The TANGLE processor

(Version 4.6)

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

For large WEB files one should have a large memory, since TANGLE keeps all the Pascal text in memory (in an abbreviated form). The program uses a few features of the local Pascal compiler that may need to be changed in other installations:

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

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

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

```
define 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.

```
⟨Compiler directives 4⟩
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⟩
    ⟨Error handling procedures 30⟩
    ⟨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;
```

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

```
⟨Constants in the outer block 8*⟩ ≡
  buf_size = 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 = 353; { should be prime }
  longest_name = 400; { module names shouldn't be longer than this }
  line_length = 72; { lines of Pascal output have at most this many characters }
  out_buf_size = 144; { length of output buffer, should be twice line_length }
  stack_size = 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*.

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

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

```
define text_char ≡ ASCII_code   { the data type of characters in text files }
define first_text_char = 0   { ordinal number of the smallest element of text_char }
define last_text_char = 255   { ordinal number of the largest element of text_char }
```

⟨ Types in the outer block 11 ⟩ +≡

```
  text_file = packed file of text_char;
```

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

```
for i ← 1 to ´37 do xchr[i] ← 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.)

⟨ Set initial values 10 ⟩ +≡

```
  for i ← 1 to ´37 do xchr[i] ← chr(i);
  for i ← ´200 to ´377 do xchr[i] ← chr(i);
```

20* Terminal output is done by writing on file *term_out*, which is assumed to consist of characters of type *text_char*:

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

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

```
⟨Set initial values 10⟩ +≡
    {Nothing need be done 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.

```
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.

```
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* and *pool*. Since these files were listed in the program header, we assume that the Pascal runtime system has checked that suitable external file names have been given.

```
⟨Set initial values 10⟩ +≡
  rewrite(Pascal_file, pascal_name);
```

28* The *input_ln* procedure brings the next line of input from the specified file into the *buffer* array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false*. The conventions of TEX are followed; i.e., *ASCII_code* numbers representing the next line of the file are input into *buffer*[0], *buffer*[1], ..., *buffer*[*limit* − 1]; trailing blanks are ignored; and the global variable *limit* is set to the length of the line. The value of *limit* must be strictly less than *buf_size*.

We assume that none of the *ASCII_code* values of *buffer*[*j*] for $0 \le j < limit$ is equal to 0, ´*177*, *line_feed*, *form_feed*, or *carriage_return*.

```
function input_ln(var f : text_file): boolean;   { inputs a line or returns false }
  var final_limit: 0 .. buf_size;   { limit without trailing blanks }
  begin limit ← 0; final_limit ← 0;
  if eof(f) then input_ln ← false
  else begin while ¬eoln(f) do
        begin buffer[limit] ← xord[getc(f)]; incr(limit);
        if buffer[limit − 1] ≠ "␣" then final_limit ← limit;
        if limit = buf_size then
          begin while ¬eoln(f) do vgetc(f);
          decr(limit);   { keep buffer[buf_size] empty }
          if final_limit > limit then final_limit ← limit;
          print_nl(´!␣Input␣line␣too␣long´); loc ← 0; error;
          end;
        end;
    read_ln(f); limit ← final_limit; input_ln ← true;
    end;
  end;
```

34* The *jump_out* procedure just cuts across all active procedure levels and jumps out of the program.

```
define jump_out ≡ uexit(1)
define fatal_error(#) ≡
          begin new_line; write(stderr, #); error; mark_fatal; jump_out;
          end
```

38* `TANGLE` has been designed to avoid the need for indices that are more than sixteen bits wide, so that it can be used on most computers. But there are programs that need more than 65536 tokens, and some programs even need more than 65536 bytes; TeX is one of these. To get around this problem, a slight complication has been added to the data structures: *byte_mem* and *tok_mem* are two-dimensional arrays, whose first index is either 0 or 1 or 2. (For generality, the first index is actually allowed to run between 0 and $ww - 1$ in *byte_mem*, or between 0 and $zz - 1$ in *tok_mem*, where *ww* and *zz* are set to 2 and 3; the program will work for any positive values of *ww* and *zz*, and it can be simplified in obvious ways if $ww = 1$ or $zz = 1$.)

define $ww = 3$ { 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 ⟩ +≡
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.

