} \leftarrow \text{get\_next}; \ \{\text{get identifier name}\} \\
& \quad \text{if next\_control} \neq \text{identifier \then} \\
& \qquad \text{begin err\_print(\text{´!Definition\_flushed,\_must\_start\_with\_\', \_identifier\_of\_length\_>\_1\');}} \\
& \qquad \text{goto continue;} \\
& \qquad \text{end;} \\
& \quad \text{next\_control} \leftarrow \text{get\_next}; \ \{\text{get token after the identifier}\} \\
& \quad \text{if next\_control} = \text{"="} \text{then} \\
& \qquad \text{begin \text{scan\_numeric}(id\_lookup(\text{numeric})); \text{goto continue;}} \\
& \qquad \text{end} \\
& \quad \text{else if next\_control} = \text{equivalence\_sign \then} \\
& \qquad \text{begin \text{define\_macro}(simple); \text{goto continue;}} \\
& \qquad \text{end} \\
& \quad \text{else \{If the next text is \text{´(#)=='} or \text{\[#]=='}, call define\_macro \and \text{goto continue 174\*\};}} \\
& \quad \text{err\_print(\text{´!Definition\_flushed\_since\_it\_starts\_badly\');}} \\
& \quad \text{end;} \\
\text{done:} \\
\]

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

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

```plaintext
define term_in ≡ stdin

 Globals in the outer block 9
+≡

d debug trouble_shooting: boolean;  { is debug_help wanted? }
d dt: integer;  { operation code for the debug_help routine }
d dd: integer;  { operand in procedures performed by debug_help }
d debug_cycle: integer;  { threshold for debug_help stopping }
d debug_skipped: integer;  { we have skipped this many debug_help calls }

gubed
```

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

```plaintext
Set initial values 10
+≡

d debug trouble_shooting ← true; debug_cycle ← 1; debug_skipped ← 0;
d trouble_shooting ← false; debug_cycle ← 99999;  { use these when it almost works }

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(banner); { print a ‘‘banner line’’ }
print ln(version_string); { 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; }
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 }
new_line;
if (history ≠ spotless) ∧ (history ≠ harmless_message) then uexit(1)
else uexit(0);
end.
System-dependent changes. Parse a Unix-style command line.

\[ \text{Define } \text{argument_is}(\#) \equiv (\text{strcmp}(\text{long_options}[\text{option_index}], \#) = 0) \]

\[ \text{(Define parse_arguments 188*)} \equiv \]

\[ \text{procedure } \text{parse_arguments}; \]
\[ \text{const } n\_\text{options} = 10; \quad \{ \text{Pascal won't count array lengths for us.} \} \]
\[ \text{var long\_options: array [0..n\_options] of getopt\_struct; } \]
\[ \text{getopt\_return\_val: integer; option\_index: c\_int\_type; current\_option: 0..n\_options; len: integer; } \]
\[ \text{begin } \{ \text{Define the option table 190*} \}; \]
\[ \text{unambig\_length } \equiv \text{def\_unambig\_length; } \]
\[ \text{repeat } \text{getopt\_return\_val } \leftarrow \text{getopt\_long\_only} (\text{argc}, \text{argv}, \text{´}\text{´}, \text{long\_options}, \text{address\_of} (\text{option\_index})); \]
\[ \text{if } \text{getopt\_return\_val } = -1 \text{ then } \]
\[ \begin{align*} \text{begin } & \text{do\_nothing; } \{ \text{End of arguments; we exit the loop below.} \} \\ \text{end} \]
\[ \text{else if } \text{getopt\_return\_val } = \text{´}\text{?}\text{´} \text{ then } \]
\[ \begin{align*} \text{begin } & \text{usage(} \text{my\_name}; \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´help´) then } \]
\[ \begin{align*} \text{begin } & \text{usage\_help(TANGLE\_HELP, nil); } \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´version´) then } \]
\[ \begin{align*} \text{begin } & \text{print\_version\_and\_exit(banner, nil, `D.E.\_Knuth´, nil); } \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´mixedcase´) then } \]
\[ \begin{align*} \text{begin } & \text{force\_uppercase } \leftarrow \text{false; force\_lowercase } \leftarrow \text{false; } \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´uppercase´) then } \]
\[ \begin{align*} \text{begin } & \text{force\_uppercase } \leftarrow \text{true; force\_lowercase } \leftarrow \text{false; } \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´lowercase´) then } \]
\[ \begin{align*} \text{begin } & \text{force\_uppercase } \leftarrow \text{false; force\_lowercase } \leftarrow \text{true; } \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´underlines´) then } \]
\[ \begin{align*} \text{begin } & \text{allow\_underlines } \leftarrow \text{true; } \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´strict´) then } \]
\[ \begin{align*} \text{begin } & \text{strict\_mode } \leftarrow \text{true; } \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´loose´) then } \]
\[ \begin{align*} \text{begin } & \text{strict\_mode } \leftarrow \text{false; } \\ \text{end} \]
\[ \text{else if } \text{argument\_is(´length´) then } \]
\[ \begin{align*} \text{begin } & \text{len } \leftarrow \text{atoi(optarg)}; \\ \text{if } (\text{len } \leq 0) \lor (\text{len } > \text{max\_id\_length}) \text{ then } \text{len } \leftarrow \text{max\_id\_length}; \\ \text{unambig\_length } \leftarrow \text{len}; \end{align*} \]
\[ \text{end; } \{ \text{Else it was a flag; getopt has already done the assignment.} \} \]
\[ \text{until } \text{getopt\_return\_val } = -1; \]
\[ \text{(Handle file name arguments 189*)} \]
\[ \text{end; } \]

This code is used in section 2*.

\[ \text{§188} \]
Now `optind` is the index of first non-option on the command line.

