COMPUTERS & TYPESETTING

JUBILEE 2021 edition

DONALD E. KNUTH

COMPUTERS & TYPESETTING / B

TEX: The Program

402. Basic scanning subroutines. Let's turn now to some procedures that TEX calls upon frequently to digest certain kinds of patterns in the input. Most of these are quite simple; some are quite elaborate. Almost all of the routines call *get_x_token*, which can cause them to be invoked recursively.

403. The *scan_left_brace* routine is called when a left brace is supposed to be the next non-blank token. (The term "left brace" means, more precisely, a character whose catcode is *left_brace*.) TEX allows `\relax` to appear before the *left_brace*.

```
procedure scan_left_brace; { reads a mandatory left_brace }
  begin ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
  if cur_cmd ≠ left_brace then
    begin print_err("Missing␣{␣inserted");
    help4("A␣left␣brace␣was␣mandatory␣here,␣so␣I´ve␣put␣one␣in.")
    ("You␣might␣want␣to␣delete␣and/or␣insert␣some␣corrections")
    ("so␣that␣I␣will␣find␣a␣matching␣right␣brace␣soon.")
    ("(If␣you´re␣confused␣by␣all␣this,␣try␣typing␣`I}´␣now.)"); back_error;
    cur_tok ← left_brace_token + "{"; cur_cmd ← left_brace; cur_chr ← "{"; incr(align_state);
    end;
  end;
```

404. ⟨ Get the next non-blank non-relax non-call token 404 ⟩ ≡

```
repeat get_x_token;
until (cur_cmd ≠ spacer) ∧ (cur_cmd ≠ relax)
```

This code is used in sections 403, 1078, 1084, 1151, 1160, 1211, 1226, and 1270.

405. The *scan_optional_equals* routine looks for an optional '`=`' sign preceded by optional spaces; '`\relax`' is not ignored here.

```
procedure scan_optional_equals;
  begin ⟨ Get the next non-blank non-call token 406 ⟩;
  if cur_tok ≠ other_token + "=" then back_input;
  end;
```

---

align_state: *integer*, §309.
back_error: **procedure**, §327.
back_input: **procedure**, §325.
begin_diagnostic: **procedure**, §245.
cur_chr: *halfword*, §297.
cur_cmd: *eight_bits*, §297.
cur_tok: *halfword*, §297.
end_diagnostic: **procedure**, §245.
free_avail = macro, §121.
get_x_token: **procedure**, §380.
help4 = macro, §79.
incr = macro, §16.
info = macro, §118.
left_brace = 1, §207.
left_brace_token = 256, §289.
link = macro, §118.
m: *halfword*, §389.
match_chr: *ASCII_code*, §389.
n: *small_number*, §389.
null = macro, §115.
other_token = 3072, §289.
p: *pointer*, §389.
print: **procedure**, §59.
print_cs: **procedure**, §262.
print_err = macro, §73.
print_int: **procedure**, §65.
print_ln: **procedure**, §57.
print_nl: **procedure**, §62.
pstack: **array**, §388.
rbrace_ptr: *pointer*, §389.
ref_count: *pointer*, §389.
relax = 0, §207.
right_brace_limit = 768, §289.
show_token_list: **procedure**, §292.
spacer = 10, §207.
temp_head = macro, §162.
token_show: **procedure**, §295.
tracing_macros = macro, §236.
warning_index: *pointer*, §305.

406. ⟨ Get the next non-blank non-call token 406 ⟩ ≡

```
repeat get_x_token;
until cur_cmd ≠ spacer
```

This code is used in sections 405, 441, 455, 503, 526, 577, 785, 791, and 1045.

407. In case you are getting bored, here is a slightly less trivial routine: Given a string of lowercase letters, like '`pt`' or '`plus`' or '`width`', the *scan_keyword* routine checks to see whether the next tokens of input match this string. The match must be exact, except that uppercase letters will match their lowercase counterparts; uppercase equivalents are determined by subtracting `"a" − "A"`, rather than using the *uc_code* table, since TEX uses this routine only for its own limited set of keywords.