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*.

```
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;
```

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

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

```
begin $i \gets id\_first$; $s \gets 0$; $h \gets 0$;
while $(i < id\_loc) \land (s < unambig\_length)$ do
  begin if $(buffer[i] \neq \texttt{"\_"}) \lor (allow\_underlines \land \neg strict\_mode)$ then
    begin if $(strict\_mode \lor force\_uppercase) \land (buffer[i] \geq \texttt{"a"})$ then $chopped\_id[s] \gets buffer[i] - {'40}$
    else if $(\neg strict\_mode \land force\_lowercase) \land (buffer[i] \geq \texttt{"A"}) \land (buffer[i] \leq \texttt{"Z"})$ then
        $chopped\_id[s] \gets buffer[i] + {'40}$
      else $chopped\_id[s] \gets buffer[i]$;
    $h \gets (h + h + chopped\_id[s]) \bmod hash\_size$; $incr(s)$;
    end;
  $incr(i)$;
  end;
$chopped\_id[s] \gets 0$;
end
```

This code is used in section 57.

63* ⟨Check if q conflicts with p 63*⟩ ≡

```
begin $k \gets byte\_start[q]$; $s \gets 0$; $w \gets q \bmod ww$;
while $(k < byte\_start[q + ww]) \land (s < unambig\_length)$ do
  begin $c \gets byte\_mem[w, k]$;
  if $c \neq \texttt{"\_"} \lor (allow\_underlines \land \neg strict\_mode)$ then
    begin if $(strict\_mode \lor force\_uppercase) \land (c \geq \texttt{"a"})$ then $c \gets c - {'40}$
    else if $(\neg strict\_mode \land force\_lowercase) \land (c \geq \texttt{"A"}) \land (c \leq \texttt{"Z"})$ then $c \gets c + {'40}$;
    if $chopped\_id[s] \neq c$ then goto not_found;
    $incr(s)$;
    end;
  $incr(k)$;
  end;
if $(k = byte\_start[q + ww]) \land (chopped\_id[s] \neq 0)$ then goto not_found;
print_nl(´!␣Identifier␣conflict␣with␣´);
for $k \gets byte\_start[q]$ to $byte\_start[q + ww] - 1$ do $print(xchr[byte\_mem[w, k]])$;
error; $q \gets 0$;    { only one conflict will be printed, since $equiv[0] = 0$ }
not_found: end
```

This code is used in section 62.

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

```
  define check_sum_prime ≡ ´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
      begin   { Change ".web" to ".pool" and use the current directory. }
      pool_name ← basename_change_suffix(web_name, ´.web´, ´.pool´); rewritebin(pool, pool_name);
      end;
    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] = """") ∨ (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.

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

```
procedure pop_level; { do this when cur_byte reaches cur_end }
  label exit;
  begin if text_link[cur_repl] = 0 then   { end of macro expansion }
    begin if (ilk[cur_name] = parametric) ∨ (ilk[cur_name] = parametric2) then
      ⟨ Remove a parameter from the parameter stack 91 ⟩;
    end
  else if text_link[cur_repl] < module_flag then   { link to a continuation }
      begin cur_repl ← text_link[cur_repl];   { we will stay on the same level }
      zo ← cur_repl mod zz; cur_byte ← tok_start[cur_repl]; cur_end ← tok_start[cur_repl + zz]; return;
      end;
  decr(stack_ptr);   { we will go down to the previous level }
  if stack_ptr > 0 then
    begin cur_state ← stack[stack_ptr]; zo ← cur_repl mod zz;
    end;
exit: end;
```

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

```
  begin case ilk[a] of
  normal: begin cur_val ← a; a ← identifier;
    end;
  numeric: begin cur_val ← equiv[a] − ´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.

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

```
⟨ Put a parameter on the parameter stack, or goto restart if error occurs 90* ⟩ ≡
  while (cur_byte = cur_end) ∧ (stack_ptr > 0) do pop_level;
  if (stack_ptr = 0) ∨ ((ilk[a] = parametric) ∧ (tok_mem[zo,
          cur_byte] ≠ "(")) ∨ ((ilk[a] = parametric2) ∧ (tok_mem[zo, cur_byte] ≠ "[")) then
    begin print_nl(´!␣No␣parameter␣given␣for␣´); print_id(a); error; goto restart;
    end;
  ⟨ Copy the parameter into tok_mem 93* ⟩;
  equiv[name_ptr] ← text_ptr; ilk[name_ptr] ← simple; w ← name_ptr mod ww; k ← byte_ptr[w];
  debug if k = max_bytes then overflow(´byte␣memory´);
  byte_mem[w, k] ← "#"; incr(k); byte_ptr[w] ← k;
  gubed   { this code has set the parameter identifier for debugging printouts }
  if name_ptr > max_names − ww then overflow(´name´);
  byte_start[name_ptr + ww] ← k; incr(name_ptr);
  if text_ptr > max_texts − zz then overflow(´text´);
  text_link[text_ptr] ← 0; tok_start[text_ptr + zz] ← tok_ptr[z]; incr(text_ptr); z ← text_ptr mod zz
```

This code is used in section 89*.