```c
if (optind + 1 > argc) ∨ (optind + 3 < argc) then
  begin write Ln(stderr, my_name, "No Need file to options for these 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)) ≠ 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.

```c
Define the option table 190*

current_option ← 0; long_options[current_option].name ← ¨help¨;
long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
long_options[current_option].val ← 0; incr(current_option);
```

Another of the standard options.

```c
Define the option table 190*

long_options[current_option].name ← ¨version¨; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

Use all mixed case.

```c
Define the option table 190*

long_options[current_option].name ← ¨mixedcase¨; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

Use all uppercase.

```c
Define the option table 190*

long_options[current_option].name ← ¨uppercase¨; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

Use all lowercase.

```c
Define the option table 190*

long_options[current_option].name ← ¨lowercase¨; long_options[current_option].has_arg ← 0;
long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```
195* Allow underlines.
\[
\langle \text{Define the option table 190*} \rangle + \equiv \\
\text{long_options}[\text{current_option}].\text{name} \leftarrow \text{`underlines`}; \text{long_options}[\text{current_option}].\text{has_arg} \leftarrow 0; \\
\text{long_options}[\text{current_option}].\text{flag} \leftarrow 0; \text{long_options}[\text{current_option}].\text{val} \leftarrow 0; \text{incr(\text{current_option});}
\]

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

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

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

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

200* Global filenames.
\[
\langle \text{Globals in the outer block 9} \rangle + \equiv \\
\text{web_name}, \text{chg_name}, \text{pascal_name}, \text{pool_name}: \text{const_c_string}; \\
\text{force_uppercase}, \text{force_lowercase}, \text{allow_underlines}, \text{strict_mode}: \text{boolean}; \\
\text{unambig_length}: 0 \ldots \text{max_id_length};
\]
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: 191*
- 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*
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_val: 99, 103, 111.
argc: 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*
atol: 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.
breakpoint: 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.
carriage_return: 15, 17*, 28*
Change file ended...: 130, 132, 137.
Change file entry did not match: 138.
change_changing: 125, 132, 134, 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*
chap_hash: 50*, 52, 60, 62.
chopped_id: 50*, 53*, 58*, 63*
chr: 12*, 13, 17*, 18.
cmdline: 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.