If a match is found, the characters are effectively removed from the input and *true* is returned. Otherwise *false* is returned, and the input is left essentially unchanged (except for the fact that some macros may have been expanded, etc.).

```
function scan_keyword(s : str_number): boolean; { look for a given string }
  label exit;
  var p: pointer; { tail of the backup list }
    q: pointer; { new node being added to the token list via store_new_token }
    k: pool_pointer; { index into str_pool }
  begin p ← backup_head; link(p) ← null; k ← str_start[s];
  while k < str_start[s + 1] do
    begin get_x_token; { recursion is possible here }
    if (cur_cs = 0) ∧ ((cur_chr = so(str_pool[k])) ∨ (cur_chr = so(str_pool[k]) − "a" + "A")) then
      begin store_new_token(cur_tok); incr(k);
      end
    else if (cur_cmd ≠ spacer) ∨ (p ≠ backup_head) then
        begin back_input;
        if p ≠ backup_head then back_list(link(backup_head));
        scan_keyword ← false; return;
        end;
    end;
  flush_list(link(backup_head)); scan_keyword ← true;
exit: end;
```

408. Here is a procedure that sounds an alarm when mu and non-mu units are being switched.

```
procedure mu_error;
  begin print_err("Incompatible␣glue␣units");
  help1("I´m␣going␣to␣assume␣that␣1mu=1pt␣when␣they´re␣mixed."); error;
  end;
```

409. The next routine '*scan_something_internal*' is used to fetch internal numeric quantities like '`\hsize`', and also to handle the '`\the`' when expanding constructions like '`\the\toks0`' and '`\the\baselineskip`'. Soon we will be considering the *scan_int* procedure, which calls *scan_something_internal*; on the other hand, *scan_something_internal* also calls *scan_int*, for constructions like '`\catcode`\$`' or '`\fontdimen 3 \ff`'. So we have to declare *scan_int* as a *forward* procedure. A few other procedures are also declared at this point.

```
procedure scan_int; forward; { scans an integer value }
```

⟨ Declare procedures that scan restricted classes of integers 433 ⟩
⟨ Declare procedures that scan font-related stuff 577 ⟩

410. TEX doesn't know exactly what to expect when *scan_something_internal* begins. For example, an integer or dimension or glue value could occur immediately after '`\hskip`'; and one can even say `\the` with respect to token lists in constructions like '`\xdef\o{\the\output}`'. On the other hand, only integers are allowed after a construction like '`\count`'. To handle the various possibilities, *scan_something_internal* has a *level* parameter, which tells the "highest" kind of quantity that *scan_something_internal* is allowed to produce. Six levels are distinguished, namely *int_val*, *dimen_val*, *glue_val*, *mu_val*, *ident_val*, and *tok_val*.

The output of *scan_something_internal* (and of the other routines *scan_int*, *scan_dimen*, and *scan_glue* below) is put into the global variable *cur_val*, and its level is put into *cur_val_level*. The highest values of *cur_val_level* are special: *mu_val* is used only when *cur_val* points to something in a "muskip" register, or to one of the three parameters `\thinmuskip`, `\medmuskip`, `\thickmuskip`; *ident_val* is used only when *cur_val* points to a font identifier; *tok_val* is used only when *cur_val* points to *null* or to the reference count of a token list. The last two cases are allowed only when *scan_something_internal* is called with *level* = *tok_val*.

If the output is glue, *cur_val* will point to a glue specification, and the reference count of that glue will have been updated to reflect this reference; if the output is a nonempty token list, *cur_val* will point to its reference count, but in this case the count will not have been updated. Otherwise *cur_val* will contain the integer or scaled value in question.

```
define int_val = 0   { integer values }
define dimen_val = 1   { dimension values }
define glue_val = 2   { glue specifications }
define mu_val = 3   { math glue specifications }
define ident_val = 4   { font identifier }
define tok_val = 5   { token lists }
```

⟨ Global variables 13 ⟩ +≡
cur_val: *integer*; { value returned by numeric scanners }
cur_val_level: *int_val* .. *tok_val*; { the "level" of this value }

