1* Introduction. This is ε-TEX, a program derived from and extending the capabilities of TEX, a document compiler intended to produce typesetting of high quality. The Pascal program that follows is the definition of TEX82, a standard version of TEX that is designed to be highly portable so that identical output will be obtainable on a great variety of computers.

The main purpose of the following program is to explain the algorithms of TEX as clearly as possible. As a result, the program will not necessarily be very efficient when a particular Pascal compiler has translated it into a particular machine language. However, the program has been written so that it can be tuned to run efficiently in a wide variety of operating environments by making comparatively few changes. Such flexibility is possible because the documentation that follows is written in the `WEB` language, which is at a higher level than Pascal; the preprocessing step that converts `WEB` to Pascal is able to introduce most of the necessary refinements. Semi-automatic translation to other languages is also feasible, because the program below does not make extensive use of features that are peculiar to Pascal.

A large piece of software like TEX has inherent complexity that cannot be reduced below a certain level of difficulty, although each individual part is fairly simple by itself. The `WEB` language is intended to make the algorithms as readable as possible, by reflecting the way the individual program pieces fit together and by providing the cross-references that connect different parts. Detailed comments about what is going on, and about why things were done in certain ways, have been liberally sprinkled throughout the program. These comments explain features of the implementation, but they rarely attempt to explain the TEX language itself, since the reader is supposed to be familiar with *The TEXbook*.

2* The present implementation has a long ancestry, beginning in the summer of 1977, when Michael F. Plass and Frank M. Liang designed and coded a prototype based on some specifications that the author had made in May of that year. This original protoTeX included macro definitions and elementary manipulations on boxes and glue, but it did not have line-breaking, page-breaking, mathematical formulas, alignment routines, error recovery, or the present semantic nest; furthermore, it used character lists instead of token lists, so that a control sequence like `\halign` was represented by a list of seven characters. A complete version of TeX was designed and coded by the author in late 1977 and early 1978; that program, like its prototype, was written in the SAIL language, for which an excellent debugging system was available. Preliminary plans to convert the SAIL code into a form somewhat like the present "web" were developed by Luis Trabb Pardo and the author at the beginning of 1979, and a complete implementation was created by Ignacio A. Zabala in 1979 and 1980. The TeX82 program, which was written by the author during the latter part of 1981 and the early part of 1982, also incorporates ideas from the 1979 implementation of TeX in MESA that was written by Leonidas Guibas, Robert Sedgewick, and Douglas Wyatt at the Xerox Palo Alto Research Center. Several hundred refinements were introduced into TeX82 based on the experiences gained with the original implementations, so that essentially every part of the system has been substantially improved. After the appearance of "Version 0" in September 1982, this program benefited greatly from the comments of many other people, notably David R. Fuchs and Howard W. Trickey. A final revision in September 1989 extended the input character set to eight-bit codes and introduced the ability to hyphenate words from different languages, based on some ideas of Michael J. Ferguson.

No doubt there still is plenty of room for improvement, but the author is firmly committed to keeping TeX82 "frozen" from now on; stability and reliability are to be its main virtues.

On the other hand, the `WEB` description can be extended without changing the core of TeX82 itself, and the program has been designed so that such extensions are not extremely difficult to make. The *banner* string defined here should be changed whenever TeX undergoes any modifications, so that it will be clear which version of TeX might be the guilty party when a problem arises.

This program contains code for various features extending TeX, therefore this program is called 'ε-TeX' and not 'TeX'; the official name 'TeX' by itself is reserved for software systems that are fully compatible with each other. A special test suite called the "`TRIP` test" is available for helping to determine whether a particular implementation deserves to be known as 'TeX' [cf. Stanford Computer Science report CS1027, November 1984].

A similar test suite called the "`e-TRIP` test" is available for helping to determine whether a particular implementation deserves to be known as 'ε-TeX'.

```
define eTeX_version = 2 { \eTeXversion }
define eTeX_revision ≡ ".6" { \eTeXrevision }
define eTeX_version_string ≡ ´-2.6´ {current ε-TeX version}
define eTeX_banner ≡ ´This␣is␣e-TeX,␣Version␣3.14159265´, eTeX_version_string
          {printed when ε-TeX starts}
define TeX_banner ≡ ´This␣is␣TeX,␣Version␣3.14159265´ {printed when TeX starts}
define banner ≡ eTeX_banner
define TEX ≡ ETEX {change program name into ETEX }
define TeXXeT_code = 0 {the TeX--XeT feature is optional}
define eTeX_states = 1 {number of ε-TeX state variables in eqtb }
```

3*. Different Pascals have slightly different conventions, and the present program expresses TEX in terms of the Pascal that was available to the author in 1982. Constructions that apply to this particular compiler, which we shall call Pascal-H, should help the reader see how to make an appropriate interface for other systems if necessary. (Pascal-H is Charles Hedrick's modification of a compiler for the DECsystem-10 that was originally developed at the University of Hamburg; cf. *SOFTWARE—Practice & Experience* **6** (1976), 29–42. The TEX program below is intended to be adaptable, without extensive changes, to most other versions of Pascal, so it does not fully use the admirable features of Pascal-H. Indeed, a conscious effort has been made here to avoid using several idiosyncratic features of standard Pascal itself, so that most of the code can be translated mechanically into other high-level languages. For example, the '**with**' and '*new*' features are not used, nor are pointer types, set types, or enumerated scalar types; there are no '**var**' parameters, except in the case of files — ε-TEX, however, does use '**var**' parameters for the *reverse* function; there are no tag fields on variant records; there are no assignments *real* ← *integer*; no procedures are declared local to other procedures.)

The portions of this program that involve system-dependent code, where changes might be necessary because of differences between Pascal compilers and/or differences between operating systems, can be identified by looking at the sections whose numbers are listed under 'system dependencies' in the index. Furthermore, the index entries for 'dirty Pascal' list all places where the restrictions of Pascal have not been followed perfectly, for one reason or another.

Incidentally, Pascal's standard *round* function can be problematical, because it disagrees with the IEEE floating-point standard. Many implementors have therefore chosen to substitute their own home-grown rounding procedure.

15*. Labels are given symbolic names by the following definitions, so that occasional **goto** statements will be meaningful. We insert the label '*exit*' just before the '**end**' of a procedure in which we have used the '**return**' statement defined below; the label '*restart*' is occasionally used at the very beginning of a procedure; and the label '*reswitch*' is occasionally used just prior to a **case** statement in which some cases change the conditions and we wish to branch to the newly applicable case. Loops that are set up with the **loop** construction defined below are commonly exited by going to '*done*' or to '*found*' or to '*not_found*', and they are sometimes repeated by going to '*continue*'. If two or more parts of a subroutine start differently but end up the same, the shared code may be gathered together at '*common_ending*'.

Incidentally, this program never declares a label that isn't actually used, because some fussy Pascal compilers will complain about redundant labels.

```
define exit = 10 { go here to leave a procedure }
define restart = 20 { go here to start a procedure again }
define reswitch = 21 { go here to start a case statement again }
define continue = 22 { go here to resume a loop }
define done = 30 { go here to exit a loop }
define done1 = 31 { like done, when there is more than one loop }
define done2 = 32 { for exiting the second loop in a long block }
define done3 = 33 { for exiting the third loop in a very long block }
define done4 = 34 { for exiting the fourth loop in an extremely long block }
define done5 = 35 { for exiting the fifth loop in an immense block }
define done6 = 36 { for exiting the sixth loop in a block }
define found = 40 { go here when you've found it }
define found1 = 41 { like found, when there's more than one per routine }
define found2 = 42 { like found, when there's more than two per routine }
define not_found = 45 { go here when you've found nothing }
define not_found1 = 46 { like not_found, when there's more than one }
define not_found2 = 47 { like not_found, when there's more than two }
define not_found3 = 48 { like not_found, when there's more than three }
define not_found4 = 49 { like not_found, when there's more than four }
define common_ending = 50 { go here when you want to merge with another branch }
```

135*. An *hlist_node* stands for a box that was made from a horizontal list. Each *hlist_node* is seven words long, and contains the following fields (in addition to the mandatory *type* and *link*, which we shall not mention explicitly when discussing the other node types): The *height* and *width* and *depth* are scaled integers denoting the dimensions of the box. There is also a *shift_amount* field, a scaled integer indicating how much this box should be lowered (if it appears in a horizontal list), or how much it should be moved to the right (if it appears in a vertical list). There is a *list_ptr* field, which points to the beginning of the list from which this box was fabricated; if *list_ptr* is *null*, the box is empty. Finally, there are three fields that represent the setting of the glue: *glue_set*(*p*) is a word of type *glue_ratio* that represents the proportionality constant for glue setting; *glue_sign*(*p*) is *stretching* or *shrinking* or *normal* depending on whether or not the glue should stretch or shrink or remain rigid; and *glue_order*(*p*) specifies the order of infinity to which glue setting applies (*normal*, *fil*, *fill*, or *filll*). The *subtype* field is not used in TEX. In ε-TEX the *subtype* field records the box direction mode *box_lr*.

define *hlist_node* = 0 { *type* of hlist nodes }

define *box_node_size* = 7 { number of words to allocate for a box node }

define *width_offset* = 1 { position of *width* field in a box node }

define *depth_offset* = 2 { position of *depth* field in a box node }

define *height_offset* = 3 { position of *height* field in a box node }

define *width*(#) ≡ *mem*[# + *width_offset*].*sc* { width of the box, in sp }

define *depth*(#) ≡ *mem*[# + *depth_offset*].*sc* { depth of the box, in sp }

define *height*(#) ≡ *mem*[# + *height_offset*].*sc* { height of the box, in sp }

define *shift_amount*(#) ≡ *mem*[# + 4].*sc* { repositioning distance, in sp }

define *list_offset* = 5 { position of *list_ptr* field in a box node }

define *list_ptr*(#) ≡ *link*(# + *list_offset*) { beginning of the list inside the box }

define *glue_order*(#) ≡ *subtype*(# + *list_offset*) { applicable order of infinity }

define *glue_sign*(#) ≡ *type*(# + *list_offset*) { stretching or shrinking }

define *normal* = 0 { the most common case when several cases are named }

define *stretching* = 1 { glue setting applies to the stretch components }

define *shrinking* = 2 { glue setting applies to the shrink components }

define *glue_offset* = 6 { position of *glue_set* in a box node }

define *glue_set*(#) ≡ *mem*[# + *glue_offset*].*gr* { a word of type *glue_ratio* for glue setting }

141*. A *mark_node* has a *mark_ptr* field that points to the reference count of a token list that contains the user's `\mark` text. In addition there is a *mark_class* field that contains the mark class.

define *mark_node* = 4 { *type* of a mark node }

define *small_node_size* = 2 { number of words to allocate for most node types }

define *mark_ptr*(#) ≡ *link*(# + 1) { head of the token list for a mark }

define *mark_class*(#) ≡ *info*(# + 1) { the mark class }

142*. An *adjust_node*, which occurs only in horizontal lists, specifies material that will be moved out into the surrounding vertical list; i.e., it is used to implement TEX's '`\vadjust`' operation. The *adjust_ptr* field points to the vlist containing this material.

define *adjust_node* = 5 { *type* of an adjust node }

define *adjust_ptr*(#) ≡ *mem*[# + 1].*int* { vertical list to be moved out of horizontal list }

147.* A *math_node*, which occurs only in horizontal lists, appears before and after mathematical formulas. The *subtype* field is *before* before the formula and *after* after it. There is a *width* field, which represents the amount of surrounding space inserted by `\mathsurround`.

In addition a *math_node* with *subtype* $>$ *after* and *width* $= 0$ will be (ab)used to record a regular *math_node* reinserted after being discarded at a line break or one of the text direction primitives (`\beginL`, `\endL`, `\beginR`, and `\endR`).

```
  define math_node = 9  { type of a math node }
  define before = 0  { subtype for math node that introduces a formula }
  define after = 1  { subtype for math node that winds up a formula }

  define M_code = 2
  define begin_M_code = M_code + before  { subtype for \beginM node }
  define end_M_code = M_code + after  { subtype for \endM node }
  define L_code = 4
  define begin_L_code = L_code + begin_M_code  { subtype for \beginL node }
  define end_L_code = L_code + end_M_code  { subtype for \endL node }
  define R_code = L_code + L_code
  define begin_R_code = R_code + begin_M_code  { subtype for \beginR node }
  define end_R_code = R_code + end_M_code  { subtype for \endR node }

  define end_LR(#) ≡ odd(subtype(#))
  define end_LR_type(#) ≡ (L_code ∗ (subtype(#) div L_code) + end_M_code)
  define begin_LR_type(#) ≡ (# − after + before)

function new_math(w : scaled; s : small_number): pointer;
  var p: pointer;  { the new node }
  begin p ← get_node(small_node_size); type(p) ← math_node; subtype(p) ← s; width(p) ← w;
  new_math ← p;
  end;
```

175* ⟨ Print a short indication of the contents of node *p* 175* ⟩ ≡

```
case type(p) of
hlist_node, vlist_node, ins_node, whatsit_node, mark_node, adjust_node, unset_node: print("[]");
rule_node: print_char("|");
glue_node: if glue_ptr(p) ≠ zero_glue then print_char("␣");
math_node: if subtype(p) ≥ L_code then print("[]")
  else print_char("$");
ligature_node: short_display(lig_ptr(p));
disc_node: begin short_display(pre_break(p)); short_display(post_break(p));
  n ← replace_count(p);
  while n > 0 do
    begin if link(p) ≠ null then p ← link(p);
    decr(n);
    end;
  end;
othercases do_nothing
endcases
```

This code is used in section 174.

184* ⟨ Display box *p* 184* ⟩ ≡

```
begin if type(p) = hlist_node then print_esc("h")
else if type(p) = vlist_node then print_esc("v")
  else print_esc("unset");
print("box("); print_scaled(height(p)); print_char("+"); print_scaled(depth(p)); print(")x");
print_scaled(width(p));
if type(p) = unset_node then ⟨ Display special fields of the unset node p 185 ⟩
else begin ⟨ Display the value of glue_set(p) 186 ⟩;
  if shift_amount(p) ≠ 0 then
    begin print(",␣shifted␣"); print_scaled(shift_amount(p));
    end;
  if eTeX_ex then ⟨ Display if this box is never to be reversed 1435* ⟩;
  end;
node_list_display(list_ptr(p));  { recursive call }
end
```

This code is used in section 183.

192* ⟨ Display math node *p* 192* ⟩ ≡

```
if subtype(p) > after then
  begin if end_LR(p) then print_esc("end")
  else print_esc("begin");
  if subtype(p) > R_code then print_char("R")
  else if subtype(p) > L_code then print_char("L")
    else print_char("M");
  end
else begin print_esc("math");
  if subtype(p) = before then print("on")
  else print("off");
  if width(p) ≠ 0 then
    begin print(",␣surrounded␣"); print_scaled(width(p));
    end;
  end
```

This code is used in section 183.

196* ⟨Display mark p 196*⟩ ≡

```
begin print_esc("mark");
if mark_class(p) ≠ 0 then
   begin print_char("s"); print_int(mark_class(p));
   end;
print_mark(mark_ptr(p));
end
```

This code is used in section 183.

208* Next are the ordinary run-of-the-mill command codes. Codes that are *min_internal* or more represent internal quantities that might be expanded by '\the'.

```
define char_num = 16  { character specified numerically ( \char ) }
define math_char_num = 17  { explicit math code ( \mathchar ) }
define mark = 18  { mark definition ( \mark ) }
define xray = 19  { peek inside of TEX ( \show, \showbox, etc. ) }
define make_box = 20  { make a box ( \box, \copy, \hbox, etc. ) }
define hmove = 21  { horizontal motion ( \moveleft, \moveright ) }
define vmove = 22  { vertical motion ( \raise, \lower ) }
define un_hbox = 23  { unglue a box ( \unhbox, \unhcopy ) }
define un_vbox = 24  { unglue a box ( \unvbox, \unvcopy ) }
        { ( or \pagediscards, \splitdiscards ) }
define remove_item = 25  { nullify last item ( \unpenalty, \unkern, \unskip ) }
define hskip = 26  { horizontal glue ( \hskip, \hfil, etc. ) }
define vskip = 27  { vertical glue ( \vskip, \vfil, etc. ) }
define mskip = 28  { math glue ( \mskip ) }
define kern = 29  { fixed space ( \kern) }
define mkern = 30  { math kern ( \mkern ) }
define leader_ship = 31  { use a box ( \shipout, \leaders, etc. ) }
define halign = 32  { horizontal table alignment ( \halign ) }
define valign = 33  { vertical table alignment ( \valign ) }
        { or text direction directives ( \beginL, etc. ) }
define no_align = 34  { temporary escape from alignment ( \noalign ) }
define vrule = 35  { vertical rule ( \vrule ) }
define hrule = 36  { horizontal rule ( \hrule ) }
define insert = 37  { vlist inserted in box ( \insert ) }
define vadjust = 38  { vlist inserted in enclosing paragraph ( \vadjust ) }
define ignore_spaces = 39  { gobble spacer tokens ( \ignorespaces ) }
define after_assignment = 40  { save till assignment is done ( \afterassignment ) }
define after_group = 41  { save till group is done ( \aftergroup ) }
define break_penalty = 42  { additional badness ( \penalty ) }
define start_par = 43  { begin paragraph ( \indent, \noindent ) }
define ital_corr = 44  { italic correction ( \/ ) }
define accent = 45  { attach accent in text ( \accent ) }
define math_accent = 46  { attach accent in math ( \mathaccent ) }
define discretionary = 47  { discretionary texts ( \-, \discretionary ) }
define eq_no = 48  { equation number ( \eqno, \leqno ) }
define left_right = 49  { variable delimiter ( \left, \right ) }
        { ( or \middle ) }
define math_comp = 50  { component of formula ( \mathbin, etc. ) }
define limit_switch = 51  { diddle limit conventions ( \displaylimits, etc. ) }
define above = 52  { generalized fraction ( \above, \atop, etc. ) }
define math_style = 53  { style specification ( \displaystyle, etc. ) }
define math_choice = 54  { choice specification ( \mathchoice ) }
define non_script = 55  { conditional math glue ( \nonscript ) }
define vcenter = 56  { vertically center a vbox ( \vcenter ) }
define case_shift = 57  { force specific case ( \lowercase, \uppercase ) }
define message = 58  { send to user ( \message, \errmessage ) }
define extension = 59  { extensions to TEX ( \write, \special, etc. ) }
define in_stream = 60  { files for reading ( \openin, \closein ) }
define begin_group = 61  { begin local grouping ( \begingroup ) }
define end_group = 62  { end local grouping ( \endgroup ) }
define omit = 63  { omit alignment template ( \omit ) }
define ex_space = 64  { explicit space ( \␣ ) }
define no_boundary = 65  { suppress boundary ligatures ( \noboundary ) }
define radical = 66  { square root and similar signs ( \radical ) }
define end_cs_name = 67  { end control sequence ( \endcsname ) }
define min_internal = 68  { the smallest code that can follow \the }
define char_given = 68  { character code defined by \chardef }
define math_given = 69  { math code defined by \mathchardef }
define last_item = 70  { most recent item ( \lastpenalty, \lastkern, \lastskip ) }
define max_non_prefixed_command = 70  { largest command code that can't be \global }
```