current_option: 188* 190* 191* 192* 193* 194* 195* 196* 197* 198* 199* 200*.
d: 145*.
dd: 179* 181.
ddt: 179* 181.

decr: 6, 28* 85* 91, 93* 99, 116* 121, 142, 148, 153, 165* 166*.
decr_unambig_length: 8* 188*.
decr_unambig_length: 8* 188*.
def_noun: 179* 180* 181.
def_macro: 179* 180* 181.
define_macro: 179* 180* 181.
definition: 139, 156, 158* 167, 173*.

definition flushed...: 173*.
digits: 119* 158*.
do_nothing: 6, 93* 102, 113, 145, 158* 165* 188*.

double @ sign missing: 149.
double_chars: 50* 64* 143, 149.
double_dot: 72, 114* 147.
EBCDIC: 115.
eight_bits: 37, 38* 53* 82, 87, 95, 101, 113, 139, 140, 141, 145, 156, 165* 170.
else: 7.
end: 3, 7.
end_comment: 72, 76, 121, 139, 147.
end_field: 78, 79.
end_of_definition: 156, 159.
endcases: 7.
eof: 28*.
eov: 28*.
equal: 66, 67, 68.
equiv: 37, 38* 47* 48, 50* 60, 62, 63* 64* 67, 84, 88, 89* 90* 157* 158* 170, 178.
equivalence_sign: 15, 114* 147, 173* 174* 176.

error: 28* 31, 34* 63* 88, 90*.
error_message: 9, 187.
exit: 5, 6, 85* 107, 127, 128, 132, 141, 172, 181.
extend_filename: 189*.
extension: 66, 68, 69.