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

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

⟨ Copy the parameter into tok_mem 93* ⟩ ≡
  bal ← 1; incr(cur_byte);   { skip the opening '(' 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 = "´";   { copy string, don't change bal }
        othercases do_nothing
        endcases;
      app_repl(b);
      end;
    end;
done:
```

This code is used in section 90*.

105* ⟨Contribution is * or / or DIV or MOD 105*⟩ ≡

```
$((t = \mathit{ident}) \land (v = 3) \land (((\mathit{out\_contrib}[1] = \texttt{"D"}) \land (\mathit{out\_contrib}[2] = \texttt{"I"}) \land (\mathit{out\_contrib}[3] = \texttt{"V"})) \lor$
    $((\mathit{out\_contrib}[1] = \texttt{"d"}) \land (\mathit{out\_contrib}[2] = \texttt{"i"}) \land (\mathit{out\_contrib}[3] = \texttt{"v"})) \lor$
    $((\mathit{out\_contrib}[1] = \texttt{"M"}) \land (\mathit{out\_contrib}[2] = \texttt{"O"}) \land (\mathit{out\_contrib}[3] = \texttt{"D"})) \lor$
    $((\mathit{out\_contrib}[1] = \texttt{"m"}) \land (\mathit{out\_contrib}[2] = \texttt{"o"}) \land (\mathit{out\_contrib}[3] = \texttt{"d"})))) \lor$
    $((t = \mathit{misc}) \land ((v = \texttt{"*"}) \lor (v = \texttt{"/"})))$
```

This code is used in section 104.

110* ⟨If previous output was DIV or MOD, **goto** *bad_case* 110*⟩ ≡

```
if $(\mathit{out\_ptr} = \mathit{break\_ptr} + 3) \lor ((\mathit{out\_ptr} = \mathit{break\_ptr} + 4) \land (\mathit{out\_buf}[\mathit{break\_ptr}] = \texttt{"␣"}))$ then
  if $((\mathit{out\_buf}[\mathit{out\_ptr} - 3] = \texttt{"D"}) \land (\mathit{out\_buf}[\mathit{out\_ptr} - 2] = \texttt{"I"}) \land (\mathit{out\_buf}[\mathit{out\_ptr} - 1] = \texttt{"V"})) \lor$
  $((\mathit{out\_buf}[\mathit{out\_ptr} - 3] = \texttt{"d"}) \land (\mathit{out\_buf}[\mathit{out\_ptr} - 2] = \texttt{"i"}) \land (\mathit{out\_buf}[\mathit{out\_ptr} - 1] = \texttt{"v"})) \lor$
  $((\mathit{out\_buf}[\mathit{out\_ptr} - 3] = \texttt{"M"}) \land (\mathit{out\_buf}[\mathit{out\_ptr} - 2] = \texttt{"O"}) \land (\mathit{out\_buf}[\mathit{out\_ptr} - 1] = \texttt{"D"})) \lor$
  $((\mathit{out\_buf}[\mathit{out\_ptr} - 3] = \texttt{"m"}) \land (\mathit{out\_buf}[\mathit{out\_ptr} - 2] = \texttt{"o"}) \land (\mathit{out\_buf}[\mathit{out\_ptr} - 1] = \texttt{"d"}))$ then
  goto $\mathit{bad\_case}$
```

This code is used in section 107.

114* ⟨ Cases like <> and := 114* ⟩ ≡