209* The next codes are special; they all relate to mode-independent assignment of values to TEX's internal registers or tables. Codes that are *max_internal* or less represent internal quantities that might be expanded by '\the'.

```
define toks_register = 71  { token list register ( \toks ) }
define assign_toks = 72  { special token list ( \output, \everypar, etc. ) }
define assign_int = 73  { user-defined integer ( \tolerance, \day, etc. ) }
define assign_dimen = 74  { user-defined length ( \hsize, etc. ) }
define assign_glue = 75  { user-defined glue ( \baselineskip, etc. ) }
define assign_mu_glue = 76  { user-defined muglue ( \thinmuskip, etc. ) }
define assign_font_dimen = 77  { user-defined font dimension ( \fontdimen ) }
define assign_font_int = 78  { user-defined font integer ( \hyphenchar, \skewchar ) }
define set_aux = 79  { specify state info ( \spacefactor, \prevdepth ) }
define set_prev_graf = 80  { specify state info ( \prevgraf ) }
define set_page_dimen = 81  { specify state info ( \pagegoal, etc. ) }
define set_page_int = 82  { specify state info ( \deadcycles, \insertpenalties ) }
          { ( or \interactionmode ) }
define set_box_dimen = 83  { change dimension of box ( \wd, \ht, \dp ) }
define set_shape = 84  { specify fancy paragraph shape ( \parshape ) }
          { (or \interlinepenalties, etc. ) }
define def_code = 85  { define a character code ( \catcode, etc. ) }
define def_family = 86  { declare math fonts ( \textfont, etc. ) }
define set_font = 87  { set current font ( font identifiers ) }
define def_font = 88  { define a font file ( \font ) }
define register = 89  { internal register ( \count, \dimen, etc. ) }
define max_internal = 89  { the largest code that can follow \the }
define advance = 90  { advance a register or parameter ( \advance ) }
define multiply = 91  { multiply a register or parameter ( \multiply ) }
define divide = 92  { divide a register or parameter ( \divide ) }
define prefix = 93  { qualify a definition ( \global, \long, \outer ) }
          { ( or \protected ) }
define let = 94  { assign a command code ( \let, \futurelet ) }
define shorthand_def = 95  { code definition ( \chardef, \countdef, etc. ) }
define read_to_cs = 96  { read into a control sequence ( \read ) }
          { ( or \readline ) }
define def = 97  { macro definition ( \def, \gdef, \xdef, \edef ) }
define set_box = 98  { set a box ( \setbox ) }
define hyph_data = 99  { hyphenation data ( \hyphenation, \patterns ) }
define set_interaction = 100  { define level of interaction ( \batchmode, etc. ) }
define max_command = 100  { the largest command code seen at big_switch }
```

210* The remaining command codes are extra special, since they cannot get through TEX's scanner to the main control routine. They have been given values higher than *max_command* so that their special nature is easily discernible. The "expandable" commands come first.

```
define undefined_cs = max_command + 1   { initial state of most eq_type fields }
define expand_after = max_command + 2   { special expansion ( \expandafter ) }
define no_expand = max_command + 3   { special nonexpansion ( \noexpand ) }
define input = max_command + 4   { input a source file ( \input, \endinput ) }
        { ( or \scantokens ) }
define if_test = max_command + 5   { conditional text ( \if, \ifcase, etc. ) }
define fi_or_else = max_command + 6   { delimiters for conditionals ( \else, etc. ) }
define cs_name = max_command + 7   { make a control sequence from tokens ( \csname ) }
define convert = max_command + 8   { convert to text ( \number, \string, etc. ) }
define the = max_command + 9   { expand an internal quantity ( \the ) }
        { ( or \unexpanded, \detokenize ) }
define top_bot_mark = max_command + 10   { inserted mark ( \topmark, etc. ) }
define call = max_command + 11   { non-long, non-outer control sequence }
define long_call = max_command + 12   { long, non-outer control sequence }
define outer_call = max_command + 13   { non-long, outer control sequence }
define long_outer_call = max_command + 14   { long, outer control sequence }
define end_template = max_command + 15   { end of an alignment template }
define dont_expand = max_command + 16   { the following token was marked by \noexpand }
define glue_ref = max_command + 17   { the equivalent points to a glue specification }
define shape_ref = max_command + 18   { the equivalent points to a parshape specification }
define box_ref = max_command + 19   { the equivalent points to a box node, or is null }
define data = max_command + 20   { the equivalent is simply a halfword number }
```

212* The state of affairs at any semantic level can be represented by five values:

mode is the number representing the semantic mode, as just explained.

head is a *pointer* to a list head for the list being built; *link*(*head*) therefore points to the first element of the list, or to *null* if the list is empty.

tail is a *pointer* to the final node of the list being built; thus, *tail* = *head* if and only if the list is empty.

prev_graf is the number of lines of the current paragraph that have already been put into the present vertical list.

aux is an auxiliary *memory_word* that gives further information that is needed to characterize the situation.

In vertical mode, *aux* is also known as *prev_depth*; it is the scaled value representing the depth of the previous box, for use in baseline calculations, or it is ≤ -1000pt if the next box on the vertical list is to be exempt from baseline calculations. In horizontal mode, *aux* is also known as *space_factor* and *clang*; it holds the current space factor used in spacing calculations, and the current language used for hyphenation. (The value of *clang* is undefined in restricted horizontal mode.) In math mode, *aux* is also known as *incompleat_noad*; if not *null*, it points to a record that represents the numerator of a generalized fraction for which the denominator is currently being formed in the current list.

There is also a sixth quantity, *mode_line*, which correlates the semantic nest with the user's input; *mode_line* contains the source line number at which the current level of nesting was entered. The negative of this line number is the *mode_line* at the level of the user's output routine.

A seventh quantity, *eTeX_aux*, is used by the extended features ε-TEX. In vertical modes it is known as *LR_save* and holds the LR stack when a paragraph is interrupted by a displayed formula. In display math mode it is known as *LR_box* and holds a pointer to a prototype box for the display. In math mode it is known as *delim_ptr* and points to the most recent *left_noad* or *middle_noad* of a *math_left_group*.

In horizontal mode, the *prev_graf* field is used for initial language data.

The semantic nest is an array called *nest* that holds the *mode*, *head*, *tail*, *prev_graf*, *aux*, and *mode_line* values for all semantic levels below the currently active one. Information about the currently active level is kept in the global quantities *mode*, *head*, *tail*, *prev_graf*, *aux*, and *mode_line*, which live in a Pascal record that is ready to be pushed onto *nest* if necessary.

```
define ignore_depth ≡ −65536000  { prev_depth value that is ignored }
⟨ Types in the outer block 18 ⟩ +≡
list_state_record = record mode_field: −mmode .. mmode; head_field, tail_field: pointer;
  eTeX_aux_field: pointer;
  pg_field, ml_field: integer; aux_field: memory_word;
  end;
```

213*
```
define mode ≡ cur_list.mode_field   { current mode }
  define head ≡ cur_list.head_field   { header node of current list }
  define tail ≡ cur_list.tail_field   { final node on current list }
  define eTeX_aux ≡ cur_list.eTeX_aux_field   { auxiliary data for ε-TEX }
  define LR_save ≡ eTeX_aux   { LR stack when a paragraph is interrupted }
  define LR_box ≡ eTeX_aux   { prototype box for display }
  define delim_ptr ≡ eTeX_aux   { most recent left or right noad of a math left group }
  define prev_graf ≡ cur_list.pg_field   { number of paragraph lines accumulated }
  define aux ≡ cur_list.aux_field   { auxiliary data about the current list }
  define prev_depth ≡ aux.sc   { the name of aux in vertical mode }
  define space_factor ≡ aux.hh.lh   { part of aux in horizontal mode }
  define clang ≡ aux.hh.rh   { the other part of aux in horizontal mode }
  define incompleat_noad ≡ aux.int   { the name of aux in math mode }
  define mode_line ≡ cur_list.ml_field   { source file line number at beginning of list }
```

```
⟨ Global variables 13 ⟩ +≡
nest: array [0 .. nest_size] of list_state_record;
nest_ptr: 0 .. nest_size;   { first unused location of nest }
max_nest_stack: 0 .. nest_size;   { maximum of nest_ptr when pushing }
cur_list: list_state_record;   { the “top” semantic state }
shown_mode: −mmode .. mmode;   { most recent mode shown by \tracingcommands }
```

215* We will see later that the vertical list at the bottom semantic level is split into two parts; the “current page” runs from *page_head* to *page_tail*, and the “contribution list” runs from *contrib_head* to *tail* of semantic level zero. The idea is that contributions are first formed in vertical mode, then “contributed” to the current page (during which time the page-breaking decisions are made). For now, we don’t need to know any more details about the page-building process.

```
⟨ Set initial values of key variables 21 ⟩ +≡
  nest_ptr ← 0; max_nest_stack ← 0; mode ← vmode; head ← contrib_head; tail ← contrib_head;
  eTeX_aux ← null; prev_depth ← ignore_depth; mode_line ← 0; prev_graf ← 0; shown_mode ← 0;
  ⟨ Start a new current page 991* ⟩;
```

216* When TEX’s work on one level is interrupted, the state is saved by calling *push_nest*. This routine changes *head* and *tail* so that a new (empty) list is begun; it does not change *mode* or *aux*.

```
procedure push_nest;   { enter a new semantic level, save the old }
  begin if nest_ptr > max_nest_stack then
    begin max_nest_stack ← nest_ptr;
    if nest_ptr = nest_size then overflow("semantic␣nest␣size", nest_size);
    end;
  nest[nest_ptr] ← cur_list;   { stack the record }
  incr(nest_ptr); head ← get_avail; tail ← head; prev_graf ← 0; mode_line ← line; eTeX_aux ← null;
  end;
```

230* Region 4 of *eqtb* contains the local quantities defined here. The bulk of this region is taken up by five tables that are indexed by eight-bit characters; these tables are important to both the syntactic and semantic portions of TEX. There are also a bunch of special things like font and token parameters, as well as the tables of `\toks` and `\box` registers.

```
define par_shape_loc = local_base   { specifies paragraph shape }
define output_routine_loc = local_base + 1   { points to token list for \output }
define every_par_loc = local_base + 2   { points to token list for \everypar }
define every_math_loc = local_base + 3   { points to token list for \everymath }
define every_display_loc = local_base + 4   { points to token list for \everydisplay }
define every_hbox_loc = local_base + 5   { points to token list for \everyhbox }
define every_vbox_loc = local_base + 6   { points to token list for \everyvbox }
define every_job_loc = local_base + 7   { points to token list for \everyjob }
define every_cr_loc = local_base + 8   { points to token list for \everycr }
define err_help_loc = local_base + 9   { points to token list for \errhelp }
define tex_toks = local_base + 10   { end of TEX's token list parameters }
define etex_toks_base = tex_toks   { base for ε-TEX's token list parameters }
define every_eof_loc = etex_toks_base   { points to token list for \everyeof }
define etex_toks = etex_toks_base + 1   { end of ε-TEX's token list parameters }
define toks_base = etex_toks   { table of 256 token list registers }
define etex_pen_base = toks_base + 256   { start of table of ε-TEX's penalties }
define inter_line_penalties_loc = etex_pen_base   { additional penalties between lines }
define club_penalties_loc = etex_pen_base + 1   { penalties for creating club lines }
define widow_penalties_loc = etex_pen_base + 2   { penalties for creating widow lines }
define display_widow_penalties_loc = etex_pen_base + 3   { ditto, just before a display }
define etex_pens = etex_pen_base + 4   { end of table of ε-TEX's penalties }
define box_base = etex_pens   { table of 256 box registers }
define cur_font_loc = box_base + 256   { internal font number outside math mode }
define math_font_base = cur_font_loc + 1   { table of 48 math font numbers }
define cat_code_base = math_font_base + 48   { table of 256 command codes (the "catcodes") }
define lc_code_base = cat_code_base + 256   { table of 256 lowercase mappings }
define uc_code_base = lc_code_base + 256   { table of 256 uppercase mappings }
define sf_code_base = uc_code_base + 256   { table of 256 spacefactor mappings }
define math_code_base = sf_code_base + 256   { table of 256 math mode mappings }
define int_base = math_code_base + 256   { beginning of region 5 }
define par_shape_ptr ≡ equiv(par_shape_loc)
define output_routine ≡ equiv(output_routine_loc)
define every_par ≡ equiv(every_par_loc)
define every_math ≡ equiv(every_math_loc)
define every_display ≡ equiv(every_display_loc)
define every_hbox ≡ equiv(every_hbox_loc)
define every_vbox ≡ equiv(every_vbox_loc)
define every_job ≡ equiv(every_job_loc)
define every_cr ≡ equiv(every_cr_loc)
define err_help ≡ equiv(err_help_loc)
define toks(#) ≡ equiv(toks_base + #)
define box(#) ≡ equiv(box_base + #)
define cur_font ≡ equiv(cur_font_loc)
define fam_fnt(#) ≡ equiv(math_font_base + #)
define cat_code(#) ≡ equiv(cat_code_base + #)
define lc_code(#) ≡ equiv(lc_code_base + #)
define uc_code(#) ≡ equiv(uc_code_base + #)
```

```
  define sf_code(#) ≡ equiv(sf_code_base + #)
  define math_code(#) ≡ equiv(math_code_base + #)
              { Note: math_code(c) is the true math code plus min_halfword }
⟨ Put each of TeX's primitives into the hash table 226 ⟩ +≡
  primitive("output", assign_toks, output_routine_loc); primitive("everypar", assign_toks, every_par_loc);
  primitive("everymath", assign_toks, every_math_loc);
  primitive("everydisplay", assign_toks, every_display_loc);
  primitive("everyhbox", assign_toks, every_hbox_loc); primitive("everyvbox", assign_toks, every_vbox_loc);
  primitive("everyjob", assign_toks, every_job_loc); primitive("everycr", assign_toks, every_cr_loc);
  primitive("errhelp", assign_toks, err_help_loc);
```

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

```
assign_toks: if chr_code ≥ toks_base then
    begin print_esc("toks"); print_int(chr_code − toks_base);
    end
  else case chr_code of
    output_routine_loc: print_esc("output");
    every_par_loc: print_esc("everypar");
    every_math_loc: print_esc("everymath");
    every_display_loc: print_esc("everydisplay");
    every_hbox_loc: print_esc("everyhbox");
    every_vbox_loc: print_esc("everyvbox");
    every_job_loc: print_esc("everyjob");
    every_cr_loc: print_esc("everycr");
      ⟨ Cases of assign_toks for print_cmd_chr 1389* ⟩
    othercases print_esc("errhelp")
    endcases;
```

232* We initialize most things to null or undefined values. An undefined font is represented by the internal code *font_base*.

However, the character code tables are given initial values based on the conventional interpretation of ASCII code. These initial values should not be changed when TEX is adapted for use with non-English languages; all changes to the initialization conventions should be made in format packages, not in TEX itself, so that global interchange of formats is possible.

```
  define null_font ≡ font_base
  define var_code ≡ ´70000   { math code meaning "use the current family" }
⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
  par_shape_ptr ← null; eq_type(par_shape_loc) ← shape_ref; eq_level(par_shape_loc) ← level_one;
  for k ← etex_pen_base to etex_pens − 1 do eqtb[k] ← eqtb[par_shape_loc];
  for k ← output_routine_loc to toks_base + 255 do eqtb[k] ← eqtb[undefined_control_sequence];
  box(0) ← null; eq_type(box_base) ← box_ref; eq_level(box_base) ← level_one;
  for k ← box_base + 1 to box_base + 255 do eqtb[k] ← eqtb[box_base];
  cur_font ← null_font; eq_type(cur_font_loc) ← data; eq_level(cur_font_loc) ← level_one;
  for k ← math_font_base to math_font_base + 47 do eqtb[k] ← eqtb[cur_font_loc];
  equiv(cat_code_base) ← 0; eq_type(cat_code_base) ← data; eq_level(cat_code_base) ← level_one;
  for k ← cat_code_base + 1 to int_base − 1 do eqtb[k] ← eqtb[cat_code_base];
  for k ← 0 to 255 do
    begin cat_code(k) ← other_char; math_code(k) ← hi(k); sf_code(k) ← 1000;
    end;
  cat_code(carriage_return) ← car_ret; cat_code("␣") ← spacer; cat_code("\") ← escape;
  cat_code("%") ← comment; cat_code(invalid_code) ← invalid_char; cat_code(null_code) ← ignore;
  for k ← "0" to "9" do math_code(k) ← hi(k + var_code);
  for k ← "A" to "Z" do
    begin cat_code(k) ← letter; cat_code(k + "a" − "A") ← letter;
    math_code(k) ← hi(k + var_code + ˝100);
    math_code(k + "a" − "A") ← hi(k + "a" − "A" + var_code + ˝100);
    lc_code(k) ← k + "a" − "A"; lc_code(k + "a" − "A") ← k + "a" − "A";
    uc_code(k) ← k; uc_code(k + "a" − "A") ← k;
    sf_code(k) ← 999;
    end;
```

233* ⟨ Show equivalent n, in region 4 233* ⟩ ≡

```
if $(n = par\_shape\_loc) \lor ((n \ge etex\_pen\_base) \land (n < etex\_pens))$ then
  begin $print\_cmd\_chr(set\_shape, n)$; $print\_char("=")$;
  if $equiv(n) = null$ then $print\_char("0")$
  else if $n > par\_shape\_loc$ then
      begin $print\_int(penalty(equiv(n)))$; $print\_char("␣")$; $print\_int(penalty(equiv(n) + 1))$;
      if $penalty(equiv(n)) > 1$ then $print\_esc("ETC.")$;
      end
    else $print\_int(info(par\_shape\_ptr))$;
  end
else if $n < toks\_base$ then
    begin $print\_cmd\_chr(assign\_toks, n)$; $print\_char("=")$;
    if $equiv(n) \ne null$ then $show\_token\_list(link(equiv(n)), null, 32)$;
    end
  else if $n < box\_base$ then
      begin $print\_esc("toks")$; $print\_int(n - toks\_base)$; $print\_char("=")$;
      if $equiv(n) \ne null$ then $show\_token\_list(link(equiv(n)), null, 32)$;
      end
    else if $n < cur\_font\_loc$ then
        begin $print\_esc("box")$; $print\_int(n - box\_base)$; $print\_char("=")$;
        if $equiv(n) = null$ then $print("void")$
        else begin $depth\_threshold \gets 0$; $breadth\_max \gets 1$; $show\_node\_list(equiv(n))$;
          end;
        end
      else if $n < cat\_code\_base$ then ⟨ Show the font identifier in $eqtb[n]$ 234 ⟩
        else ⟨ Show the halfword code in $eqtb[n]$ 235 ⟩
```

This code is used in section 252.