Extra @: 165*.
Extra @: 165*.
Extra @: 165*.
Extra @: 165*.
f: 28*.
false: 28* 29, 125, 126, 127, 132, 134, 144, 146, 180* 183, 188*.
fatal_error: 34* 35, 36.
fatal_message: 9, 187.
fflush: 22*.
fflush: 22*.
final_limit: 28*.
first_text_char: 12* 18*.
flag: 190* 191* 192* 193* 194* 195* 196* 197* 198* 199*.
flush_buffer: 97, 98, 122.
force_line: 72, 76, 113, 139.
force_lowercase: 58* 63* 116* 188* 200*.
force_uppercase: 58* 63* 116* 188* 200*.
form_feed: 15, 28*.
format: 139, 156, 158* 167, 173*.
forward: 30.
found: 5, 53* 55, 56, 66, 87, 89* 145, 146, 165* 168.
frac: 100, 101, 102, 104, 113, 120.
Fraction too long: 120.
get_fraction: 113, 119* 120.
get_line: 124, 135, 140, 141, 145, 153.
get_output: 86, 87, 94, 112, 113, 117, 118, 119* 120.
getc: 28*.
getopt: 188*.
getopt_long_only: 188*.
getopt_return_val: 188*.
getopt_struct: 188*.
greater: 66, 68, 69.
greater_or_equal: 15, 114* 147.
gubed: 3.
h: 51, 53*.
harmless_message: 9, 182* 187.
has_arg: 190* 191* 192* 193* 194* 195* 196* 197* 198* 199*.
hash: 39, 50* 52, 55.
hash_size: 8* 50* 51, 52, 53* 54, 58*.
hex: 72, 76, 119* 139, 150, 158*.
history: 9, 10, 182* 187.
Hm... n of the preceding...: 133.
i: 16, 53*.
Identifier conflict: 63
ignore: 139, 140, 150.
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.
Input ended in mid-comment: 141.
Input ended in section name: 153.
Input line too long: 28*
input_ln: 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.
linesdont_match: 127, 132.
link: 37, 38*, 39, 48, 50*, 55.
Long line must be truncated: 97.
longest_name: 10, 118, 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*, 58*, 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]: 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*
normal: 47*50*53*57*59*60*61*89*158*167.
Not present: <section name>: 88.*
not_equal: 15*114*147.*
not_found: 5*53*63.*
not_sign: 15*114.*
num_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.*
othercases: 7.*
others: 7.*
out_app: 95*102*104*106*108.*
outbuf: 31*33*94*95*96*97*99*100*109*110*181*184.*
outbuf_size: 8*31*94*97*99.*
out_constr: 100*101*105*113*114*116*117*118*119*120*121*181.*
out_ptr: 33*94*95*96*97*98*99*101*102*106*107*109*110*111*122*181.*
out_sign: 95*103*104*107*108.*
out_state: 95*96*101*102*104*106*107*108*111*113*117*122.*
outval: 95*103*104*106*107*108.*
output_state: 78*79.*
overflow: 36*61*67*73*84*90*93*165.*
p: 49*53*66*69*74*84*157*170*172.*
pack: 61.*
param: 72*76*87*93*165.*
parametric: 47*53*85*89*90*93*164*165*166*174.*
parametric2: 47*53*85*89*90*93*165*174.*
pascal: 2*25*26*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: 20*31*32*33*49*63*74*75*76*88*93*97*139*155*181*182*186.*
print_id: 49*75*88*90*181.*
print_in: 20*32*33*182.*
print_nl: 20*28*63*88*90*112*155*181*184*186*187.*
print_repl: 74*181.*
print_version_and_exit: 188.*
Program ended at brace level n: 98.*
push_level: 84*88*89*92.*
q: 53*66*69*157.*
r: 69.*
read: 181.*
read_in: 28.*
remove_suffix: 189.*
repl_field: 78*79.*
restart: 5*87*88*89*90*92*101*102*104*135*145*150.*
reswitch: 5*113*117*119*120*157*158*165*169.*
return: 5*6.*
rewrite: 26.*
rewritebin: 64.*
link: 48*66*67*69.*
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.*
semi_ptr: 94*96*97*98*101.*
send_out: 100*101*112*113*114*116*117*118*119*120*121*122.*
send_sign: 100*106*112*113.*
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_sign: 95*102*106*108.*
sign_val_val: 95*102*106*108.*
simple: 47*53*89*90*164*173.*
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.
strcmp: 188, 189.
strict_mode: 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.
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.
uneq: 34, 182.
unambig_length: 47, 50, 58, 63, 188, 200.
unbreakable: 95, 102, 113, 117.
up_to: 116, 145.
upcase: 119, 120.
usage: 188, 189.
usage_help: 188.
Use == for macros: 174.

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_name: 24, 189, 200.
Where is the match: 129, 133, 137.
wi: 41, 42.
wo: 185, 186.
write: 20, 34, 64, 97, 184.
write_line: 20, 64, 97, 184, 189.
x: 73.
chr: 13, 14, 16, 17, 18, 32, 33, 49, 63, 64, 75, 76, 97, 155, 167, 181, 184.
xclause: 6.
xord: 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.
Append \textit{out\_val} to buffer 103 \quad Used in sections 102 and 104.

Append the decimal value of \textit{v}, with parentheses if negative 111 \quad Used in section 107.

Cases involving \texttt{@} and \texttt{\@} 121 \quad Used in section 113.

Cases like \texttt{<>} and \texttt{:=} 114* \quad Used in section 113.

Cases related to constants, possibly leading to \texttt{get\_fraction} or \texttt{reswitch} 119* \quad Used in section 113.

Cases related to identifiers 116* \quad Used in section 113.

Check for ambiguity and update secondary hash 62* \quad Used in section 61.

Check for overlone name 155 \quad Used in section 153.

Check if \textit{q} conflicts with \textit{p} 63* \quad Used in section 62.

Check that all changes have been read 138 \quad Used in section 183.

Check that = or \equiv follows this module name, otherwise \textbf{return} 176 \quad Used in section 175.

Compare name \textit{p} with current identifier, \textbf{goto} \textit{found} if equal 56 \quad Used in section 55.