---

back_input: **procedure**, §325.
back_list = macro, §323.
backup_head = macro, §162.
cur_chr: *halfword*, §297.
cur_cmd: *eight_bits*, §297.
cur_cs: *pointer*, §297.
cur_tok: *halfword*, §297.
error: **procedure**, §82.
exit = 10, §15.
flush_list: **procedure**, §123.
get_x_token: **procedure**, §380.
help1 = macro, §79.
incr = macro, §16.
level: *small_number*, §413.
link = macro, §118.
null = macro, §115.
pointer = macro, §115.
pool_pointer = 0 .. *pool_size*, §38.
print_err = macro, §73.
scan_dimen: **procedure**, §448.
scan_glue: **procedure**, §461.
scan_int: **procedure**, §440.
scan_something_internal: **procedure**, §413.
so = macro, §38.
spacer = 10, §207.
store_new_token = macro, §371.
str_number = 0 .. *max_strings*, §38.
str_pool: **packed array**, §39.
str_start: **array**, §39.
uc_code = macro, §230.

411. The hash table is initialized with '\count', '\dimen', '\skip', and '\muskip' all having *register* as their command code; they are distinguished by the *chr_code*, which is either *int_val*, *dimen_val*, *glue_val*, or *mu_val*.

```
⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive("count", register, int_val); primitive("dimen", register, dimen_val);
  primitive("skip", register, glue_val); primitive("muskip", register, mu_val);
```

412.
```
⟨ Cases of print_cmd_chr for symbolic printing of primitives 227 ⟩ +≡
register: if chr_code = int_val then print_esc("count")
  else if chr_code = dimen_val then print_esc("dimen")
    else if chr_code = glue_val then print_esc("skip")
      else print_esc("muskip");
```

413. OK, we're ready for *scan_something_internal* itself. A second parameter, *negative*, is set *true* if the value that is found should be negated. It is assumed that *cur_cmd* and *cur_chr* represent the first token of the internal quantity to be scanned; an error will be signalled if *cur_cmd* < *min_internal* or *cur_cmd* > *max_internal*.

```
  define scanned_result_end(#) ≡ cur_val_level ← #; end
  define scanned_result(#) ≡ begin cur_val ← #; scanned_result_end
procedure scan_something_internal(level : small_number; negative : boolean);
        { fetch an internal parameter }
  var m: halfword;  { chr_code part of the operand token }
    p: 0 .. nest_size;  { index into nest }
  begin m ← cur_chr;
  case cur_cmd of
  def_code: ⟨ Fetch a character code from some table 414 ⟩;
  toks_register, assign_toks, def_family, set_font, def_font: ⟨ Fetch a token list or font identifier, provided
        that level = tok_val 415 ⟩;
  assign_int: scanned_result(eqtb[m].int)(int_val);
  assign_dimen: scanned_result(eqtb[m].sc)(dimen_val);
  assign_glue: scanned_result(equiv(m))(glue_val);
  assign_mu_glue: scanned_result(equiv(m))(mu_val);
  set_aux: ⟨ Fetch the space_factor or the prev_depth 418 ⟩;
  set_prev_graf: ⟨ Fetch the prev_graf 422 ⟩;
  set_page_int: ⟨ Fetch the dead_cycles or the insert_penalties 419 ⟩;
  set_page_dimen: ⟨ Fetch something on the page_so_far 421 ⟩;
  set_shape: ⟨ Fetch the par_shape size 423 ⟩;
  set_box_dimen: ⟨ Fetch a box dimension 420 ⟩;
  char_given, math_given: scanned_result(cur_chr)(int_val);
  assign_font_dimen: ⟨ Fetch a font dimension 425 ⟩;
  assign_font_int: ⟨ Fetch a font integer 426 ⟩;
  register: ⟨ Fetch a register 427 ⟩;
  last_item: ⟨ Fetch an item in the current node, if appropriate 424 ⟩;
  othercases ⟨ Complain that \the can't do this; give zero result 428 ⟩
  endcases;
  while cur_val_level > level do ⟨ Convert cur_val to a lower level 429 ⟩;
  ⟨ Fix the reference count, if any, and negate cur_val if negative 430 ⟩;
  end;
```

414. ⟨ Fetch a character code from some table 414 ⟩ ≡

```
begin scan_char_num;
if m = math_code_base then scanned_result(ho(math_code(cur_val)))(int_val)
else if m < math_code_base then scanned_result(equiv(m + cur_val))(int_val)
  else scanned_result(eqtb[m + cur_val].int)(int_val);
end
```

This code is used in section 413.