236* Region 5 of *eqtb* contains the integer parameters and registers defined here, as well as the *del_code* table. The latter table differs from the *cat_code* .. *math_code* tables that precede it, since delimiter codes are fullword integers while the other kinds of codes occupy at most a halfword. This is what makes region 5 different from region 4. We will store the *eq_level* information in an auxiliary array of quarterwords that will be defined later.

```
define pretolerance_code = 0 { badness tolerance before hyphenation }
define tolerance_code = 1 { badness tolerance after hyphenation }
define line_penalty_code = 2 { added to the badness of every line }
define hyphen_penalty_code = 3 { penalty for break after discretionary hyphen }
define ex_hyphen_penalty_code = 4 { penalty for break after explicit hyphen }
define club_penalty_code = 5 { penalty for creating a club line }
define widow_penalty_code = 6 { penalty for creating a widow line }
define display_widow_penalty_code = 7 { ditto, just before a display }
define broken_penalty_code = 8 { penalty for breaking a page at a broken line }
define bin_op_penalty_code = 9 { penalty for breaking after a binary operation }
define rel_penalty_code = 10 { penalty for breaking after a relation }
define pre_display_penalty_code = 11 { penalty for breaking just before a displayed formula }
define post_display_penalty_code = 12 { penalty for breaking just after a displayed formula }
define inter_line_penalty_code = 13 { additional penalty between lines }
define double_hyphen_demerits_code = 14 { demerits for double hyphen break }
define final_hyphen_demerits_code = 15 { demerits for final hyphen break }
define adj_demerits_code = 16 { demerits for adjacent incompatible lines }
define mag_code = 17 { magnification ratio }
define delimiter_factor_code = 18 { ratio for variable-size delimiters }
define looseness_code = 19 { change in number of lines for a paragraph }
define time_code = 20 { current time of day }
define day_code = 21 { current day of the month }
define month_code = 22 { current month of the year }
define year_code = 23 { current year of our Lord }
define show_box_breadth_code = 24 { nodes per level in show_box }
define show_box_depth_code = 25 { maximum level in show_box }
define hbadness_code = 26 { hboxes exceeding this badness will be shown by hpack }
define vbadness_code = 27 { vboxes exceeding this badness will be shown by vpack }
define pausing_code = 28 { pause after each line is read from a file }
define tracing_online_code = 29 { show diagnostic output on terminal }
define tracing_macros_code = 30 { show macros as they are being expanded }
define tracing_stats_code = 31 { show memory usage if TEX knows it }
define tracing_paragraphs_code = 32 { show line-break calculations }
define tracing_pages_code = 33 { show page-break calculations }
define tracing_output_code = 34 { show boxes when they are shipped out }
define tracing_lost_chars_code = 35 { show characters that aren't in the font }
define tracing_commands_code = 36 { show command codes at big_switch }
define tracing_restores_code = 37 { show equivalents when they are restored }
define uc_hyph_code = 38 { hyphenate words beginning with a capital letter }
define output_penalty_code = 39 { penalty found at current page break }
define max_dead_cycles_code = 40 { bound on consecutive dead cycles of output }
define hang_after_code = 41 { hanging indentation changes after this many lines }
define floating_penalty_code = 42 { penalty for insertions heldover after a split }
define global_defs_code = 43 { override \global specifications }
define cur_fam_code = 44 { current family }
define escape_char_code = 45 { escape character for token output }
define default_hyphen_char_code = 46 { value of \hyphenchar when a font is loaded }
define default_skew_char_code = 47 { value of \skewchar when a font is loaded }
define end_line_char_code = 48 { character placed at the right end of the buffer }
define new_line_char_code = 49 { character that prints as print_ln }
define language_code = 50 { current hyphenation table }
define left_hyphen_min_code = 51 { minimum left hyphenation fragment size }
define right_hyphen_min_code = 52 { minimum right hyphenation fragment size }
define holding_inserts_code = 53 { do not remove insertion nodes from \box255 }
define error_context_lines_code = 54 { maximum intermediate line pairs shown }
define tex_int_pars = 55 { total number of TeX's integer parameters }
define etex_int_base = tex_int_pars { base for ε-TeX's integer parameters }
define tracing_assigns_code = etex_int_base { show assignments }
define tracing_groups_code = etex_int_base + 1 { show save/restore groups }
define tracing_ifs_code = etex_int_base + 2 { show conditionals }
define tracing_scan_tokens_code = etex_int_base + 3 { show pseudo file open and close }
define tracing_nesting_code = etex_int_base + 4 { show incomplete groups and ifs within files }
define pre_display_direction_code = etex_int_base + 5 { text direction preceding a display }
define last_line_fit_code = etex_int_base + 6 { adjustment for last line of paragraph }
define saving_vdiscards_code = etex_int_base + 7 { save items discarded from vlists }
define saving_hyph_codes_code = etex_int_base + 8 { save hyphenation codes for languages }
define eTeX_state_code = etex_int_base + 9 { ε-TeX state variables }
define etex_int_pars = eTeX_state_code + eTeX_states { total number of ε-TeX's integer parameters }
define int_pars = etex_int_pars { total number of integer parameters }
define count_base = int_base + int_pars { 256 user \count registers }
define del_code_base = count_base + 256 { 256 delimiter code mappings }
define dimen_base = del_code_base + 256 { beginning of region 6 }
define del_code(#) ≡ eqtb[del_code_base + #].int
define count(#) ≡ eqtb[count_base + #].int
define int_par(#) ≡ eqtb[int_base + #].int { an integer parameter }
define pretolerance ≡ int_par(pretolerance_code)
define tolerance ≡ int_par(tolerance_code)
define line_penalty ≡ int_par(line_penalty_code)
define hyphen_penalty ≡ int_par(hyphen_penalty_code)
define ex_hyphen_penalty ≡ int_par(ex_hyphen_penalty_code)
define club_penalty ≡ int_par(club_penalty_code)
define widow_penalty ≡ int_par(widow_penalty_code)
define display_widow_penalty ≡ int_par(display_widow_penalty_code)
define broken_penalty ≡ int_par(broken_penalty_code)
define bin_op_penalty ≡ int_par(bin_op_penalty_code)
define rel_penalty ≡ int_par(rel_penalty_code)
define pre_display_penalty ≡ int_par(pre_display_penalty_code)
define post_display_penalty ≡ int_par(post_display_penalty_code)
define inter_line_penalty ≡ int_par(inter_line_penalty_code)
define double_hyphen_demerits ≡ int_par(double_hyphen_demerits_code)
define final_hyphen_demerits ≡ int_par(final_hyphen_demerits_code)
define adj_demerits ≡ int_par(adj_demerits_code)
define mag ≡ int_par(mag_code)
define delimiter_factor ≡ int_par(delimiter_factor_code)
define looseness ≡ int_par(looseness_code)
define time ≡ int_par(time_code)
define day ≡ int_par(day_code)
define month ≡ int_par(month_code)
define year ≡ int_par(year_code)
```

```
  define show_box_breadth ≡ int_par(show_box_breadth_code)
  define show_box_depth ≡ int_par(show_box_depth_code)
  define hbadness ≡ int_par(hbadness_code)
  define vbadness ≡ int_par(vbadness_code)
  define pausing ≡ int_par(pausing_code)
  define tracing_online ≡ int_par(tracing_online_code)
  define tracing_macros ≡ int_par(tracing_macros_code)
  define tracing_stats ≡ int_par(tracing_stats_code)
  define tracing_paragraphs ≡ int_par(tracing_paragraphs_code)
  define tracing_pages ≡ int_par(tracing_pages_code)
  define tracing_output ≡ int_par(tracing_output_code)
  define tracing_lost_chars ≡ int_par(tracing_lost_chars_code)
  define tracing_commands ≡ int_par(tracing_commands_code)
  define tracing_restores ≡ int_par(tracing_restores_code)
  define uc_hyph ≡ int_par(uc_hyph_code)
  define output_penalty ≡ int_par(output_penalty_code)
  define max_dead_cycles ≡ int_par(max_dead_cycles_code)
  define hang_after ≡ int_par(hang_after_code)
  define floating_penalty ≡ int_par(floating_penalty_code)
  define global_defs ≡ int_par(global_defs_code)
  define cur_fam ≡ int_par(cur_fam_code)
  define escape_char ≡ int_par(escape_char_code)
  define default_hyphen_char ≡ int_par(default_hyphen_char_code)
  define default_skew_char ≡ int_par(default_skew_char_code)
  define end_line_char ≡ int_par(end_line_char_code)
  define new_line_char ≡ int_par(new_line_char_code)
  define language ≡ int_par(language_code)
  define left_hyphen_min ≡ int_par(left_hyphen_min_code)
  define right_hyphen_min ≡ int_par(right_hyphen_min_code)
  define holding_inserts ≡ int_par(holding_inserts_code)
  define error_context_lines ≡ int_par(error_context_lines_code)

  define tracing_assigns ≡ int_par(tracing_assigns_code)
  define tracing_groups ≡ int_par(tracing_groups_code)
  define tracing_ifs ≡ int_par(tracing_ifs_code)
  define tracing_scan_tokens ≡ int_par(tracing_scan_tokens_code)
  define tracing_nesting ≡ int_par(tracing_nesting_code)
  define pre_display_direction ≡ int_par(pre_display_direction_code)
  define last_line_fit ≡ int_par(last_line_fit_code)
  define saving_vdiscards ≡ int_par(saving_vdiscards_code)
  define saving_hyph_codes ≡ int_par(saving_hyph_codes_code)

⟨ Assign the values depth_threshold ← show_box_depth and breadth_max ← show_box_breadth 236* ⟩ ≡
  depth_threshold ← show_box_depth; breadth_max ← show_box_breadth
```

This code is used in section 198.

237.* We can print the symbolic name of an integer parameter as follows.

```
procedure print_param(n : integer);
  begin case n of
  pretolerance_code: print_esc("pretolerance");
  tolerance_code: print_esc("tolerance");
  line_penalty_code: print_esc("linepenalty");
  hyphen_penalty_code: print_esc("hyphenpenalty");
  ex_hyphen_penalty_code: print_esc("exhyphenpenalty");
  club_penalty_code: print_esc("clubpenalty");
  widow_penalty_code: print_esc("widowpenalty");
  display_widow_penalty_code: print_esc("displaywidowpenalty");
  broken_penalty_code: print_esc("brokenpenalty");
  bin_op_penalty_code: print_esc("binoppenalty");
  rel_penalty_code: print_esc("relpenalty");
  pre_display_penalty_code: print_esc("predisplaypenalty");
  post_display_penalty_code: print_esc("postdisplaypenalty");
  inter_line_penalty_code: print_esc("interlinepenalty");
  double_hyphen_demerits_code: print_esc("doublehyphendemerits");
  final_hyphen_demerits_code: print_esc("finalhyphendemerits");
  adj_demerits_code: print_esc("adjdemerits");
  mag_code: print_esc("mag");
  delimiter_factor_code: print_esc("delimiterfactor");
  looseness_code: print_esc("looseness");
  time_code: print_esc("time");
  day_code: print_esc("day");
  month_code: print_esc("month");
  year_code: print_esc("year");
  show_box_breadth_code: print_esc("showboxbreadth");
  show_box_depth_code: print_esc("showboxdepth");
  hbadness_code: print_esc("hbadness");
  vbadness_code: print_esc("vbadness");
  pausing_code: print_esc("pausing");
  tracing_online_code: print_esc("tracingonline");
  tracing_macros_code: print_esc("tracingmacros");
  tracing_stats_code: print_esc("tracingstats");
  tracing_paragraphs_code: print_esc("tracingparagraphs");
  tracing_pages_code: print_esc("tracingpages");
  tracing_output_code: print_esc("tracingoutput");
  tracing_lost_chars_code: print_esc("tracinglostchars");
  tracing_commands_code: print_esc("tracingcommands");
  tracing_restores_code: print_esc("tracingrestores");
  uc_hyph_code: print_esc("uchyph");
  output_penalty_code: print_esc("outputpenalty");
  max_dead_cycles_code: print_esc("maxdeadcycles");
  hang_after_code: print_esc("hangafter");
  floating_penalty_code: print_esc("floatingpenalty");
  global_defs_code: print_esc("globaldefs");
  cur_fam_code: print_esc("fam");
  escape_char_code: print_esc("escapechar");
  default_hyphen_char_code: print_esc("defaulthyphenchar");
  default_skew_char_code: print_esc("defaultskewchar");
  end_line_char_code: print_esc("endlinechar");
```

```
new_line_char_code: print_esc("newlinechar");
language_code: print_esc("language");
left_hyphen_min_code: print_esc("lefthyphenmin");
right_hyphen_min_code: print_esc("righthyphenmin");
holding_inserts_code: print_esc("holdinginserts");
error_context_lines_code: print_esc("errorcontextlines");
  ⟨ Cases for print_param 1390* ⟩
othercases print("[unknown␣integer␣parameter!]")
endcases;
end;
```

264* We need to put TEX's "primitive" control sequences into the hash table, together with their command code (which will be the *eq_type*) and an operand (which will be the *equiv*). The *primitive* procedure does this, in a way that no TEX user can. The global value *cur_val* contains the new *eqtb* pointer after *primitive* has acted.

```
init procedure primitive(s : str_number; c : quarterword; o : halfword);
var k: pool_pointer; { index into str_pool }
  j: 0 . . buf_size; { index into buffer }
  l: small_number; { length of the string }
begin if $s < 256$ then cur_val ← s + single_base
else begin k ← str_start[s]; l ← str_start[s + 1] − k;
      { we will move s into the (possibly non-empty) buffer }
  if $first + l > buf\_size + 1$ then overflow("buffer␣size", buf_size);
  for j ← 0 to l − 1 do buffer[first + j] ← so(str_pool[k + j]);
  cur_val ← id_lookup(first, l); { no_new_control_sequence is false }
  flush_string; text(cur_val) ← s; { we don't want to have the string twice }
  end;
eq_level(cur_val) ← level_one; eq_type(cur_val) ← c; equiv(cur_val) ← o;
end;
tini
```

265* Many of TEX's primitives need no *equiv*, since they are identifiable by their *eq_type* alone. These primitives are loaded into the hash table as follows:

```
⟨Put each of TEX's primitives into the hash table 226⟩ +≡
  primitive("␣", ex_space, 0);
  primitive("/", ital_corr, 0);
  primitive("accent", accent, 0);
  primitive("advance", advance, 0);
  primitive("afterassignment", after_assignment, 0);
  primitive("aftergroup", after_group, 0);
  primitive("begingroup", begin_group, 0);
  primitive("char", char_num, 0);
  primitive("csname", cs_name, 0);
  primitive("delimiter", delim_num, 0);
  primitive("divide", divide, 0);
  primitive("endcsname", end_cs_name, 0);
  primitive("endgroup", end_group, 0); text(frozen_end_group) ← "endgroup";
  eqtb[frozen_end_group] ← eqtb[cur_val];
  primitive("expandafter", expand_after, 0);
  primitive("font", def_font, 0);
  primitive("fontdimen", assign_font_dimen, 0);
  primitive("halign", halign, 0);
  primitive("hrule", hrule, 0);
  primitive("ignorespaces", ignore_spaces, 0);
  primitive("insert", insert, 0);
  primitive("mark", mark, 0);
  primitive("mathaccent", math_accent, 0);
  primitive("mathchar", math_char_num, 0);
  primitive("mathchoice", math_choice, 0);
  primitive("multiply", multiply, 0);
  primitive("noalign", no_align, 0);
  primitive("noboundary", no_boundary, 0);
  primitive("noexpand", no_expand, 0);
  primitive("nonscript", non_script, 0);
  primitive("omit", omit, 0);
  primitive("parshape", set_shape, par_shape_loc);
  primitive("penalty", break_penalty, 0);
  primitive("prevgraf", set_prev_graf, 0);
  primitive("radical", radical, 0);
  primitive("read", read_to_cs, 0);
  primitive("relax", relax, 256);   { cf. scan_file_name }
  text(frozen_relax) ← "relax"; eqtb[frozen_relax] ← eqtb[cur_val];
  primitive("setbox", set_box, 0);
  primitive("the", the, 0);
  primitive("toks", toks_register, mem_bot);
  primitive("vadjust", vadjust, 0);
  primitive("valign", valign, 0);
  primitive("vcenter", vcenter, 0);
  primitive("vrule", vrule, 0);
```

266* Each primitive has a corresponding inverse, so that it is possible to display the cryptic numeric contents of *eqtb* in symbolic form. Every call of *primitive* in this program is therefore accompanied by some straightforward code that forms part of the *print_cmd_chr* routine below.

```
⟨ Cases of print_cmd_chr for symbolic printing of primitives 227 ⟩ +≡
accent: print_esc("accent");
advance: print_esc("advance");
after_assignment: print_esc("afterassignment");
after_group: print_esc("aftergroup");
assign_font_dimen: print_esc("fontdimen");
begin_group: print_esc("begingroup");
break_penalty: print_esc("penalty");
char_num: print_esc("char");
cs_name: print_esc("csname");
def_font: print_esc("font");
delim_num: print_esc("delimiter");
divide: print_esc("divide");
end_cs_name: print_esc("endcsname");
end_group: print_esc("endgroup");
ex_space: print_esc("␣");
expand_after: if chr_code = 0 then print_esc("expandafter")
      ⟨ Cases of expandafter for print_cmd_chr 1498* ⟩;
halign: print_esc("halign");
hrule: print_esc("hrule");
ignore_spaces: print_esc("ignorespaces");
insert: print_esc("insert");
ital_corr: print_esc("/");
mark: begin print_esc("mark");
  if chr_code > 0 then print_char("s");
  end;
math_accent: print_esc("mathaccent");
math_char_num: print_esc("mathchar");
math_choice: print_esc("mathchoice");
multiply: print_esc("multiply");
no_align: print_esc("noalign");
no_boundary: print_esc("noboundary");
no_expand: print_esc("noexpand");
non_script: print_esc("nonscript");
omit: print_esc("omit");
radical: print_esc("radical");
read_to_cs: if chr_code = 0 then print_esc("read") ⟨ Cases of read for print_cmd_chr 1495* ⟩;
relax: print_esc("relax");
set_box: print_esc("setbox");
set_prev_graf: print_esc("prevgraf");
set_shape: case chr_code of
  par_shape_loc: print_esc("parshape");
    ⟨ Cases of set_shape for print_cmd_chr 1600* ⟩
  end; { there are no other cases }
the: if chr_code = 0 then print_esc("the") ⟨ Cases of the for print_cmd_chr 1418* ⟩;
toks_register: ⟨ Cases of toks_register for print_cmd_chr 1568* ⟩;
vadjust: print_esc("vadjust");
valign: if chr_code = 0 then print_esc("valign")
  ⟨ Cases of valign for print_cmd_chr 1433* ⟩;
```

vcenter: *print_esc*("`vcenter`");
vrule: *print_esc*("`vrule`");

268.* Saving and restoring equivalents. The nested structure provided by '{...}' groups in TEX means that *eqtb* entries valid in outer groups should be saved and restored later if they are overridden inside the braces. When a new *eqtb* value is being assigned, the program therefore checks to see if the previous entry belongs to an outer level. In such a case, the old value is placed on the *save_stack* just before the new value enters *eqtb*. At the end of a grouping level, i.e., when the right brace is sensed, the *save_stack* is used to restore the outer values, and the inner ones are destroyed.

Entries on the *save_stack* are of type *memory_word*. The top item on this stack is *save_stack*[*p*], where $p = save\_ptr - 1$; it contains three fields called *save_type*, *save_level*, and *save_index*, and it is interpreted in one of five ways:

1) If *save_type*(*p*) = *restore_old_value*, then *save_index*(*p*) is a location in *eqtb* whose current value should be destroyed at the end of the current group and replaced by *save_stack*[$p - 1$]. Furthermore if *save_index*(*p*) ≥ *int_base*, then *save_level*(*p*) should replace the corresponding entry in *xeq_level*.

2) If *save_type*(*p*) = *restore_zero*, then *save_index*(*p*) is a location in *eqtb* whose current value should be destroyed at the end of the current group, when it should be replaced by the current value of *eqtb*[*undefined_control_sequence*].

3) If *save_type*(*p*) = *insert_token*, then *save_index*(*p*) is a token that should be inserted into TEX's input when the current group ends.

4) If *save_type*(*p*) = *level_boundary*, then *save_level*(*p*) is a code explaining what kind of group we were previously in, and *save_index*(*p*) points to the level boundary word at the bottom of the entries for that group. Furthermore, in extended ε-TEX mode, *save_stack*[$p - 1$] contains the source line number at which the current level of grouping was entered.