Compiler directives 4* \quad Used in section 2*.

Compress two-symbol combinations like \texttt{':='} 147 \quad Used in section 145.

Compute the hash code \textit{h} 54 \quad Used in section 53*.

Compute the name location \textit{p} 55 \quad Used in section 53*.

Compute the secondary hash code \textit{h} and put the first characters into the auxiliary array \textit{chopped\_id} 58* \quad Used in section 57.

Constants in the outer block 8* \quad Used in section 2*.

Contribution is * or / or \texttt{DIV} or \texttt{MOD} 105* \quad Used in section 104.

Copy a string from the buffer to \texttt{tok\_mem} 168 \quad Used in section 165*.

Copy the parameter into \texttt{tok\_mem} 93* \quad Used in section 90*.

Copy verbatim string from the buffer to \texttt{tok\_mem} 169 \quad Used in section 167.

Define and output a new string of the pool 64* \quad Used in section 61.

Define the option table 190*, 191*, 192*, 193*, 194*, 195*, 196*, 197*, 198*, 199* \quad Used in section 188*.

Define \texttt{parse\_arguments} 188* \quad Used in section 2*.

Display one-byte token \textit{a} 76 \quad Used in section 74.

Display two-byte token starting with \textit{a} 75 \quad Used in section 74.

Do special things when \textit{c} = "\texttt{@\texttt{n}}, "\texttt{\textbackslash\texttt{n}}, "\texttt{\{\texttt{n}}, "\texttt{\}n}; \textbf{return} at end 142 \quad Used in section 141.

Empty the last line from the buffer 98 \quad Used in section 112.

Enter a new identifier into the table at position \textit{p} 61 \quad Used in section 57.

Enter a new module name into the tree 67 \quad Used in section 66.

Error handling procedures 30, 31 \quad Used in section 2*.

Expand macro \textit{a} and \textbf{goto} \textit{found}, or \textbf{goto} \textit{restart} if no output found 89* \quad Used in section 87.

Expand module \texttt{a} = \texttt{\textasciitilde24000}, \textbf{goto} \textit{restart} 88 \quad Used in section 87.

Finish off the string pool file 184 \quad Used in section 182*.

Force a line break 122 \quad Used in section 113.

Get a preprocessed string 149 \quad Used in section 145.

Get an identifier 148 \quad Used in section 145.

Get control code and possible module name 150 \quad Used in section 145.

Get the buffer ready for appending the new information 102 \quad Used in section 101.

Give double-definition error, if necessary, and change \textit{p} to type ? 59 \quad Used in section 57.

Global 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* \quad Used in section 2*.

Go to \textit{found} if \textit{c} is a hexadecimal digit, otherwise set \texttt{scanning\_hex} \leftarrow \textit{false} 146 \quad Used in section 145.

Handle cases of \texttt{send\_val} when \texttt{out\_state} contains a sign 108 \quad Used in section 107.

Handle file name arguments 189* \quad Used in section 188*.

If end of name, \textbf{goto} \textit{done} 154 \quad Used in section 153.

If previous output was * or /, \textbf{goto} \textit{bad\_case} 109 \quad Used in section 107.

If previous output was \texttt{DIV} or \texttt{MOD}, \textbf{goto} \textit{bad\_case} 110* \quad Used in section 107.

If the current line starts with \texttt{?y}, report any discrepancies and \textbf{return} 133 \quad Used in section 132.

If the next text is \texttt{\{(#)=} or \texttt{\{'[#]='}, call \texttt{define\_macro} and \textbf{goto} \textit{continue} 174* \quad Used in section 173*.
(In cases that \textit{a} is a non-ASCII token (identifier, module name, etc.), either process it and change \textit{a} to a byte that should be stored, or \textbf{goto} \textit{continue} if \textit{a} should be ignored, or \textbf{goto} \textit{done} if \textit{a} signals the end of this replacement text 167) \ Used in section 165*.

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