```
and_sign: begin out_contrib[1] ← "a"; out_contrib[2] ← "n"; out_contrib[3] ← "d"; send_out(ident, 3);
  end;
not_sign: begin out_contrib[1] ← "n"; out_contrib[2] ← "o"; out_contrib[3] ← "t"; send_out(ident, 3);
  end;
set_element_sign: begin out_contrib[1] ← "i"; out_contrib[2] ← "n"; send_out(ident, 2);
  end;
or_sign: begin out_contrib[1] ← "o"; out_contrib[2] ← "r"; send_out(ident, 2);
  end;
left_arrow: begin out_contrib[1] ← ":"; out_contrib[2] ← "="; send_out(str, 2);
  end;
not_equal: begin out_contrib[1] ← "<"; out_contrib[2] ← ">"; send_out(str, 2);
  end;
less_or_equal: begin out_contrib[1] ← "<"; out_contrib[2] ← "="; send_out(str, 2);
  end;
greater_or_equal: begin out_contrib[1] ← ">"; out_contrib[2] ← "="; send_out(str, 2);
  end;
equivalence_sign: begin out_contrib[1] ← "="; out_contrib[2] ← "="; send_out(str, 2);
  end;
double_dot: begin out_contrib[1] ← "."; out_contrib[2] ← "."; send_out(str, 2);
  end;
```

This code is used in section 113.

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.

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

```
  define digits ≡ "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
⟨Cases related to constants, possibly leading to get_fraction or reswitch 119*⟩ ≡
digits: begin n ← 0;
  repeat cur_char ← cur_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;
```

This code is used in section 113.

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

```
define add_in(#) ≡
        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⟩;
    "#": 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
```

This code is used in section 165*.

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

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

This code is used in section 172.

174* ⟨If the next text is '(#)==' 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;
  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 $ddt \le 0$; thus you type $2n + 1$ integers, ending with zero or a negative number. Then control either passes to the breakpoint, allowing you to look at and/or change variables (if you typed zero), or to exit the routine (if you typed a negative value).

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

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

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

```
⟨Set initial values 10⟩ +≡
  debug trouble_shooting ← true; debug_cycle ← 1; debug_skipped ← 0;
  trouble_shooting ← false; debug_cycle ← 99999;  { use these when it almost works }
  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 }
  ⟨Print the job history 187⟩;
  new_line;
  if (history ≠ spotless) ∧ (history ≠ harmless_message) 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 189*⟩;
  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; {Now optind is the index of first non-option on the command line.}
  if (optind + 1 ≠ argc) ∧ (optind + 2 ≠ argc) then
    begin write_ln(stderr, my_name, ´:␣Need␣one␣or␣two␣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 chg_name ← extend_filename(cmdline(optind + 1), ´ch´);
    end;  { Change ".web" to ".p" and use the current directory. }
  pascal_name ← basename_change_suffix(web_name, ´.web´, ´.p´);
  end;
```

This code is used in section 2*.

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

```
⟨ Define the option table 189* ⟩ ≡
  current_option ← 0; long_options[current_option].name ← ´help´;
  long_options[current_option].has_arg ← 0; long_options[current_option].flag ← 0;
  long_options[current_option].val ← 0; incr(current_option);
```

See also sections 190*, 191*, 192*, 193*, 194*, 195*, 196*, 197*, and 198*.

This code is used in section 188*.

190*. Another of the standard options.

```
⟨ Define the option table 189* ⟩ +≡
  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);
```

191*. Use all mixed case.

```
⟨ Define the option table 189* ⟩ +≡
  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);
```

192*. Use all uppercase.

```
⟨ Define the option table 189* ⟩ +≡
  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);
```

193*. Use all lowercase.

```
⟨ Define the option table 189* ⟩ +≡
  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);
```

194*. Allow underlines.

```
⟨ Define the option table 189* ⟩ +≡
  long_options[current_option].name ← ´underlines´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

195*. Strict comparisons.

```
⟨ Define the option table 189* ⟩ +≡
  long_options[current_option].name ← ´strict´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

196*. Loose comparisons.

```
⟨ Define the option table 189* ⟩ +≡
  long_options[current_option].name ← ´loose´; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

197* Loose comparisons.

```
⟨ Define the option table 189* ⟩ +≡
  long_options[current_option].name ← ´length´; long_options[current_option].has_arg ← 1;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0; incr(current_option);
```

198* An element with all zeros always ends the list.

```
⟨ Define the option table 189* ⟩ +≡
  long_options[current_option].name ← 0; long_options[current_option].has_arg ← 0;
  long_options[current_option].flag ← 0; long_options[current_option].val ← 0;
```

199* Global filenames.

```
⟨ Globals in the outer block 9 ⟩ +≡
web_name, chg_name, pascal_name, pool_name: const_c_string;
force_uppercase, force_lowercase, allow_underlines, strict_mode: boolean;
unambig_length: 0 .. max_id_length;
```

200*. 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.