5) If *save_type*(*p*) = *restore_sa*, then *sa_chain* points to a chain of sparse array entries to be restored at the end of the current group. Furthermore *save_index*(*p*) and *save_level*(*p*) should replace the values of *sa_chain* and *sa_level* respectively.

```
define save_type(#) ≡ save_stack[#].hh.b0   { classifies a save_stack entry }
define save_level(#) ≡ save_stack[#].hh.b1   { saved level for regions 5 and 6, or group code }
define save_index(#) ≡ save_stack[#].hh.rh   { eqtb location or token or save_stack location }
define restore_old_value = 0   { save_type when a value should be restored later }
define restore_zero = 1   { save_type when an undefined entry should be restored }
define insert_token = 2   { save_type when a token is being saved for later use }
define level_boundary = 3   { save_type corresponding to beginning of group }
define restore_sa = 4   { save_type when sparse array entries should be restored }
```

⟨ Declare ε-TEX procedures for tracing and input 284* ⟩

273.* The following macro is used to test if there is room for up to seven more entries on *save_stack*. By making a conservative test like this, we can get by with testing for overflow in only a few places.

```
define check_full_save_stack ≡
        if save_ptr > max_save_stack then
          begin max_save_stack ← save_ptr;
          if max_save_stack > save_size − 7 then overflow("save␣size", save_size);
          end
```

274* Procedure *new_save_level* is called when a group begins. The argument is a group identification code like '*hbox_group*'. After calling this routine, it is safe to put five more entries on *save_stack*.

In some cases integer-valued items are placed onto the *save_stack* just below a *level_boundary* word, because this is a convenient place to keep information that is supposed to "pop up" just when the group has finished. For example, when '`\hbox to 100pt{...}`' is being treated, the 100pt dimension is stored on *save_stack* just before *new_save_level* is called.

We use the notation *saved*(k) to stand for an integer item that appears in location *save_ptr* $+ k$ of the save stack.

```
  define saved(#) ≡ save_stack[save_ptr + #].int
procedure new_save_level(c : group_code);   { begin a new level of grouping }
  begin check_full_save_stack;
  if eTeX_ex then
    begin saved(0) ← line; incr(save_ptr);
    end;
  save_type(save_ptr) ← level_boundary; save_level(save_ptr) ← cur_group;
  save_index(save_ptr) ← cur_boundary;
  if cur_level = max_quarterword then
    overflow("grouping␣levels", max_quarterword − min_quarterword);
        { quit if (cur_level + 1) is too big to be stored in eqtb }
  cur_boundary ← save_ptr; cur_group ← c;
  stat if tracing_groups > 0 then group_trace(false);
  tats
  incr(cur_level); incr(save_ptr);
  end;
```

275* Just before an entry of *eqtb* is changed, the following procedure should be called to update the other data structures properly. It is important to keep in mind that reference counts in *mem* include references from within *save_stack*, so these counts must be handled carefully.

```
procedure eq_destroy(w : memory_word);   { gets ready to forget w }
  var q: pointer;   { equiv field of w }
  begin case eq_type_field(w) of
  call, long_call, outer_call, long_outer_call: delete_token_ref(equiv_field(w));
  glue_ref: delete_glue_ref(equiv_field(w));
  shape_ref: begin q ← equiv_field(w);   { we need to free a \parshape block }
    if q ≠ null then free_node(q, info(q) + info(q) + 1);
    end;   { such a block is 2n + 1 words long, where n = info(q) }
  box_ref: flush_node_list(equiv_field(w));
    ⟨ Cases for eq_destroy 1569* ⟩
  othercases do_nothing
  endcases;
  end;
```

277*. The procedure *eq_define* defines an *eqtb* entry having specified *eq_type* and *equiv* fields, and saves the former value if appropriate. This procedure is used only for entries in the first four regions of *eqtb*, i.e., only for entries that have *eq_type* and *equiv* fields. After calling this routine, it is safe to put four more entries on *save_stack*, provided that there was room for four more entries before the call, since *eq_save* makes the necessary test.

```
  define assign_trace(#) ≡
          stat if tracing_assigns $> 0$ then restore_trace(#);
          tats
procedure eq_define(p : pointer; t : quarterword; e : halfword);   { new data for eqtb }
  label exit;
  begin if eTeX_ex $\land$ (eq_type(p) $= t$) $\land$ (equiv(p) $= e$) then
    begin assign_trace(p, "reassigning")
    eq_destroy(eqtb[p]); return;
    end;
  assign_trace(p, "changing")
  if eq_level(p) = cur_level then eq_destroy(eqtb[p])
  else if cur_level $>$ level_one then eq_save(p, eq_level(p));
  eq_level(p) $\leftarrow$ cur_level; eq_type(p) $\leftarrow t$; equiv(p) $\leftarrow e$; assign_trace(p, "into")
exit: end;
```

278*. The counterpart of *eq_define* for the remaining (fullword) positions in *eqtb* is called *eq_word_define*. Since *xeq_level*[*p*] $\geq$ *level_one* for all *p*, a '*restore_zero*' will never be used in this case.

```
procedure eq_word_define(p : pointer; w : integer);
  label exit;
  begin if eTeX_ex $\land$ (eqtb[p].int $= w$) then
    begin assign_trace(p, "reassigning")
    return;
    end;
  assign_trace(p, "changing")
  if xeq_level[p] $\neq$ cur_level then
    begin eq_save(p, xeq_level[p]); xeq_level[p] $\leftarrow$ cur_level;
    end;
  eqtb[p].int $\leftarrow w$; assign_trace(p, "into")
exit: end;
```

279*. The *eq_define* and *eq_word_define* routines take care of local definitions. Global definitions are done in almost the same way, but there is no need to save old values, and the new value is associated with *level_one*.

```
procedure geq_define(p : pointer; t : quarterword; e : halfword);   { global eq_define }
  begin assign_trace(p, "globally␣changing")
  begin eq_destroy(eqtb[p]); eq_level(p) $\leftarrow$ level_one; eq_type(p) $\leftarrow t$; equiv(p) $\leftarrow e$;
  end; assign_trace(p, "into")
  end;

procedure geq_word_define(p : pointer; w : integer);   { global eq_word_define }
  begin assign_trace(p, "globally␣changing")
  begin eqtb[p].int $\leftarrow w$; xeq_level[p] $\leftarrow$ level_one;
  end; assign_trace(p, "into")
  end;
```

281*. The *unsave* routine goes the other way, taking items off of *save_stack*. This routine takes care of restoration when a level ends; everything belonging to the topmost group is cleared off of the save stack.

```
procedure back_input; forward;
procedure unsave; { pops the top level off the save stack }
  label done;
  var p: pointer; { position to be restored }
    l: quarterword; { saved level, if in fullword regions of eqtb }
    t: halfword; { saved value of cur_tok }
    a: boolean; { have we already processed an \aftergroup ? }
  begin a ← false;
  if cur_level > level_one then
    begin decr(cur_level); ⟨Clear off top level from save_stack 282*⟩;
    end
  else confusion("curlevel"); { unsave is not used when cur_group = bottom_level }
  end;
```

282*. ⟨Clear off top level from *save_stack* 282*⟩ ≡

```
  loop begin decr(save_ptr);
    if save_type(save_ptr) = level_boundary then goto done;
    p ← save_index(save_ptr);
    if save_type(save_ptr) = insert_token then ⟨Insert token p into TEX's input 326*⟩
    else if save_type(save_ptr) = restore_sa then
        begin sa_restore; sa_chain ← p; sa_level ← save_level(save_ptr);
        end
      else begin if save_type(save_ptr) = restore_old_value then
          begin l ← save_level(save_ptr); decr(save_ptr);
          end
        else save_stack[save_ptr] ← eqtb[undefined_control_sequence];
        ⟨Store save_stack[save_ptr] in eqtb[p], unless eqtb[p] holds a global value 283⟩;
        end;
    end;
done: stat if tracing_groups > 0 then group_trace(true);
  tats
  if grp_stack[in_open] = cur_boundary then group_warning;
        { groups possibly not properly nested with files }
  cur_group ← save_level(save_ptr); cur_boundary ← save_index(save_ptr);
  if eTeX_ex then decr(save_ptr)
```

This code is used in section 281*.

284*. ⟨Declare ε-TEX procedures for tracing and input 284*⟩ ≡

```
  stat procedure restore_trace(p : pointer; s : str_number); { eqtb[p] has just been restored or retained }
  begin begin_diagnostic; print_char("{"); print(s); print_char("␣"); show_eqtb(p); print_char("}");
  end_diagnostic(false);
  end;
  tats
```

See also sections 1392*, 1393*, 1491*, 1492*, 1509*, 1511*, 1512*, 1556*, 1558*, 1572*, 1573*, 1574*, 1575*, and 1576*.

This code is used in section 268*.

289.* Token lists. A TEX token is either a character or a control sequence, and it is represented internally in one of two ways: (1) A character whose ASCII code number is c and whose command code is m is represented as the number $2^8m + c$; the command code is in the range $1 \le m \le 14$. (2) A control sequence whose *eqtb* address is p is represented as the number $cs\_token\_flag + p$. Here $cs\_token\_flag = 2^{12} - 1$ is larger than $2^8m + c$, yet it is small enough that $cs\_token\_flag + p < max\_halfword$; thus, a token fits comfortably in a halfword.

A token t represents a *left_brace* command if and only if $t < left\_brace\_limit$; it represents a *right_brace* command if and only if we have $left\_brace\_limit \le t < right\_brace\_limit$; and it represents a *match* or *end_match* command if and only if $match\_token \le t \le end\_match\_token$. The following definitions take care of these token-oriented constants and a few others.

```
define cs_token_flag ≡ ´7777  { amount added to the eqtb location in a token that stands for a control
        sequence; is a multiple of 256, less 1 }
define left_brace_token = ´0400   { $2^8 \cdot left\_brace$ }
define left_brace_limit = ´1000   { $2^8 \cdot (left\_brace + 1)$ }
define right_brace_token = ´1000   { $2^8 \cdot right\_brace$ }
define right_brace_limit = ´1400   { $2^8 \cdot (right\_brace + 1)$ }
define math_shift_token = ´1400   { $2^8 \cdot math\_shift$ }
define tab_token = ´2000   { $2^8 \cdot tab\_mark$ }
define out_param_token = ´2400   { $2^8 \cdot out\_param$ }
define space_token = ´5040   { $2^8 \cdot spacer$ + "␣" }
define letter_token = ´5400   { $2^8 \cdot letter$ }
define other_token = ´6000   { $2^8 \cdot other\_char$ }
define match_token = ´6400   { $2^8 \cdot match$ }
define end_match_token = ´7000   { $2^8 \cdot end\_match$ }
define protected_token = ´7001   { $2^8 \cdot end\_match + 1$ }
```

294.* The procedure usually "learns" the character code used for macro parameters by seeing one in a *match* command before it runs into any *out_param* commands.

```
⟨ Display the token $(m, c)$ 294* ⟩ ≡
  case m of
  left_brace, right_brace, math_shift, tab_mark, sup_mark, sub_mark, spacer, letter, other_char: print(c);
  mac_param: begin print(c); print(c);
    end;
  out_param: begin print(match_chr);
    if $c \le 9$ then print_char(c + "0")
    else begin print_char("!"); return;
      end;
    end;
  match: begin match_chr ← c; print(c); incr(n); print_char(n);
    if $n >$ "9" then return;
    end;
  end_match: if $c = 0$ then print("->");
    othercases print_esc("BAD.")
  endcases
```

This code is used in section 293.

296* The *print_meaning* subroutine displays *cur_cmd* and *cur_chr* in symbolic form, including the expansion of a macro or mark.

```
procedure print_meaning;
  begin print_cmd_chr(cur_cmd, cur_chr);
  if cur_cmd ≥ call then
    begin print_char(":"); print_ln; token_show(cur_chr);
    end
  else if (cur_cmd = top_bot_mark) ∧ (cur_chr < marks_code) then
      begin print_char(":"); print_ln; token_show(cur_mark[cur_chr]);
      end;
  end;
```

298*. The *print_cmd_chr* routine prints a symbolic interpretation of a command code and its modifier. This is used in certain '`You can´t`' error messages, and in the implementation of diagnostic routines like `\show`.

The body of *print_cmd_chr* is a rather tedious listing of print commands, and most of it is essentially an inverse to the *primitive* routine that enters a TEX primitive into *eqtb*. Therefore much of this procedure appears elsewhere in the program, together with the corresponding *primitive* calls.

```
  define chr_cmd(#) ≡
            begin print(#); print_ASCII(chr_code);
            end
⟨Declare the procedure called print_cmd_chr 298*⟩ ≡
procedure print_cmd_chr(cmd : quarterword; chr_code : halfword);
  var n: integer;  {temp variable}
  begin case cmd of
  left_brace: chr_cmd("begin-group␣character␣");
  right_brace: chr_cmd("end-group␣character␣");
  math_shift: chr_cmd("math␣shift␣character␣");
  mac_param: chr_cmd("macro␣parameter␣character␣");
  sup_mark: chr_cmd("superscript␣character␣");
  sub_mark: chr_cmd("subscript␣character␣");
  endv: print("end␣of␣alignment␣template");
  spacer: chr_cmd("blank␣space␣");
  letter: chr_cmd("the␣letter␣");
  other_char: chr_cmd("the␣character␣");
  ⟨Cases of print_cmd_chr for symbolic printing of primitives 227⟩
  othercases print("[unknown␣command␣code!]")
  endcases;
  end;
```

This code is used in section 252.

299*. Here is a procedure that displays the current command.

```
procedure show_cur_cmd_chr;
  var n: integer;  { level of \if...\fi nesting }
    l: integer;  { line where \if started }
    p: pointer;
  begin begin_diagnostic; print_nl("{");
  if mode ≠ shown_mode then
    begin print_mode(mode); print(":␣"); shown_mode ← mode;
    end;
  print_cmd_chr(cur_cmd, cur_chr);
  if tracing_ifs > 0 then
    if cur_cmd ≥ if_test then
      if cur_cmd ≤ fi_or_else then
        begin print(":␣");
        if cur_cmd = fi_or_else then
          begin print_cmd_chr(if_test, cur_if); print_char("␣"); n ← 0; l ← if_line;
          end
        else begin n ← 1; l ← line;
          end;
        p ← cond_ptr;
        while p ≠ null do
          begin incr(n); p ← link(p);
          end;
        print("(level␣"); print_int(n); print_char(")"); print_if_line(l);
        end;
  print_char("}"); end_diagnostic(false);
  end;
```

303* Let's look more closely now at the control variables (*state*, *index*, *start*, *loc*, *limit*, *name*), assuming that TEX is reading a line of characters that have been input from some file or from the user's terminal. There is an array called *buffer* that acts as a stack of all lines of characters that are currently being read from files, including all lines on subsidiary levels of the input stack that are not yet completed. TEX will return to the other lines when it is finished with the present input file.

(Incidentally, on a machine with byte-oriented addressing, it might be appropriate to combine *buffer* with the *str_pool* array, letting the buffer entries grow downward from the top of the string pool and checking that these two tables don't bump into each other.)

The line we are currently working on begins in position *start* of the buffer; the next character we are about to read is *buffer*[*loc*]; and *limit* is the location of the last character present. If $loc > limit$, the line has been completely read. Usually *buffer*[*limit*] is the *end_line_char*, denoting the end of a line, but this is not true if the current line is an insertion that was entered on the user's terminal in response to an error message.

The *name* variable is a string number that designates the name of the current file, if we are reading a text file. It is zero if we are reading from the terminal; it is $n + 1$ if we are reading from input stream n, where $0 \le n \le 16$. (Input stream 16 stands for an invalid stream number; in such cases the input is actually from the terminal, under control of the procedure *read_toks*.) Finally $18 \le name \le 19$ indicates that we are reading a pseudo file created by the `\scantokens` command.

The *state* variable has one of three values, when we are scanning such files:

1) *state* = *mid_line* is the normal state.

2) *state* = *skip_blanks* is like *mid_line*, but blanks are ignored.

3) *state* = *new_line* is the state at the beginning of a line.

These state values are assigned numeric codes so that if we add the state code to the next character's command code, we get distinct values. For example, '*mid_line* + *spacer*' stands for the case that a blank space character occurs in the middle of a line when it is not being ignored; after this case is processed, the next value of *state* will be *skip_blanks*.

define *mid_line* = 1 { *state* code when scanning a line of characters }
define *skip_blanks* = 2 + *max_char_code* { *state* code when ignoring blanks }
define *new_line* = 3 + *max_char_code* + *max_char_code* { *state* code at start of line }

307* However, all this discussion about input state really applies only to the case that we are inputting from a file. There is another important case, namely when we are currently getting input from a token list. In this case *state* = *token_list*, and the conventions about the other state variables are different:

loc is a pointer to the current node in the token list, i.e., the node that will be read next. If *loc* = *null*, the token list has been fully read.

start points to the first node of the token list; this node may or may not contain a reference count, depending on the type of token list involved.

token_type, which takes the place of *index* in the discussion above, is a code number that explains what kind of token list is being scanned.

name points to the *eqtb* address of the control sequence being expanded, if the current token list is a macro.

param_start, which takes the place of *limit*, tells where the parameters of the current macro begin in the *param_stack*, if the current token list is a macro.

The *token_type* can take several values, depending on where the current token list came from:

parameter, if a parameter is being scanned;
u_template, if the $\langle u_j\rangle$ part of an alignment template is being scanned;
v_template, if the $\langle v_j\rangle$ part of an alignment template is being scanned;
backed_up, if the token list being scanned has been inserted as 'to be read again'.
inserted, if the token list being scanned has been inserted as the text expansion of a `\count` or similar variable;
macro, if a user-defined control sequence is being scanned;
output_text, if an `\output` routine is being scanned;
every_par_text, if the text of `\everypar` is being scanned;
every_math_text, if the text of `\everymath` is being scanned;
every_display_text, if the text of `\everydisplay` is being scanned;
every_hbox_text, if the text of `\everyhbox` is being scanned;
every_vbox_text, if the text of `\everyvbox` is being scanned;
every_job_text, if the text of `\everyjob` is being scanned;
every_cr_text, if the text of `\everycr` is being scanned;
mark_text, if the text of a `\mark` is being scanned;
write_text, if the text of a `\write` is being scanned.

The codes for *output_text*, *every_par_text*, etc., are equal to a constant plus the corresponding codes for token list parameters *output_routine_loc*, *every_par_loc*, etc. The token list begins with a reference count if and only if *token_type* ≥ *macro*.

Since ε-TEX's additional token list parameters precede *toks_base*, the corresponding token types must precede *write_text*.

define *token_list* = 0 { *state* code when scanning a token list }
define *token_type* ≡ *index* { type of current token list }
define *param_start* ≡ *limit* { base of macro parameters in *param_stack* }
define *parameter* = 0 { *token_type* code for parameter }
define *u_template* = 1 { *token_type* code for $\langle u_j\rangle$ template }
define *v_template* = 2 { *token_type* code for $\langle v_j\rangle$ template }
define *backed_up* = 3 { *token_type* code for text to be reread }
define *inserted* = 4 { *token_type* code for inserted texts }
define *macro* = 5 { *token_type* code for defined control sequences }
define *output_text* = 6 { *token_type* code for output routines }
define *every_par_text* = 7 { *token_type* code for `\everypar` }
define *every_math_text* = 8 { *token_type* code for `\everymath` }
define *every_display_text* = 9 { *token_type* code for `\everydisplay` }
define *every_hbox_text* = 10 { *token_type* code for `\everyhbox` }
define *every_vbox_text* = 11 { *token_type* code for `\everyvbox` }

```
define every_job_text = 12  { token_type code for \everyjob }
define every_cr_text = 13  { token_type code for \everycr }
define mark_text = 14  { token_type code for \topmark, etc. }
define eTeX_text_offset = output_routine_loc − output_text
define every_eof_text = every_eof_loc − eTeX_text_offset  { token_type code for \everyeof }
define write_text = toks_base − eTeX_text_offset  { token_type code for \write }
```

311* The status at each level is indicated by printing two lines, where the first line indicates what was read so far and the second line shows what remains to be read. The context is cropped, if necessary, so that the first line contains at most *half_error_line* characters, and the second contains at most *error_line*. Non-current input levels whose *token_type* is '*backed_up*' are shown only if they have not been fully read.

```
procedure show_context; { prints where the scanner is }
  label done;
  var old_setting: 0 .. max_selector;  { saved selector setting }
    nn: integer;  { number of contexts shown so far, less one }
    bottom_line: boolean;  { have we reached the final context to be shown? }
    ⟨ Local variables for formatting calculations 315 ⟩
  begin base_ptr ← input_ptr; input_stack[base_ptr] ← cur_input;  { store current state }
  nn ← −1; bottom_line ← false;
  loop begin cur_input ← input_stack[base_ptr];  { enter into the context }
    if (state ≠ token_list) then
      if (name > 19) ∨ (base_ptr = 0) then bottom_line ← true;
    if (base_ptr = input_ptr) ∨ bottom_line ∨ (nn < error_context_lines) then
      ⟨ Display the current context 312 ⟩
    else if nn = error_context_lines then
        begin print_nl("..."); incr(nn);  { omitted if error_context_lines < 0 }
        end;
    if bottom_line then goto done;
    decr(base_ptr);
    end;
done: cur_input ← input_stack[input_ptr];  { restore original state }
  end;
```

313* This routine should be changed, if necessary, to give the best possible indication of where the current line resides in the input file. For example, on some systems it is best to print both a page and line number.

```
⟨ Print location of current line 313* ⟩ ≡
  if name ≤ 17 then
    if terminal_input then
      if base_ptr = 0 then print_nl("<*>")
      else print_nl("<insert>␣")
    else begin print_nl("<read␣");
      if name = 17 then print_char("*") else print_int(name − 1);
      print_char(">");
      end
  else begin print_nl("l.");
    if index = in_open then print_int(line)
    else print_int(line_stack[index + 1]);  { input from a pseudo file }
    end;
  print_char("␣")
```

This code is used in section 312.