415. ⟨ Fetch a token list or font identifier, provided that $level = tok\_val$ 415 ⟩ ≡

```
if level ≠ tok_val then
  begin print_err("Missing␣number,␣treated␣as␣zero");
  help3("A␣number␣should␣have␣been␣here;␣I␣inserted␣`0´.")
  ("(If␣you␣can´t␣figure␣out␣why␣I␣needed␣to␣see␣a␣number,")
  ("look␣up␣`weird␣error´␣in␣the␣index␣to␣The␣TeXbook.)"); back_error;
  scanned_result(0)(dimen_val);
  end
else if cur_cmd ≤ assign_toks then
    begin if cur_cmd < assign_toks then { cur_cmd = toks_register }
      begin scan_eight_bit_int; m ← toks_base + cur_val;
      end;
    scanned_result(equiv(m))(tok_val);
    end
  else begin back_input; scan_font_ident; scanned_result(font_id_base + cur_val)(ident_val);
    end
```

This code is used in section 413.

---

416. Users refer to '`\the\spacefactor`' only in horizontal mode, and to '`\the\prevdepth`' only in vertical mode; so we put the associated mode in the modifier part of the *set_aux* command. The *set_page_int* command has modifier 0 or 1, for '`\deadcycles`' and '`\insertpenalties`', respectively. The *set_box_dimen* command is modified by either *width_offset*, *height_offset*, or *depth_offset*. And the *last_item* command is modified by either *int_val*, *dimen_val*, *glue_val*, *input_line_no_code*, or *badness_code*.

```
define input_line_no_code = glue_val + 1  { code for \inputlineno }
define badness_code = glue_val + 2  { code for \badness }
```

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡

```
primitive("spacefactor", set_aux, hmode); primitive("prevdepth", set_aux, vmode);
primitive("deadcycles", set_page_int, 0); primitive("insertpenalties", set_page_int, 1);
primitive("wd", set_box_dimen, width_offset); primitive("ht", set_box_dimen, height_offset);
primitive("dp", set_box_dimen, depth_offset); primitive("lastpenalty", last_item, int_val);
primitive("lastkern", last_item, dimen_val); primitive("lastskip", last_item, glue_val);
primitive("inputlineno", last_item, input_line_no_code);
primitive("badness", last_item, badness_code);
```

417. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
set_aux: if chr_code = vmode then print_esc("prevdepth") else print_esc("spacefactor");
set_page_int: if chr_code = 0 then print_esc("deadcycles") else print_esc("insertpenalties");
set_box_dimen: if chr_code = width_offset then print_esc("wd")
  else if chr_code = height_offset then print_esc("ht")
    else print_esc("dp");
last_item: case chr_code of
  int_val: print_esc("lastpenalty");
  dimen_val: print_esc("lastkern");
  glue_val: print_esc("lastskip");
  input_line_no_code: print_esc("inputlineno");
  othercases print_esc("badness")
  endcases;
```

418. ⟨ Fetch the *space_factor* or the *prev_depth* 418 ⟩ ≡

```
if abs(mode) ≠ m then
  begin print_err("Improper "); print_cmd_chr(set_aux, m);
  help4("You can refer to \spacefactor only in horizontal mode;")
  ("you can refer to \prevdepth only in vertical mode; and")
  ("neither of these is meaningful inside \write. So")
  ("I'm forgetting what you said and using zero instead."); error;
  if level ≠ tok_val then scanned_result(0)(dimen_val)
  else scanned_result(0)(int_val);
  end
else if m = vmode then scanned_result(prev_depth)(dimen_val)
  else scanned_result(space_factor)(int_val)
```

This code is used in section 413.

419. ⟨ Fetch the *dead_cycles* or the *insert_penalties* 419 ⟩ ≡

```
begin if m = 0 then cur_val ← dead_cycles else cur_val ← insert_penalties;
cur_val_level ← int_val;
```