314* ⟨ Print type of token list 314* ⟩ ≡

```
case token_type of
parameter: print_nl("<argument>␣");
u_template, v_template: print_nl("<template>␣");
backed_up: if loc = null then print_nl("<recently␣read>␣")
  else print_nl("<to␣be␣read␣again>␣");
inserted: print_nl("<inserted␣text>␣");
macro: begin print_ln; print_cs(name);
  end;
output_text: print_nl("<output>␣");
every_par_text: print_nl("<everypar>␣");
every_math_text: print_nl("<everymath>␣");
every_display_text: print_nl("<everydisplay>␣");
every_hbox_text: print_nl("<everyhbox>␣");
every_vbox_text: print_nl("<everyvbox>␣");
every_job_text: print_nl("<everyjob>␣");
every_cr_text: print_nl("<everycr>␣");
mark_text: print_nl("<mark>␣");
every_eof_text: print_nl("<everyeof>␣");
write_text: print_nl("<write>␣");
othercases print_nl("?")  { this should never happen }
endcases
```

This code is used in section 312.

326.* ⟨Insert token p into TEX's input 326*⟩ ≡

```
begin $t \gets cur\_tok$; $cur\_tok \gets p$;
if $a$ then
   begin $p \gets get\_avail$; $info(p) \gets cur\_tok$; $link(p) \gets loc$; $loc \gets p$; $start \gets p$;
   if $cur\_tok < right\_brace\_limit$ then
      if $cur\_tok < left\_brace\_limit$ then $decr(align\_state)$
      else $incr(align\_state)$;
   end
else begin $back\_input$; $a \gets eTeX\_ex$;
   end;
$cur\_tok \gets t$;
end
```

This code is used in section 282*.

328.* The *begin_file_reading* procedure starts a new level of input for lines of characters to be read from a file, or as an insertion from the terminal. It does not take care of opening the file, nor does it set *loc* or *limit* or *line*.

```
procedure begin_file_reading;
begin if $in\_open = max\_in\_open$ then overflow("text␣input␣levels", $max\_in\_open$);
if $first = buf\_size$ then overflow("buffer␣size", $buf\_size$);
$incr(in\_open)$; push_input; $index \gets in\_open$; $eof\_seen[index] \gets false$;
$grp\_stack[index] \gets cur\_boundary$; $if\_stack[index] \gets cond\_ptr$; $line\_stack[index] \gets line$; $start \gets first$;
$state \gets mid\_line$; $name \gets 0$;   { terminal_input is now true }
end;
```

329.* Conversely, the variables must be downdated when such a level of input is finished:

```
procedure end_file_reading;
begin $first \gets start$; $line \gets line\_stack[index]$;
if $(name = 18) \lor (name = 19)$ then pseudo_close
else if $name > 17$ then a_close(cur_file);   { forget it }
pop_input; $decr(in\_open)$;
end;
```

331.* To get TEX's whole input mechanism going, we perform the following actions.

⟨Initialize the input routines 331*⟩ ≡

```
begin $input\_ptr \gets 0$; $max\_in\_stack \gets 0$; $in\_open \gets 0$; $open\_parens \gets 0$; $max\_buf\_stack \gets 0$;
$grp\_stack[0] \gets 0$; $if\_stack[0] \gets null$; $param\_ptr \gets 0$; $max\_param\_stack \gets 0$; $first \gets buf\_size$;
repeat $buffer[first] \gets 0$; $decr(first)$;
until $first = 0$;
$scanner\_status \gets normal$; $warning\_index \gets null$; $first \gets 1$; $state \gets new\_line$; $start \gets 1$; $index \gets 0$;
$line \gets 0$; $name \gets 0$; $force\_eof \gets false$; $align\_state \gets 1000000$;
if $\neg init\_terminal$ then goto final_end;
$limit \gets last$; $first \gets last + 1$;   { init_terminal has set loc and last }
end
```

This code is used in section 1337*.

362* ⟨Read next line of file into *buffer*, or **goto** *restart* if the file has ended 362*⟩ ≡

```
begin incr(line); first ← start;
if ¬force_eof then
  if name ≤ 19 then
    begin if pseudo_input then  { not end of file }
      firm_up_the_line  { this sets limit }
    else if (every_eof ≠ null) ∧ ¬eof_seen[index] then
        begin limit ← first − 1; eof_seen[index] ← true;  { fake one empty line }
        begin_token_list(every_eof, every_eof_text); goto restart;
        end
      else force_eof ← true;
    end
  else begin if input_ln(cur_file, true) then  { not end of file }
      firm_up_the_line  { this sets limit }
    else if (every_eof ≠ null) ∧ ¬eof_seen[index] then
        begin limit ← first − 1; eof_seen[index] ← true;  { fake one empty line }
        begin_token_list(every_eof, every_eof_text); goto restart;
        end
      else force_eof ← true;
    end;
if force_eof then
  begin if tracing_nesting > 0 then
    if (grp_stack[in_open] ≠ cur_boundary) ∨ (if_stack[in_open] ≠ cond_ptr) then file_warning;
          { give warning for some unfinished groups and/or conditionals }
  if name ≥ 19 then
    begin print_char(")"); decr(open_parens); update_terminal;  { show user that file has been read }
    end;
  force_eof ← false; end_file_reading;  { resume previous level }
  check_outer_validity; goto restart;
  end;
if end_line_char_inactive then decr(limit)
else buffer[limit] ← end_line_char;
first ← limit + 1; loc ← start;  { ready to read }
end
```

This code is used in section 360.

366*. Expanding the next token. Only a dozen or so command codes > *max_command* can possibly be returned by *get_next*; in increasing order, they are *undefined_cs*, *expand_after*, *no_expand*, *input*, *if_test*, *fi_or_else*, *cs_name*, *convert*, *the*, *top_bot_mark*, *call*, *long_call*, *outer_call*, *long_outer_call*, and *end_template*.

The *expand* subroutine is used when *cur_cmd* > *max_command*. It removes a "call" or a conditional or one of the other special operations just listed. It follows that *expand* might invoke itself recursively. In all cases, *expand* destroys the current token, but it sets things up so that the next *get_next* will deliver the appropriate next token. The value of *cur_tok* need not be known when *expand* is called.

Since several of the basic scanning routines communicate via global variables, their values are saved as local variables of *expand* so that recursive calls don't invalidate them.

```
⟨ Declare the procedure called macro_call 389* ⟩
⟨ Declare the procedure called insert_relax 379 ⟩
⟨ Declare ε-TEX procedures for expanding 1487* ⟩
procedure pass_text; forward;
procedure start_input; forward;
procedure conditional; forward;
procedure get_x_token; forward;
procedure conv_toks; forward;
procedure ins_the_toks; forward;
procedure expand;
  label reswitch;
  var t: halfword; { token that is being "expanded after" }
    p, q, r: pointer; { for list manipulation }
    j: 0 .. buf_size; { index into buffer }
    cv_backup: integer; { to save the global quantity cur_val }
    cvl_backup, radix_backup, co_backup: small_number; { to save cur_val_level, etc. }
    backup_backup: pointer; { to save link(backup_head) }
    save_scanner_status: small_number; { temporary storage of scanner_status }
  begin cv_backup ← cur_val; cvl_backup ← cur_val_level; radix_backup ← radix; co_backup ← cur_order;
  backup_backup ← link(backup_head);
reswitch: if cur_cmd < call then ⟨ Expand a nonmacro 367* ⟩
  else if cur_cmd < end_template then macro_call
    else ⟨ Insert a token containing frozen_endv 375 ⟩;
  cur_val ← cv_backup; cur_val_level ← cvl_backup; radix ← radix_backup; cur_order ← co_backup;
  link(backup_head) ← backup_backup;
  end;
```

367* ⟨ Expand a nonmacro 367* ⟩ ≡

```
  begin if tracing_commands > 1 then show_cur_cmd_chr;
  case cur_cmd of
  top_bot_mark: ⟨ Insert the appropriate mark text into the scanner 386* ⟩;
  expand_after: if cur_chr = 0 then ⟨ Expand the token after the next token 368 ⟩
    else ⟨ Negate a boolean conditional and goto reswitch 1500* ⟩;
  no_expand: ⟨ Suppress expansion of the next token 369 ⟩;
  cs_name: ⟨ Manufacture a control sequence name 372 ⟩;
  convert: conv_toks;   { this procedure is discussed in Part 27 below }
  the: ins_the_toks;   { this procedure is discussed in Part 27 below }
  if_test: conditional;   { this procedure is discussed in Part 28 below }
  fi_or_else: ⟨ Terminate the current conditional and skip to \fi 510* ⟩;
  input: ⟨ Initiate or terminate input from a file 378* ⟩;
  othercases ⟨ Complain about an undefined macro 370 ⟩
  endcases;
  end
```

This code is used in section 366*.

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

```
input: if chr_code = 0 then print_esc("input")
  ⟨ Cases of input for print_cmd_chr 1483* ⟩
else print_esc("endinput");
```

378* ⟨ Initiate or terminate input from a file 378* ⟩ ≡

```
  if cur_chr = 1 then force_eof ← true
  ⟨ Cases for input 1484* ⟩
else if name_in_progress then insert_relax
  else start_input
```

This code is used in section 367*.

382* A control sequence that has been `\def`'ed by the user is expanded by TEX's *macro_call* procedure.

Before we get into the details of *macro_call*, however, let's consider the treatment of primitives like `\topmark`, since they are essentially macros without parameters. The token lists for such marks are kept in a global array of five pointers; we refer to the individual entries of this array by symbolic names *top_mark*, etc. The value of *top_mark* is either *null* or a pointer to the reference count of a token list.

```
  define marks_code ≡ 5   { add this for \topmarks etc. }
  define top_mark_code = 0   { the mark in effect at the previous page break }
  define first_mark_code = 1   { the first mark between top_mark and bot_mark }
  define bot_mark_code = 2   { the mark in effect at the current page break }
  define split_first_mark_code = 3   { the first mark found by \vsplit }
  define split_bot_mark_code = 4   { the last mark found by \vsplit }
  define top_mark ≡ cur_mark[top_mark_code]
  define first_mark ≡ cur_mark[first_mark_code]
  define bot_mark ≡ cur_mark[bot_mark_code]
  define split_first_mark ≡ cur_mark[split_first_mark_code]
  define split_bot_mark ≡ cur_mark[split_bot_mark_code]
```

⟨ Global variables 13 ⟩ +≡

```
cur_mark: array [top_mark_code .. split_bot_mark_code] of pointer;   { token lists for marks }
```

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

```
top_bot_mark: begin case (chr_code mod marks_code) of
  first_mark_code: print_esc("firstmark");
  bot_mark_code: print_esc("botmark");
  split_first_mark_code: print_esc("splitfirstmark");
  split_bot_mark_code: print_esc("splitbotmark");
  othercases print_esc("topmark")
  endcases;
  if chr_code ≥ marks_code then print_char("s");
  end;
```

386* The following code is activated when *cur_cmd* = *top_bot_mark* and when *cur_chr* is a code like *top_mark_code*.

⟨ Insert the appropriate mark text into the scanner 386* ⟩ ≡

```
  begin t ← cur_chr mod marks_code;
  if cur_chr ≥ marks_code then scan_register_num else cur_val ← 0;
  if cur_val = 0 then cur_ptr ← cur_mark[t]
  else ⟨ Compute the mark pointer for mark type t and class cur_val 1559* ⟩;
  if cur_ptr ≠ null then begin_token_list(cur_ptr, mark_text);
  end
```

This code is used in section 367*.

389*. After parameter scanning is complete, the parameters are moved to the *param_stack*. Then the macro body is fed to the scanner; in other words, *macro_call* places the defined text of the control sequence at the top of TEX's input stack, so that *get_next* will proceed to read it next.

The global variable *cur_cs* contains the *eqtb* address of the control sequence being expanded, when *macro_call* begins. If this control sequence has not been declared `\long`, i.e., if its command code in the *eq_type* field is not *long_call* or *long_outer_call*, its parameters are not allowed to contain the control sequence `\par`. If an illegal `\par` appears, the macro call is aborted, and the `\par` will be rescanned.

```
⟨ Declare the procedure called macro_call 389* ⟩ ≡
procedure macro_call; { invokes a user-defined control sequence }
  label exit, continue, done, done1, found;
  var r: pointer; { current node in the macro's token list }
    p: pointer; { current node in parameter token list being built }
    q: pointer; { new node being put into the token list }
    s: pointer; { backup pointer for parameter matching }
    t: pointer; { cycle pointer for backup recovery }
    u, v: pointer; { auxiliary pointers for backup recovery }
    rbrace_ptr: pointer; { one step before the last right_brace token }
    n: small_number; { the number of parameters scanned }
    unbalance: halfword; { unmatched left braces in current parameter }
    m: halfword; { the number of tokens or groups (usually) }
    ref_count: pointer; { start of the token list }
    save_scanner_status: small_number; { scanner_status upon entry }
    save_warning_index: pointer; { warning_index upon entry }
    match_chr: ASCII_code; { character used in parameter }
  begin save_scanner_status ← scanner_status; save_warning_index ← warning_index;
  warning_index ← cur_cs; ref_count ← cur_chr; r ← link(ref_count); n ← 0;
  if tracing_macros > 0 then ⟨ Show the text of the macro being expanded 401 ⟩;
  if info(r) = protected_token then r ← link(r);
  if info(r) ≠ end_match_token then ⟨ Scan the parameters and make link(r) point to the macro body;
        but return if an illegal \par is detected 391 ⟩;
  ⟨ Feed the macro body and its parameters to the scanner 390 ⟩;
exit: scanner_status ← save_scanner_status; warning_index ← save_warning_index;
  end;
```

This code is used in section 366*.

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 ε-TeX procedures for scanning 1413* ⟩
⟨ Declare procedures that scan font-related stuff 577 ⟩
```

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* more than *mem_bot* (dynamic variable-size nodes cannot have these values)

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

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

```
register: ⟨ Cases of register for print_cmd_chr 1567* ⟩;
```

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 }
  label exit;
  var m: halfword;  { chr_code part of the operand token }
    q, r: pointer;  { general purpose indices }
    tx: pointer;  { effective tail node }
    i: four_quarters;  { character info }
    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 ⟩;
exit: end;
```

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 }
         if m = mem_bot then
            begin scan_register_num;
            if cur_val < 256 then cur_val ← equiv(toks_base + cur_val)
            else begin find_sa_element(tok_val, cur_val, false);
               if cur_ptr = null then cur_val ← null
               else cur_val ← sa_ptr(cur_ptr);
               end;
            end
         else cur_val ← sa_ptr(m)
      else cur_val ← equiv(m);
      cur_val_level ← 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*. ε-TEX inserts *last_node_type_code* after *glue_val* and adds the codes for its extensions: *eTeX_version_code*, ...

```
define last_node_type_code = glue_val + 1   { code for \lastnodetype }
define input_line_no_code = glue_val + 2   { code for \inputlineno }
define badness_code = input_line_no_code + 1   { code for \badness }
define eTeX_int = badness_code + 1   { first of ε-TEX codes for integers }
define eTeX_dim = eTeX_int + 8   { first of ε-TEX codes for dimensions }
define eTeX_glue = eTeX_dim + 9   { first of ε-TEX codes for glue }
define eTeX_mu = eTeX_glue + 1   { first of ε-TEX codes for muglue }
define eTeX_expr = eTeX_mu + 1   { first of ε-TEX codes for expressions }
```

⟨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")
  ⟨ Cases of set_page_int for print_cmd_chr 1424* ⟩ 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");
    ⟨ Cases of last_item for print_cmd_chr 1381* ⟩
  othercases print_esc("badness")
  endcases;
```

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

```
  begin if m = 0 then cur_val ← dead_cycles
  ⟨ Cases for ‘Fetch the dead_cycles or the insert_penalties’ 1425* ⟩
else cur_val ← insert_penalties; cur_val_level ← int_val;
  end
```

This code is used in section 413*.

420.* ⟨ Fetch a box dimension 420* ⟩ ≡

```
  begin scan_register_num; fetch_box(q);
  if q = null then cur_val ← 0 else cur_val ← mem[q + m].sc;
  cur_val_level ← dimen_val;
  end
```

This code is used in section 413*.

423.* ⟨ Fetch the *par_shape* size 423* ⟩ ≡

```
  begin if m > par_shape_loc then ⟨ Fetch a penalties array element 1601* ⟩
  else if par_shape_ptr = null then cur_val ← 0
    else cur_val ← info(par_shape_ptr);
  cur_val_level ← int_val;
  end
```

This code is used in section 413*.

424* Here is where `\lastpenalty`, `\lastkern`, `\lastskip`, and `\lastnodetype` are implemented. The reference count for `\lastskip` will be updated later.

We also handle `\inputlineno` and `\badness` here, because they are legal in similar contexts.

The macro *find_effective_tail_eTeX* sets *tx* to the last non-`\endM` node of the current list.

```
define find_effective_tail_eTeX ≡ tx ← tail;
        if ¬is_char_node(tx) then
          if (type(tx) = math_node) ∧ (subtype(tx) = end_M_code) then
            begin r ← head;
            repeat q ← r; r ← link(q);
            until r = tx;
            tx ← q;
            end
define find_effective_tail ≡ find_effective_tail_eTeX
```

```
⟨ Fetch an item in the current node, if appropriate 424* ⟩ ≡
  if m ≥ input_line_no_code then
    if m ≥ eTeX_glue then ⟨ Process an expression and return 1515* ⟩
    else if m ≥ eTeX_dim then
        begin case m of
          ⟨ Cases for fetching a dimension value 1402* ⟩
        end;  { there are no other cases }
        cur_val_level ← dimen_val;
        end
      else begin case m of
        input_line_no_code: cur_val ← line;
        badness_code: cur_val ← last_badness;
          ⟨ Cases for fetching an integer value 1382* ⟩
        end;  { there are no other cases }
        cur_val_level ← int_val;
        end
  else begin if cur_chr = glue_val then cur_val ← zero_glue else cur_val ← 0;
    find_effective_tail;
    if cur_chr = last_node_type_code then
      begin cur_val_level ← int_val;
      if (tx = head) ∨ (mode = 0) then cur_val ← −1;
      end
    else cur_val_level ← cur_chr;
    if ¬is_char_node(tx) ∧ (mode ≠ 0) then
      case cur_chr of
      int_val: if type(tx) = penalty_node then cur_val ← penalty(tx);
      dimen_val: if type(tx) = kern_node then cur_val ← width(tx);
      glue_val: if type(tx) = glue_node then
          begin cur_val ← glue_ptr(tx);
          if subtype(tx) = mu_glue then cur_val_level ← mu_val;
          end;
      last_node_type_code: if type(tx) ≤ unset_node then cur_val ← type(tx) + 1
        else cur_val ← unset_node + 2;
      end  { there are no other cases }
    else if (mode = vmode) ∧ (tx = head) then
        case cur_chr of
        int_val: cur_val ← last_penalty;
        dimen_val: cur_val ← last_kern;
        glue_val: if last_glue ≠ max_halfword then cur_val ← last_glue;
        last_node_type_code: cur_val ← last_node_type;
        end; { there are no other cases }
    end
```

This code is used in section 413*.

427* ⟨Fetch a register 427*⟩ ≡

```
  begin if (m < mem_bot) ∨ (m > lo_mem_stat_max) then
    begin cur_val_level ← sa_type(m);
    if cur_val_level < glue_val then cur_val ← sa_int(m)
    else cur_val ← sa_ptr(m);
    end
  else begin scan_register_num; cur_val_level ← m − mem_bot;
    if cur_val > 255 then
      begin find_sa_element(cur_val_level, cur_val, false);
      if cur_ptr = null then
        if cur_val_level < glue_val then cur_val ← 0
        else cur_val ← zero_glue
      else if cur_val_level < glue_val then cur_val ← sa_int(cur_ptr)
        else cur_val ← sa_ptr(cur_ptr);
      end
    else case cur_val_level of
      int_val: cur_val ← count(cur_val);
      dimen_val: cur_val ← dimen(cur_val);
      glue_val: cur_val ← skip(cur_val);
      mu_val: cur_val ← mu_skip(cur_val);
      end; { there are no other cases }
    end;
  end
```

This code is used in section 413*.

461* The final member of TEX's value-scanning trio is *scan_glue*, which makes *cur_val* point to a glue specification. The reference count of that glue spec will take account of the fact that *cur_val* is pointing to it.

The *level* parameter should be either *glue_val* or *mu_val*.

Since *scan_dimen* was so much more complex than *scan_int*, we might expect *scan_glue* to be even worse. But fortunately, it is very simple, since most of the work has already been done.

```
procedure scan_glue(level : small_number);   { sets cur_val to a glue spec pointer }
  label exit;
  var negative: boolean;   { should the answer be negated? }
    q: pointer;   { new glue specification }
    mu: boolean;   { does level = mu_val? }
  begin mu ← (level = mu_val); ⟨ Get the next non-blank non-sign token; set negative appropriately 441 ⟩;
  if (cur_cmd ≥ min_internal) ∧ (cur_cmd ≤ max_internal) then
    begin scan_something_internal(level, negative);
    if cur_val_level ≥ glue_val then
      begin if cur_val_level ≠ level then mu_error;
      return;
      end;
    if cur_val_level = int_val then scan_dimen(mu, false, true)
    else if level = mu_val then mu_error;
    end
  else begin back_input; scan_dimen(mu, false, false);
    if negative then negate(cur_val);
    end;
  ⟨ Create a new glue specification whose width is cur_val; scan for its stretch and shrink components 462 ⟩;
exit: end;
```

⟨ Declare procedures needed for expressions 1517* ⟩

464.* Building token lists. The token lists for macros and for other things like `\mark` and `\output` and `\write` are produced by a procedure called *scan_toks*.

Before we get into the details of *scan_toks*, let's consider a much simpler task, that of converting the current string into a token list. The *str_toks* function does this; it classifies spaces as type *spacer* and everything else as type *other_char*.

The token list created by *str_toks* begins at *link*(*temp_head*) and ends at the value p that is returned. (If $p =$ *temp_head*, the list is empty.)

```
⟨ Declare ε-TEX procedures for token lists 1414* ⟩
function str_toks(b : pool_pointer): pointer;  { changes the string str_pool[b .. pool_ptr] to a token list }
  var p: pointer;  { tail of the token list }
    q: pointer;  { new node being added to the token list via store_new_token }
    t: halfword;  { token being appended }
    k: pool_pointer;  { index into str_pool }
  begin str_room(1); p ← temp_head; link(p) ← null; k ← b;
  while k < pool_ptr do
    begin t ← so(str_pool[k]);
    if t = "␣" then t ← space_token
    else t ← other_token + t;
    fast_store_new_token(t); incr(k);
    end;
  pool_ptr ← b; str_toks ← p;
  end;
```

465.* The main reason for wanting *str_toks* is the next function, *the_toks*, which has similar input/output characteristics.

This procedure is supposed to scan something like '`\skip\count12`', i.e., whatever can follow '`\the`', and it constructs a token list containing something like '`-3.0pt minus 0.5fill`'.

```
function the_toks: pointer;
  label exit;
  var old_setting: 0 .. max_selector;  { holds selector setting }
    p, q, r: pointer;  { used for copying a token list }
    b: pool_pointer;  { base of temporary string }
    c: small_number;  { value of cur_chr }
  begin ⟨ Handle \unexpanded or \detokenize and return 1419* ⟩;
  get_x_token; scan_something_internal(tok_val, false);
  if cur_val_level ≥ ident_val then ⟨ Copy the token list 466 ⟩
  else begin old_setting ← selector; selector ← new_string; b ← pool_ptr;
    case cur_val_level of
    int_val: print_int(cur_val);
    dimen_val: begin print_scaled(cur_val); print("pt");
      end;
    glue_val: begin print_spec(cur_val, "pt"); delete_glue_ref(cur_val);
      end;
    mu_val: begin print_spec(cur_val, "mu"); delete_glue_ref(cur_val);
      end;
    end;  { there are no other cases }
    selector ← old_setting; the_toks ← str_toks(b);
    end;
exit: end;
```

468* The primitives `\number`, `\romannumeral`, `\string`, `\meaning`, `\fontname`, and `\jobname` are defined as follows.

ε-TeX adds `\eTeXrevision` such that *job_name_code* remains last.

```
define number_code = 0   { command code for \number }
define roman_numeral_code = 1   { command code for \romannumeral }
define string_code = 2   { command code for \string }
define meaning_code = 3   { command code for \meaning }
define font_name_code = 4   { command code for \fontname }
define etex_convert_base = 5   { base for ε-TeX's command codes }
define eTeX_revision_code = etex_convert_base   { command code for \eTeXrevision }
define etex_convert_codes = etex_convert_base + 1   { end of ε-TeX's command codes }
define job_name_code = etex_convert_codes   { command code for \jobname }
```

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

```
  primitive("number", convert, number_code);
  primitive("romannumeral", convert, roman_numeral_code);
  primitive("string", convert, string_code);
  primitive("meaning", convert, meaning_code);
  primitive("fontname", convert, font_name_code);
  primitive("jobname", convert, job_name_code);
```

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

```
convert: case chr_code of
  number_code: print_esc("number");
  roman_numeral_code: print_esc("romannumeral");
  string_code: print_esc("string");
  meaning_code: print_esc("meaning");
  font_name_code: print_esc("fontname");
  eTeX_revision_code: print_esc("eTeXrevision");
  othercases print_esc("jobname")
  endcases;
```

471* ⟨Scan the argument for command *c* 471*⟩ ≡

```
  case c of
  number_code, roman_numeral_code: scan_int;
  string_code, meaning_code: begin save_scanner_status ← scanner_status; scanner_status ← normal;
    get_token; scanner_status ← save_scanner_status;
    end;
  font_name_code: scan_font_ident;
  eTeX_revision_code: do_nothing;
  job_name_code: if job_name = 0 then open_log_file;
  end   { there are no other cases }
```

This code is used in section 470.

472* ⟨ Print the result of command *c* 472* ⟩ ≡

```
case c of
number_code: print_int(cur_val);
roman_numeral_code: print_roman_int(cur_val);
string_code: if cur_cs ≠ 0 then sprint_cs(cur_cs)
  else print_char(cur_chr);
meaning_code: print_meaning;
font_name_code: begin print(font_name[cur_val]);
  if font_size[cur_val] ≠ font_dsize[cur_val] then
    begin print("␣at␣"); print_scaled(font_size[cur_val]); print("pt");
    end;
  end;
eTeX_revision_code: print(eTeX_revision);
job_name_code: print(job_name);
end { there are no other cases }
```

This code is used in section 470.

478* Here we insert an entire token list created by *the_toks* without expanding it further.

⟨ Expand the next part of the input 478* ⟩ ≡

```
  begin loop
    begin get_next;
    if cur_cmd ≥ call then
      if info(link(cur_chr)) = protected_token then
        begin cur_cmd ← relax; cur_chr ← no_expand_flag;
        end;
    if cur_cmd ≤ max_command then goto done2;
    if cur_cmd ≠ the then expand
    else begin q ← the_toks;
      if link(temp_head) ≠ null then
        begin link(p) ← link(temp_head); p ← q;
        end;
      end;
    end;
done2: x_token
  end
```

This code is used in section 477.

482* The *read_toks* procedure constructs a token list like that for any macro definition, and makes *cur_val* point to it. Parameter *r* points to the control sequence that will receive this token list.

```
procedure read_toks(n : integer; r : pointer; j : halfword);
  label done;
  var p: pointer;   { tail of the token list }
    q: pointer;   { new node being added to the token list via store_new_token }
    s: integer;   { saved value of align_state }
    m: small_number;   { stream number }
  begin scanner_status ← defining; warning_index ← r; def_ref ← get_avail;
  token_ref_count(def_ref) ← null; p ← def_ref;   { the reference count }
  store_new_token(end_match_token);
  if $(n < 0) \lor (n > 15)$ then m ← 16 else m ← n;
  s ← align_state; align_state ← 1000000;   { disable tab marks, etc. }
  repeat ⟨Input and store tokens from the next line of the file 483*⟩;
  until align_state = 1000000;
  cur_val ← def_ref; scanner_status ← normal; align_state ← s;
  end;
```

483* ⟨Input and store tokens from the next line of the file 483*⟩ ≡

```
  begin_file_reading; name ← m + 1;
  if read_open[m] = closed then ⟨Input for \read from the terminal 484⟩
  else if read_open[m] = just_open then ⟨Input the first line of read_file[m] 485⟩
    else ⟨Input the next line of read_file[m] 486⟩;
  limit ← last;
  if end_line_char_inactive then decr(limit)
  else buffer[limit] ← end_line_char;
  first ← limit + 1; loc ← start; state ← new_line;
  ⟨Handle \readline and goto done 1496*⟩;
  loop begin get_token;
    if cur_tok = 0 then goto done;   { cur_cmd = cur_chr = 0 will occur at the end of the line }
    if align_state < 1000000 then   { unmatched '}' aborts the line }
      begin repeat get_token;
      until cur_tok = 0;
      align_state ← 1000000; goto done;
      end;
    store_new_token(cur_tok);
    end;
done: end_file_reading
```

This code is used in section 482*.

487*. Conditional processing. We consider now the way TEX handles various kinds of `\if` commands.

define *unless_code* = 32 { amount added for '`\unless`' prefix }

```
define if_char_code = 0   { '\if' }
define if_cat_code = 1   { '\ifcat' }
define if_int_code = 2   { '\ifnum' }
define if_dim_code = 3   { '\ifdim' }
define if_odd_code = 4   { '\ifodd' }
define if_vmode_code = 5   { '\ifvmode' }
define if_hmode_code = 6   { '\ifhmode' }
define if_mmode_code = 7   { '\ifmmode' }
define if_inner_code = 8   { '\ifinner' }
define if_void_code = 9   { '\ifvoid' }
define if_hbox_code = 10   { '\ifhbox' }
define if_vbox_code = 11   { '\ifvbox' }
define ifx_code = 12   { '\ifx' }
define if_eof_code = 13   { '\ifeof' }
define if_true_code = 14   { '\iftrue' }
define if_false_code = 15   { '\iffalse' }
define if_case_code = 16   { '\ifcase' }
```

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

```
primitive("if", if_test, if_char_code); primitive("ifcat", if_test, if_cat_code);
primitive("ifnum", if_test, if_int_code); primitive("ifdim", if_test, if_dim_code);
primitive("ifodd", if_test, if_odd_code); primitive("ifvmode", if_test, if_vmode_code);
primitive("ifhmode", if_test, if_hmode_code); primitive("ifmmode", if_test, if_mmode_code);
primitive("ifinner", if_test, if_inner_code); primitive("ifvoid", if_test, if_void_code);
primitive("ifhbox", if_test, if_hbox_code); primitive("ifvbox", if_test, if_vbox_code);
primitive("ifx", if_test, ifx_code); primitive("ifeof", if_test, if_eof_code);
primitive("iftrue", if_test, if_true_code); primitive("iffalse", if_test, if_false_code);
primitive("ifcase", if_test, if_case_code);
```

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

```
if_test: begin if chr_code ≥ unless_code then print_esc("unless");
  case chr_code mod unless_code of
  if_cat_code: print_esc("ifcat");
  if_int_code: print_esc("ifnum");
  if_dim_code: print_esc("ifdim");
  if_odd_code: print_esc("ifodd");
  if_vmode_code: print_esc("ifvmode");
  if_hmode_code: print_esc("ifhmode");
  if_mmode_code: print_esc("ifmmode");
  if_inner_code: print_esc("ifinner");
  if_void_code: print_esc("ifvoid");
  if_hbox_code: print_esc("ifhbox");
  if_vbox_code: print_esc("ifvbox");
  ifx_code: print_esc("ifx");
  if_eof_code: print_esc("ifeof");
  if_true_code: print_esc("iftrue");
  if_false_code: print_esc("iffalse");
  if_case_code: print_esc("ifcase");
    ⟨ Cases of if_test for print_cmd_chr 1499* ⟩
  othercases print_esc("if")
  endcases;
  end;
```

494* Here is a procedure that ignores text until coming to an `\or`, `\else`, or `\fi` at level zero of `\if`...`\fi` nesting. After it has acted, *cur_chr* will indicate the token that was found, but *cur_tok* will not be set (because this makes the procedure run faster).

```
procedure pass_text;
  label done;
  var l: integer;   { level of \if...\fi nesting }
    save_scanner_status: small_number;   { scanner_status upon entry }
  begin save_scanner_status ← scanner_status; scanner_status ← skipping; l ← 0; skip_line ← line;
  loop begin get_next;
    if cur_cmd = fi_or_else then
      begin if l = 0 then goto done;
      if cur_chr = fi_code then decr(l);
      end
    else if cur_cmd = if_test then incr(l);
    end;
done: scanner_status ← save_scanner_status;
  if tracing_ifs > 0 then show_cur_cmd_chr;
  end;
```

496* ⟨ Pop the condition stack 496* ⟩ ≡

```
  begin if if_stack[in_open] = cond_ptr then if_warning;
        { conditionals possibly not properly nested with files }
  p ← cond_ptr; if_line ← if_line_field(p); cur_if ← subtype(p); if_limit ← type(p); cond_ptr ← link(p);
  free_node(p, if_node_size);
  end
```

This code is used in sections 498*, 500, 509, and 510*.

498*. A condition is started when the *expand* procedure encounters an *if_test* command; in that case *expand* reduces to *conditional*, which is a recursive procedure.

```
procedure conditional;
  label exit, common_ending;
  var b: boolean;   { is the condition true? }
    r: "<" .. ">";   { relation to be evaluated }
    m, n: integer;   { to be tested against the second operand }
    p, q: pointer;   { for traversing token lists in \ifx tests }
    save_scanner_status: small_number;   { scanner_status upon entry }
    save_cond_ptr: pointer;   { cond_ptr corresponding to this conditional }
    this_if: small_number;   { type of this conditional }
    is_unless: boolean;   { was this if preceded by '\unless' ? }
  begin if tracing_ifs $> 0$ then
    if tracing_commands $\le 1$ then show_cur_cmd_chr;
  ⟨ Push the condition stack 495 ⟩; save_cond_ptr $\leftarrow$ cond_ptr; is_unless $\leftarrow$ (cur_chr $\ge$ unless_code);
  this_if $\leftarrow$ cur_chr mod unless_code;
  ⟨ Either process \ifcase or set b to the value of a boolean condition 501* ⟩;
  if is_unless then $b \leftarrow \neg b$;
  if tracing_commands $> 1$ then ⟨ Display the value of b 502 ⟩;
  if b then
    begin change_if_limit(else_code, save_cond_ptr); return;   { wait for \else or \fi }
    end;
  ⟨ Skip to \else or \fi, then goto common_ending 500 ⟩;
common_ending: if cur_chr = fi_code then ⟨ Pop the condition stack 496* ⟩
  else if_limit $\leftarrow$ fi_code;   { wait for \fi }
exit: end;
```

501*. ⟨ Either process `\ifcase` or set *b* to the value of a boolean condition 501* ⟩ ≡

```
  case this_if of
  if_char_code, if_cat_code: ⟨ Test if two characters match 506 ⟩;
  if_int_code, if_dim_code: ⟨ Test relation between integers or dimensions 503 ⟩;
  if_odd_code: ⟨ Test if an integer is odd 504 ⟩;
  if_vmode_code: $b \leftarrow (abs(mode) = vmode)$;
  if_hmode_code: $b \leftarrow (abs(mode) = hmode)$;
  if_mmode_code: $b \leftarrow (abs(mode) = mmode)$;
  if_inner_code: $b \leftarrow (mode < 0)$;
  if_void_code, if_hbox_code, if_vbox_code: ⟨ Test box register status 505* ⟩;
  ifx_code: ⟨ Test if two tokens match 507 ⟩;
  if_eof_code: begin scan_four_bit_int; $b \leftarrow (read\_open[cur\_val] = closed)$;
    end;
  if_true_code: $b \leftarrow true$;
  if_false_code: $b \leftarrow false$;
    ⟨ Cases for conditional 1501* ⟩
  if_case_code: ⟨ Select the appropriate case and return or goto common_ending 509 ⟩;
  end   { there are no other cases }
```

This code is used in section 498*.

505* ⟨ Test box register status 505* ⟩ ≡

```
begin scan_register_num; fetch_box(p);
if this_if = if_void_code then b ← (p = null)
else if p = null then b ← false
  else if this_if = if_hbox_code then b ← (type(p) = hlist_node)
    else b ← (type(p) = vlist_node);
end
```

This code is used in section 501*.

510* The processing of conditionals is complete except for the following code, which is actually part of *expand*. It comes into play when `\or`, `\else`, or `\fi` is scanned.

⟨ Terminate the current conditional and skip to `\fi` 510* ⟩ ≡

```
begin if tracing_ifs > 0 then
  if tracing_commands ≤ 1 then show_cur_cmd_chr;
if cur_chr > if_limit then
  if if_limit = if_code then insert_relax   { condition not yet evaluated }
  else begin print_err("Extra␣"); print_cmd_chr(fi_or_else, cur_chr);
    help1("I´m␣ignoring␣this;␣it␣doesn´t␣match␣any␣\if."); error;
    end
else begin while cur_chr ≠ fi_code do pass_text;   { skip to \fi }
  ⟨ Pop the condition stack 496* ⟩;
  end;
end
```

This code is used in section 367*.

536* ⟨Print the banner line, including the date and time 536*⟩ ≡

```
begin wlog(banner); slow_print(format_ident); print("␣␣"); print_int(day); print_char("␣");
months ← ´JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
for k ← 3 * month − 2 to 3 * month do wlog(months[k]);
print_char("␣"); print_int(year); print_char("␣"); print_two(time div 60); print_char(":");
print_two(time mod 60);
if eTeX_ex then
  begin ; wlog_cr; wlog(´entering␣extended␣mode´);
  end;
end
```

This code is used in section 534.

581*. When TEX wants to typeset a character that doesn't exist, the character node is not created; thus the output routine can assume that characters exist when it sees them. The following procedure prints a warning message unless the user has suppressed it.

```
procedure char_warning(f : internal_font_number; c : eight_bits);
  var old_setting: integer;  { saved value of tracing_online }
  begin if tracing_lost_chars > 0 then
    begin old_setting ← tracing_online;
    if eTeX_ex ∧ (tracing_lost_chars > 1) then tracing_online ← 1;
    begin begin_diagnostic; print_nl("Missing␣character:␣There␣is␣no␣"); print_ASCII(c);
    print("␣in␣font␣"); slow_print(font_name[f]); print_char("!"); end_diagnostic(false);
    end; tracing_online ← old_setting;
    end;
  end;
```

616* The actual distances by which we want to move might be computed as the sum of several separate movements. For example, there might be several glue nodes in succession, or we might want to move right by the width of some box plus some amount of glue. More importantly, the baselineskip distances are computed in terms of glue together with the depth and height of adjacent boxes, and we want the `DVI` file to lump these three quantities together into a single motion.

Therefore, TEX maintains two pairs of global variables: *dvi_h* and *dvi_v* are the h and v coordinates corresponding to the commands actually output to the `DVI` file, while *cur_h* and *cur_v* are the coordinates corresponding to the current state of the output routines. Coordinate changes will accumulate in *cur_h* and *cur_v* without being reflected in the output, until such a change becomes necessary or desirable; we can call the *movement* procedure whenever we want to make *dvi_h* = *cur_h* or *dvi_v* = *cur_v*.

The current font reflected in the `DVI` output is called *dvi_f*; there is no need for a '*cur_f*' variable.

The depth of nesting of *hlist_out* and *vlist_out* is called *cur_s*; this is essentially the depth of *push* commands in the `DVI` output.

For mixed direction text (TEX--**XƎT**) the current text direction is called *cur_dir*. As the box being shipped out will never be used again and soon be recycled, we can simply reverse any R-text (i.e., right-to-left) segments of hlist nodes as well as complete hlist nodes embedded in such segments. Moreover this can be done iteratively rather than recursively. There are, however, two complications related to leaders that require some additional bookkeeping: (1) One and the same hlist node might be used more than once (but never inside both L- and R-text); and (2) leader boxes inside hlists must be aligned with respect to the left edge of the original hlist.

A math node is changed into a kern node whenever the text direction remains the same, it is replaced by an *edge_node* if the text direction changes; the subtype of an an *hlist_node* inside R-text is changed to *reversed* once its hlist has been reversed.

```
define reversed = 1   { subtype for an hlist_node whose hlist has been reversed }
define dlist = 2   { subtype for an hlist_node from display math mode }
define box_lr(#) ≡ (qo(subtype(#)))   { direction mode of a box }
define set_box_lr(#) ≡ subtype(#) ← set_box_lr_end
define set_box_lr_end(#) ≡ qi(#)

define left_to_right = 0
define right_to_left = 1
define reflected ≡ 1 − cur_dir   { the opposite of cur_dir }

define synch_h ≡
        if cur_h ≠ dvi_h then
          begin movement(cur_h − dvi_h, right1); dvi_h ← cur_h;
          end
define synch_v ≡
        if cur_v ≠ dvi_v then
          begin movement(cur_v − dvi_v, down1); dvi_v ← cur_v;
          end
```

```
⟨ Global variables 13 ⟩ +≡
dvi_h, dvi_v: scaled;   { a DVI reader program thinks we are here }
cur_h, cur_v: scaled;   { TEX thinks we are here }
dvi_f: internal_font_number;   { the current font }
cur_s: integer;   { current depth of output box nesting, initially −1 }
```

619*. The recursive procedures *hlist_out* and *vlist_out* each have local variables *save_h* and *save_v* to hold the values of *dvi_h* and *dvi_v* just before entering a new level of recursion. In effect, the values of *save_h* and *save_v* on TEX's run-time stack correspond to the values of *h* and *v* that a DVI-reading program will push onto its coordinate stack.

```
  define move_past = 13  { go to this label when advancing past glue or a rule }
  define fin_rule = 14  { go to this label to finish processing a rule }
  define next_p = 15  { go to this label when finished with node p }

⟨ Declare procedures needed in hlist_out, vlist_out 1368 ⟩
procedure hlist_out;  { output an hlist_node box }
  label reswitch, move_past, fin_rule, next_p;
  var base_line: scaled;  { the baseline coordinate for this box }
    left_edge: scaled;  { the left coordinate for this box }
    save_h, save_v: scaled;  { what dvi_h and dvi_v should pop to }
    this_box: pointer;  { pointer to containing box }
    g_order: glue_ord;  { applicable order of infinity for glue }
    g_sign: normal .. shrinking;  { selects type of glue }
    p: pointer;  { current position in the hlist }
    save_loc: integer;  { DVI byte location upon entry }
    leader_box: pointer;  { the leader box being replicated }
    leader_wd: scaled;  { width of leader box being replicated }
    lx: scaled;  { extra space between leader boxes }
    outer_doing_leaders: boolean;  { were we doing leaders? }
    edge: scaled;  { right edge of sub-box or leader space }
    prev_p: pointer;  { one step behind p }
    glue_temp: real;  { glue value before rounding }
    cur_glue: real;  { glue seen so far }
    cur_g: scaled;  { rounded equivalent of cur_glue times the glue ratio }
  begin cur_g ← 0; cur_glue ← float_constant(0); this_box ← temp_ptr; g_order ← glue_order(this_box);
  g_sign ← glue_sign(this_box); p ← list_ptr(this_box); incr(cur_s);
  if $cur\_s > 0$ then dvi_out(push);
  if $cur\_s > max\_push$ then max_push ← cur_s;
  save_loc ← dvi_offset + dvi_ptr; base_line ← cur_v; prev_p ← this_box + list_offset;
  ⟨ Initialize hlist_out for mixed direction typesetting 1445* ⟩;
  left_edge ← cur_h;
  while $p \neq null$ do ⟨ Output node p for hlist_out and move to the next node, maintaining the condition
        $cur\_v = base\_line$ 620* ⟩;
  ⟨ Finish hlist_out for mixed direction typesetting 1446* ⟩;
  prune_movements(save_loc);
  if $cur\_s > 0$ then dvi_pop(save_loc);
  decr(cur_s);
  end;
```

620.* We ought to give special care to the efficiency of one part of *hlist_out*, since it belongs to TEX's inner loop. When a *char_node* is encountered, we save a little time by processing several nodes in succession until reaching a non-*char_node*. The program uses the fact that *set_char_0* = 0.

```
⟨ Output node p for hlist_out and move to the next node, maintaining the condition cur_v = base_line 620* ⟩ ≡
reswitch: if is_char_node(p) then
    begin synch_h; synch_v;
    repeat f ← font(p); c ← character(p);
      if f ≠ dvi_f then ⟨ Change font dvi_f to f 621 ⟩;
      if c ≥ qi(128) then dvi_out(set1);
      dvi_out(qo(c));
      cur_h ← cur_h + char_width(f)(char_info(f)(c)); prev_p ← link(prev_p);
          { N.B.: not prev_p ← p, p might be lig_trick }
      p ← link(p);
    until ¬is_char_node(p);
    dvi_h ← cur_h;
    end
  else ⟨ Output the non-char_node p for hlist_out and move to the next node 622* ⟩
```

This code is used in section 619*.

622.*

```
⟨ Output the non-char_node p for hlist_out and move to the next node 622* ⟩ ≡
  begin case type(p) of
  hlist_node, vlist_node: ⟨ Output a box in an hlist 623* ⟩;
  rule_node: begin rule_ht ← height(p); rule_dp ← depth(p); rule_wd ← width(p); goto fin_rule;
    end;
  whatsit_node: ⟨ Output the whatsit node p in an hlist 1367 ⟩;
  glue_node: ⟨ Move right or output leaders 625* ⟩;
  kern_node: cur_h ← cur_h + width(p);
  math_node: ⟨ Handle a math node in hlist_out 1447* ⟩;
  ligature_node: ⟨ Make node p look like a char_node and goto reswitch 652 ⟩;
    ⟨ Cases of hlist_out that arise in mixed direction text only 1451* ⟩
  othercases do_nothing
  endcases;
  goto next_p;
fin_rule: ⟨ Output a rule in an hlist 624 ⟩;
move_past: cur_h ← cur_h + rule_wd;
next_p: prev_p ← p; p ← link(p);
  end
```

This code is used in section 620*.

623.*

```
⟨ Output a box in an hlist 623* ⟩ ≡
  if list_ptr(p) = null then cur_h ← cur_h + width(p)
  else begin save_h ← dvi_h; save_v ← dvi_v; cur_v ← base_line + shift_amount(p);
        { shift the box down }
    temp_ptr ← p; edge ← cur_h + width(p);
    if cur_dir = right_to_left then cur_h ← edge;
    if type(p) = vlist_node then vlist_out else hlist_out;
    dvi_h ← save_h; dvi_v ← save_v; cur_h ← edge; cur_v ← base_line;
    end
```

This code is used in section 622*.

```
625.* define billion ≡ float_constant(1000000000)
  define vet_glue(#) ≡ glue_temp ← #;
        if glue_temp > billion then glue_temp ← billion
        else if glue_temp < −billion then glue_temp ← −billion
  define round_glue ≡ g ← glue_ptr(p); rule_wd ← width(g) − cur_g;
        if g_sign ≠ normal then
          begin if g_sign = stretching then
            begin if stretch_order(g) = g_order then
              begin cur_glue ← cur_glue + stretch(g); vet_glue(float(glue_set(this_box)) ∗ cur_glue);
              cur_g ← round(glue_temp);
              end;
            end
          else if shrink_order(g) = g_order then
              begin cur_glue ← cur_glue − shrink(g); vet_glue(float(glue_set(this_box)) ∗ cur_glue);
              cur_g ← round(glue_temp);
              end;
          end;
        rule_wd ← rule_wd + cur_g

⟨ Move right or output leaders 625* ⟩ ≡
  begin round_glue;
  if eTeX_ex then ⟨ Handle a glue node for mixed direction typesetting 1430* ⟩;
  if subtype(p) ≥ a_leaders then
    ⟨ Output leaders in an hlist, goto fin_rule if a rule or to next_p if done 626* ⟩;
  goto move_past;
  end
```

This code is used in section 622*.

```
626.* ⟨ Output leaders in an hlist, goto fin_rule if a rule or to next_p if done 626* ⟩ ≡
  begin leader_box ← leader_ptr(p);
  if type(leader_box) = rule_node then
    begin rule_ht ← height(leader_box); rule_dp ← depth(leader_box); goto fin_rule;
    end;
  leader_wd ← width(leader_box);
  if (leader_wd > 0) ∧ (rule_wd > 0) then
    begin rule_wd ← rule_wd + 10;  { compensate for floating-point rounding }
    if cur_dir = right_to_left then cur_h ← cur_h − 10;
    edge ← cur_h + rule_wd; lx ← 0; ⟨ Let cur_h be the position of the first box, and set leader_wd + lx to
          the spacing between corresponding parts of boxes 627 ⟩;
    while cur_h + leader_wd ≤ edge do
      ⟨ Output a leader box at cur_h, then advance cur_h by leader_wd + lx 628* ⟩;
    if cur_dir = right_to_left then cur_h ← edge
    else cur_h ← edge − 10;
    goto next_p;
    end;
  end
```

This code is used in section 625*.

628* The '*synch*' operations here are intended to decrease the number of bytes needed to specify horizontal and vertical motion in the DVI output.

```
⟨ Output a leader box at cur_h, then advance cur_h by leader_wd + lx 628* ⟩ ≡
  begin cur_v ← base_line + shift_amount(leader_box); synch_v; save_v ← dvi_v;
  synch_h; save_h ← dvi_h; temp_ptr ← leader_box;
  if cur_dir = right_to_left then cur_h ← cur_h + leader_wd;
  outer_doing_leaders ← doing_leaders; doing_leaders ← true;
  if type(leader_box) = vlist_node then vlist_out else hlist_out;
  doing_leaders ← outer_doing_leaders; dvi_v ← save_v; dvi_h ← save_h; cur_v ← base_line;
  cur_h ← save_h + leader_wd + lx;
  end
```

This code is used in section 626*.

632* The *synch_v* here allows the DVI output to use one-byte commands for adjusting v in most cases, since the baselineskip distance will usually be constant.

```
⟨ Output a box in a vlist 632* ⟩ ≡
  if list_ptr(p) = null then cur_v ← cur_v + height(p) + depth(p)
  else begin cur_v ← cur_v + height(p); synch_v; save_h ← dvi_h; save_v ← dvi_v;
    if cur_dir = right_to_left then cur_h ← left_edge − shift_amount(p)
    else cur_h ← left_edge + shift_amount(p); { shift the box right }
    temp_ptr ← p;
    if type(p) = vlist_node then vlist_out else hlist_out;
    dvi_h ← save_h; dvi_v ← save_v; cur_v ← save_v + depth(p); cur_h ← left_edge;
    end
```

This code is used in section 631.

633* ⟨ Output a rule in a vlist, **goto** *next_p* 633* ⟩ ≡

```
  if is_running(rule_wd) then rule_wd ← width(this_box);
  rule_ht ← rule_ht + rule_dp; { this is the rule thickness }
  cur_v ← cur_v + rule_ht;
  if (rule_ht > 0) ∧ (rule_wd > 0) then { we don't output empty rules }
    begin if cur_dir = right_to_left then cur_h ← cur_h − rule_wd;
    synch_h; synch_v; dvi_out(put_rule); dvi_four(rule_ht); dvi_four(rule_wd); cur_h ← left_edge;
    end;
  goto next_p
```

This code is used in section 631.

637* When we reach this part of the program, *cur_v* indicates the top of a leader box, not its baseline.

```
⟨ Output a leader box at cur_v, then advance cur_v by leader_ht + lx 637* ⟩ ≡
  begin if cur_dir = right_to_left then cur_h ← left_edge − shift_amount(leader_box)
  else cur_h ← left_edge + shift_amount(leader_box);
  synch_h; save_h ← dvi_h;
  cur_v ← cur_v + height(leader_box); synch_v; save_v ← dvi_v; temp_ptr ← leader_box;
  outer_doing_leaders ← doing_leaders; doing_leaders ← true;
  if type(leader_box) = vlist_node then vlist_out else hlist_out;
  doing_leaders ← outer_doing_leaders; dvi_v ← save_v; dvi_h ← save_h; cur_h ← left_edge;
  cur_v ← save_v − height(leader_box) + leader_ht + lx;
  end
```

This code is used in section 635.

638* The *hlist_out* and *vlist_out* procedures are now complete, so we are ready for the *ship_out* routine that gets them started in the first place.

```
procedure ship_out(p : pointer);   { output the box p }
  label done;
  var page_loc: integer;   { location of the current bop }
    j, k: 0 . . 9;   { indices to first ten count registers }
    s: pool_pointer;   { index into str_pool }
    old_setting: 0 . . max_selector;   { saved selector setting }
  begin if tracing_output > 0 then
    begin print_nl(""); print_ln; print("Completed␣box␣being␣shipped␣out");
    end;
  if term_offset > max_print_line − 9 then print_ln
  else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
  print_char("["); j ← 9;
  while (count(j) = 0) ∧ (j > 0) do decr(j);
  for k ← 0 to j do
    begin print_int(count(k));
    if k < j then print_char(".");
    end;
  update_terminal;
  if tracing_output > 0 then
    begin print_char("]"); begin_diagnostic; show_box(p); end_diagnostic(true);
    end;
  ⟨ Ship box p out 640 ⟩;
  if eTeX_ex then ⟨ Check for LR anomalies at the end of ship_out 1465* ⟩;
  if tracing_output ≤ 0 then print_char("]");
  dead_cycles ← 0; update_terminal;   { progress report }
  ⟨ Flush the box from memory, showing statistics if requested 639 ⟩;
  end;
```

649.* Here now is *hpack*, which contains few if any surprises.

```
function hpack(p : pointer; w : scaled; m : small_number): pointer;
  label reswitch, common_ending, exit;
  var r: pointer; { the box node that will be returned }
    q: pointer; { trails behind p }
    h, d, x: scaled; { height, depth, and natural width }
    s: scaled; { shift amount }
    g: pointer; { points to a glue specification }
    o: glue_ord; { order of infinity }
    f: internal_font_number; { the font in a char_node }
    i: four_quarters; { font information about a char_node }
    hd: eight_bits; { height and depth indices for a character }
  begin last_badness ← 0; r ← get_node(box_node_size); type(r) ← hlist_node;
  subtype(r) ← min_quarterword; shift_amount(r) ← 0; q ← r + list_offset; link(q) ← p;
  h ← 0; ⟨Clear dimensions to zero 650⟩;
  if TeXXeT_en then ⟨Initialize the LR stack 1441*⟩;
  while p ≠ null do ⟨Examine node p in the hlist, taking account of its effect on the dimensions of the
        new box, or moving it to the adjustment list; then advance p to the next node 651*⟩;
  if adjust_tail ≠ null then link(adjust_tail) ← null;
  height(r) ← h; depth(r) ← d;
  ⟨Determine the value of width(r) and the appropriate glue setting; then return or goto
      common_ending 657⟩;
common_ending: ⟨Finish issuing a diagnostic message for an overfull or underfull hbox 663⟩;
exit: if TeXXeT_en then ⟨Check for LR anomalies at the end of hpack 1443*⟩;
  hpack ← r;
  end;
```

651.* ⟨Examine node *p* in the hlist, taking account of its effect on the dimensions of the new box, or moving it to the adjustment list; then advance *p* to the next node 651*⟩ ≡

```
  begin reswitch: while is_char_node(p) do ⟨Incorporate character dimensions into the dimensions of the
        hbox that will contain it, then move to the next node 654⟩;
  if p ≠ null then
    begin case type(p) of
    hlist_node, vlist_node, rule_node, unset_node: ⟨Incorporate box dimensions into the dimensions of the
          hbox that will contain it 653⟩;
    ins_node, mark_node, adjust_node: if adjust_tail ≠ null then
        ⟨Transfer node p to the adjustment list 655⟩;
    whatsit_node: ⟨Incorporate a whatsit node into an hbox 1360⟩;
    glue_node: ⟨Incorporate glue into the horizontal totals 656⟩;
    kern_node: x ← x + width(p);
    math_node: begin x ← x + width(p);
      if TeXXeT_en then ⟨Adjust the LR stack for the hpack routine 1442*⟩;
      end;
    ligature_node: ⟨Make node p look like a char_node and goto reswitch 652⟩;
    othercases do_nothing
    endcases;
    p ← link(p);
    end;
  end
```

This code is used in section 649*.

687.* A few more kinds of noads will complete the set: An *under_noad* has its nucleus underlined; an *over_noad* has it overlined. An *accent_noad* places an accent over its nucleus; the accent character appears as *fam*(*accent_chr*(*p*)) and *character*(*accent_chr*(*p*)). A *vcenter_noad* centers its nucleus vertically with respect to the axis of the formula; in such noads we always have *math_type*(*nucleus*(*p*)) = *sub_box*.

And finally, we have *left_noad* and *right_noad* types, to implement TEX's `\left` and `\right` as well as ε-TEX's `\middle`. The *nucleus* of such noads is replaced by a *delimiter* field; thus, for example, '`\left(`' produces a *left_noad* such that *delimiter*(*p*) holds the family and character codes for all left parentheses. A *left_noad* never appears in an mlist except as the first element, and a *right_noad* never appears in an mlist except as the last element; furthermore, we either have both a *left_noad* and a *right_noad*, or neither one is present. The *subscr* and *supscr* fields are always *empty* in a *left_noad* and a *right_noad*.

```
define under_noad = fraction_noad + 1   { type of a noad for underlining }
define over_noad = under_noad + 1   { type of a noad for overlining }
define accent_noad = over_noad + 1   { type of a noad for accented subformulas }
define accent_noad_size = 5   { number of mem words in an accent noad }
define accent_chr(#) ≡ # + 4   { the accent_chr field of an accent noad }
define vcenter_noad = accent_noad + 1   { type of a noad for \vcenter }
define left_noad = vcenter_noad + 1   { type of a noad for \left }
define right_noad = left_noad + 1   { type of a noad for \right }
define delimiter ≡ nucleus   { delimiter field in left and right noads }
define middle_noad ≡ 1   { subtype of right noad representing \middle }
define scripts_allowed(#) ≡ (type(#) ≥ ord_noad) ∧ (type(#) < left_noad)
```

696*. ⟨Display normal noad p 696*⟩ ≡

```
begin case type(p) of
ord_noad: print_esc("mathord");
op_noad: print_esc("mathop");
bin_noad: print_esc("mathbin");
rel_noad: print_esc("mathrel");
open_noad: print_esc("mathopen");
close_noad: print_esc("mathclose");
punct_noad: print_esc("mathpunct");
inner_noad: print_esc("mathinner");
over_noad: print_esc("overline");
under_noad: print_esc("underline");
vcenter_noad: print_esc("vcenter");
radical_noad: begin print_esc("radical"); print_delimiter(left_delimiter(p));
  end;
accent_noad: begin print_esc("accent"); print_fam_and_char(accent_chr(p));
  end;
left_noad: begin print_esc("left"); print_delimiter(delimiter(p));
  end;
right_noad: begin if subtype(p) = normal then print_esc("right")
  else print_esc("middle");
  print_delimiter(delimiter(p));
  end;
end;
if type(p) < left_noad then
  begin if subtype(p) ≠ normal then
    if subtype(p) = limits then print_esc("limits")
    else print_esc("nolimits");
  print_subsidiary_data(nucleus(p), ".");
  end;
print_subsidiary_data(supscr(p), "^"); print_subsidiary_data(subscr(p), "_");
end
```

This code is used in section 690.

727* We use the fact that no character nodes appear in an mlist, hence the field *type*(*q*) is always present.

```
⟨ Process node-or-noad $q$ as much as possible in preparation for the second pass of mlist_to_hlist, then move
        to the next item in the mlist 727* ⟩ ≡
  begin ⟨ Do first-pass processing based on type(q); goto done_with_noad if a noad has been fully
        processed, goto check_dimensions if it has been translated into new_hlist(q), or goto done_with_node
        if a node has been fully processed 728 ⟩;
check_dimensions: $z \gets hpack(new\_hlist(q), natural)$;
  if $height(z) > max\_h$ then $max\_h \gets height(z)$;
  if $depth(z) > max\_d$ then $max\_d \gets depth(z)$;
  $free\_node(z, box\_node\_size)$;
done_with_noad: $r \gets q$; $r\_type \gets type(r)$;
  if $r\_type = right\_noad$ then
    begin $r\_type \gets left\_noad$; $cur\_style \gets style$;
    ⟨ Set up the values of cur_size and cur_mu, based on cur_style 703 ⟩;
    end;
done_with_node: $q \gets link(q)$;
  end
```

This code is used in section 726.

760* We have now tied up all the loose ends of the first pass of *mlist_to_hlist*. The second pass simply goes through and hooks everything together with the proper glue and penalties. It also handles the *left_noad* and *right_noad* that might be present, since *max_h* and *max_d* are now known. Variable *p* points to a node at the current end of the final hlist.

```
⟨ Make a second pass over the mlist, removing all noads and inserting the proper spacing and
        penalties 760* ⟩ ≡
  $p \gets temp\_head$; $link(p) \gets null$; $q \gets mlist$; $r\_type \gets 0$; $cur\_style \gets style$;
  ⟨ Set up the values of cur_size and cur_mu, based on cur_style 703 ⟩;
  while $q \neq null$ do
    begin ⟨ If node q is a style node, change the style and goto delete_q; otherwise if it is not a noad, put
          it into the hlist, advance q, and goto done; otherwise set s to the size of noad q, set t to the
          associated type (ord_noad .. inner_noad), and set pen to the associated penalty 761 ⟩;
    ⟨ Append inter-element spacing based on r_type and t 766 ⟩;
    ⟨ Append any new_hlist entries for q, and any appropriate penalties 767 ⟩;
    if $type(q) = right\_noad$ then $t \gets open\_noad$;
    $r\_type \gets t$;
  delete_q: $r \gets q$; $q \gets link(q)$; $free\_node(r, s)$;
  done: end
```

This code is used in section 726.

762* The *make_left_right* function constructs a left or right delimiter of the required size and returns the value *open_noad* or *close_noad*. The *right_noad* and *left_noad* will both be based on the original *style*, so they will have consistent sizes.

We use the fact that $right\_noad - left\_noad = close\_noad - open\_noad$.

```
⟨Declare math construction procedures 734⟩ +≡
function make_left_right(q : pointer; style : small_number; max_d, max_h : scaled): small_number;
  var delta, delta1, delta2: scaled;  { dimensions used in the calculation }
  begin cur_style ← style; ⟨Set up the values of cur_size and cur_mu, based on cur_style 703⟩;
  delta2 ← max_d + axis_height(cur_size); delta1 ← max_h + max_d − delta2;
  if delta2 > delta1 then delta1 ← delta2;  { delta1 is max distance from axis }
  delta ← (delta1 div 500) ∗ delimiter_factor; delta2 ← delta1 + delta1 − delimiter_shortfall;
  if delta < delta2 then delta ← delta2;
  new_hlist(q) ← var_delimiter(delimiter(q), cur_size, delta);
  make_left_right ← type(q) − (left_noad − open_noad);  { open_noad or close_noad }
  end;
```

785* The tricky part about alignments is getting the templates into the scanner at the right time, and recovering control when a row or column is finished.

We usually begin a row after each `\cr` has been sensed, unless that `\cr` is followed by `\noalign` or by the right brace that terminates the alignment. The *align_peek* routine is used to look ahead and do the right thing; it either gets a new row started, or gets a `\noalign` started, or finishes off the alignment.

```
⟨Declare the procedure called align_peek 785*⟩ ≡
procedure align_peek;
  label restart;
  begin restart: align_state ← 1000000;
  repeat get_x_or_protected;
  until cur_cmd ≠ spacer;
  if cur_cmd = no_align then
    begin scan_left_brace; new_save_level(no_align_group);
    if mode = −vmode then normal_paragraph;
    end
  else if cur_cmd = right_brace then fin_align
    else if (cur_cmd = car_ret) ∧ (cur_chr = cr_cr_code) then goto restart   { ignore \crcr }
      else begin init_row;   { start a new row }
        init_col;   { start a new column and replace what we peeked at }
        end;
  end;
```

This code is used in section 800.

791* When the *endv* command at the end of a $\langle v_j\rangle$ template comes through the scanner, things really start to happen; and it is the *fin_col* routine that makes them happen. This routine returns *true* if a row as well as a column has been finished.

```
function fin_col: boolean;
  label exit;
  var p: pointer;  { the alignrecord after the current one }
    q, r: pointer;  { temporary pointers for list manipulation }
    s: pointer;  { a new span node }
    u: pointer;  { a new unset box }
    w: scaled;  { natural width }
    o: glue_ord;  { order of infinity }
    n: halfword;  { span counter }
  begin if cur_align = null then confusion("endv");
  q ← link(cur_align); if q = null then confusion("endv");
  if align_state < 500000 then fatal_error("(interwoven␣alignment␣preambles␣are␣not␣allowed)");
  p ← link(q); ⟨ If the preamble list has been traversed, check that the row has ended 792 ⟩;
  if extra_info(cur_align) ≠ span_code then
    begin unsave; new_save_level(align_group);
    ⟨ Package an unset box for the current column and record its width 796 ⟩;
    ⟨ Copy the tabskip glue between columns 795 ⟩;
    if extra_info(cur_align) ≥ cr_code then
      begin fin_col ← true; return;
      end;
    init_span(p);
    end;
  align_state ← 1000000;
  repeat get_x_or_protected;
  until cur_cmd ≠ spacer;
  cur_align ← p; init_col; fin_col ← false;
exit: end;
```

807* The unset box *q* represents a row that contains one or more unset boxes, depending on how soon `\cr` occurred in that row.

```
⟨ Set the unset box q and the unset boxes in it 807* ⟩ ≡
  begin if mode = −vmode then
    begin type(q) ← hlist_node; width(q) ← width(p);
    if nest[nest_ptr − 1].mode_field = mmode then set_box_lr(q)(dlist);  { for ship_out }
    end
  else begin type(q) ← vlist_node; height(q) ← height(p);
    end;
  glue_order(q) ← glue_order(p); glue_sign(q) ← glue_sign(p); glue_set(q) ← glue_set(p);
  shift_amount(q) ← o; r ← link(list_ptr(q)); s ← link(list_ptr(p));
  repeat ⟨ Set the glue in node r and change it from an unset node 808* ⟩;
    r ← link(link(r)); s ← link(link(s));
  until r = null;
  end
```

This code is used in section 805.

808*. A box made from spanned columns will be followed by tabskip glue nodes and by empty boxes as if there were no spanning. This permits perfect alignment of subsequent entries, and it prevents values that depend on floating point arithmetic from entering into the dimensions of any boxes.

```
⟨ Set the glue in node r and change it from an unset node 808* ⟩ ≡
  $n \gets span\_count(r)$; $t \gets width(s)$; $w \gets t$; $u \gets hold\_head$; $set\_box\_lr(r)(0)$;   { for ship_out }
  while $n > min\_quarterword$ do
    begin $decr(n)$; ⟨ Append tabskip glue and an empty box to list $u$, and update $s$ and $t$ as the prototype
        nodes are passed 809 ⟩;
    end;
  if $mode = -vmode$ then
    ⟨ Make the unset node $r$ into an hlist_node of width $w$, setting the glue as if the width were $t$ 810 ⟩
  else ⟨ Make the unset node $r$ into a vlist_node of height $w$, setting the glue as if the height were $t$ 811 ⟩;
  $shift\_amount(r) \gets 0$;
  if $u \neq hold\_head$ then   { append blank boxes to account for spanned nodes }
    begin $link(u) \gets link(r)$; $link(r) \gets link(hold\_head)$; $r \gets u$;
    end
```

This code is used in section 807*.

814* The *line_break* procedure should be invoked only in horizontal mode; it leaves that mode and places its output into the current vlist of the enclosing vertical mode (or internal vertical mode). There is one explicit parameter: *d* is true for partial paragraphs preceding display math mode; in this case the amount of additional penalty inserted before the final line is *display_widow_penalty* instead of *widow_penalty*.

There are also a number of implicit parameters: The hlist to be broken starts at *link*(*head*), and it is nonempty. The value of *prev_graf* in the enclosing semantic level tells where the paragraph should begin in the sequence of line numbers, in case hanging indentation or `\parshape` is in use; *prev_graf* is zero unless this paragraph is being continued after a displayed formula. Other implicit parameters, such as the *par_shape_ptr* and various penalties to use for hyphenation, etc., appear in *eqtb*.

After *line_break* has acted, it will have updated the current vlist and the value of *prev_graf*. Furthermore, the global variable *just_box* will point to the final box created by *line_break*, so that the width of this line can be ascertained when it is necessary to decide whether to use *above_display_skip* or *above_display_short_skip* before a displayed formula.

```
⟨ Global variables 13 ⟩ +≡
just_box: pointer;  { the hlist_node for the last line of the new paragraph }
```

815* Since *line_break* is a rather lengthy procedure—sort of a small world unto itself—we must build it up little by little, somewhat more cautiously than we have done with the simpler procedures of TEX. Here is the general outline.

```
⟨ Declare subprocedures for line_break 826 ⟩
procedure line_break(d : boolean);
  label done, done1, done2, done3, done4, done5, continue;
  var ⟨ Local variables for line breaking 862 ⟩
  begin pack_begin_line ← mode_line;  { this is for over/underfull box messages }
  ⟨ Get ready to start line breaking 816* ⟩;
  ⟨ Find optimal breakpoints 863* ⟩;
  ⟨ Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and
      append them to the current vertical list 876* ⟩;
  ⟨ Clean up the memory by removing the break nodes 865 ⟩;
  pack_begin_line ← 0;
  end;
⟨ Declare ε-TEX procedures for use by main_control 1387* ⟩
```

816* The first task is to move the list from *head* to *temp_head* and go into the enclosing semantic level. We also append the `\parfillskip` glue to the end of the paragraph, removing a space (or other glue node) if it was there, since spaces usually precede blank lines and instances of '`$$`'. The *par_fill_skip* is preceded by an infinite penalty, so it will never be considered as a potential breakpoint.

This code assumes that a *glue_node* and a *penalty_node* occupy the same number of *mem* words.

```
⟨ Get ready to start line breaking 816* ⟩ ≡
  link(temp_head) ← link(head);
  if is_char_node(tail) then tail_append(new_penalty(inf_penalty))
  else if type(tail) ≠ glue_node then tail_append(new_penalty(inf_penalty))
    else begin type(tail) ← penalty_node; delete_glue_ref(glue_ptr(tail)); flush_node_list(leader_ptr(tail));
      penalty(tail) ← inf_penalty;
      end;
  link(tail) ← new_param_glue(par_fill_skip_code); last_line_fill ← link(tail);
  init_cur_lang ← prev_graf mod ´200000; init_l_hyf ← prev_graf div ´20000000;
  init_r_hyf ← (prev_graf div ´200000) mod ´100; pop_nest;
```

See also sections 827*, 834, and 848.

This code is used in section 815*.

819* An active node for a given breakpoint contains six fields:

link points to the next node in the list of active nodes; the last active node has *link* = *last_active*.

break_node points to the passive node associated with this breakpoint.

line_number is the number of the line that follows this breakpoint.

fitness is the fitness classification of the line ending at this breakpoint.

type is either *hyphenated* or *unhyphenated*, depending on whether this breakpoint is a *disc_node*.

total_demerits is the minimum possible sum of demerits over all lines leading from the beginning of the paragraph to this breakpoint.

The value of *link*(*active*) points to the first active node on a linked list of all currently active nodes. This list is in order by *line_number*, except that nodes with *line_number* > *easy_line* may be in any order relative to each other.

```
define active_node_size_normal = 3   { number of words in normal active nodes }
define fitness ≡ subtype   { very_loose_fit .. tight_fit on final line for this break }
define break_node ≡ rlink   { pointer to the corresponding passive node }
define line_number ≡ llink   { line that begins at this breakpoint }
define total_demerits(#) ≡ mem[# + 2].int   { the quantity that TeX minimizes }
define unhyphenated = 0   { the type of a normal active break node }
define hyphenated = 1   { the type of an active node that breaks at a disc_node }
define last_active ≡ active   { the active list ends where it begins }
```

827* ⟨ Get ready to start line breaking 816* ⟩ +≡

```
no_shrink_error_yet ← true;
check_shrinkage(left_skip); check_shrinkage(right_skip);
q ← left_skip; r ← right_skip; background[1] ← width(q) + width(r);
background[2] ← 0; background[3] ← 0; background[4] ← 0; background[5] ← 0;
background[2 + stretch_order(q)] ← stretch(q);
background[2 + stretch_order(r)] ← background[2 + stretch_order(r)] + stretch(r);
background[6] ← shrink(q) + shrink(r); ⟨ Check for special treatment of last line of paragraph 1578* ⟩;
```

829* The heart of the line-breaking procedure is '*try_break*', a subroutine that tests if the current breakpoint *cur_p* is feasible, by running through the active list to see what lines of text can be made from active nodes to *cur_p*. If feasible breaks are possible, new break nodes are created. If *cur_p* is too far from an active node, that node is deactivated.

The parameter *pi* to *try_break* is the penalty associated with a break at *cur_p*; we have $pi = eject\_penalty$ if the break is forced, and $pi = inf\_penalty$ if the break is illegal.

The other parameter, *break_type*, is set to *hyphenated* or *unhyphenated*, depending on whether or not the current break is at a *disc_node*. The end of a paragraph is also regarded as '*hyphenated*'; this case is distinguishable by the condition $cur\_p = null$.

```
  define copy_to_cur_active(#) ≡ cur_active_width[#] ← active_width[#]
  define deactivate = 60  { go here when node r should be deactivated }
⟨ Declare subprocedures for line_break 826 ⟩ +≡
procedure try_break(pi : integer; break_type : small_number);
  label exit, done, done1, continue, deactivate, found, not_found;
  var r: pointer;  { runs through the active list }
    prev_r: pointer;  { stays a step behind r }
    old_l: halfword;  { maximum line number in current equivalence class of lines }
    no_break_yet: boolean;  { have we found a feasible break at cur_p? }
    ⟨ Other local variables for try_break 830 ⟩
  begin ⟨ Make sure that pi is in the proper range 831 ⟩;
  no_break_yet ← true; prev_r ← active; old_l ← 0; do_all_six(copy_to_cur_active);
  loop begin continue: r ← link(prev_r); ⟨ If node r is of type delta_node, update cur_active_width, set
        prev_r and prev_prev_r, then goto continue 832 ⟩;
    ⟨ If a line number class has ended, create new active nodes for the best feasible breaks in that class;
        then return if r = last_active, otherwise compute the new line_width 835 ⟩;
    ⟨ Consider the demerits for a line from r to cur_p; deactivate node r if it should no longer be active;
        then goto continue if a line from r to cur_p is infeasible, otherwise record a new feasible
        break 851* ⟩;
    end;
exit: stat ⟨ Update the value of printed_node for symbolic displays 858 ⟩ tats
  end;
```

845* When we create an active node, we also create the corresponding passive node.

```
⟨ Insert a new active node from best_place[fit_class] to cur_p 845* ⟩ ≡
  begin q ← get_node(passive_node_size); link(q) ← passive; passive ← q; cur_break(q) ← cur_p;
  stat incr(pass_number); serial(q) ← pass_number; tats
  prev_break(q) ← best_place[fit_class];
  q ← get_node(active_node_size); break_node(q) ← passive; line_number(q) ← best_pl_line[fit_class] + 1;
  fitness(q) ← fit_class; type(q) ← break_type; total_demerits(q) ← minimal_demerits[fit_class];
  if do_last_line_fit then ⟨ Store additional data in the new active node 1586* ⟩;
  link(q) ← r; link(prev_r) ← q; prev_r ← q;
  stat if tracing_paragraphs > 0 then ⟨ Print a symbolic description of the new break node 846* ⟩;
  tats
  end
```

This code is used in section 836.

846* ⟨ Print a symbolic description of the new break node 846* ⟩ ≡

```
begin print_nl("@@"); print_int(serial(passive)); print(":␣line␣"); print_int(line_number(q) − 1);
print_char("."); print_int(fit_class);
if break_type = hyphenated then print_char("-");
print("␣t="); print_int(total_demerits(q));
if do_last_line_fit then ⟨ Print additional data in the new active node 1587* ⟩;
print("␣->␣@@");
if prev_break(passive) = null then print_char("0")
else print_int(serial(prev_break(passive)));
end
```

This code is used in section 845*.

851* The remaining part of *try_break* deals with the calculation of demerits for a break from r to *cur_p*.

The first thing to do is calculate the badness, b. This value will always be between zero and *inf_bad* + 1; the latter value occurs only in the case of lines from r to *cur_p* that cannot shrink enough to fit the necessary width. In such cases, node r will be deactivated. We also deactivate node r when a break at *cur_p* is forced, since future breaks must go through a forced break.

⟨ Consider the demerits for a line from r to *cur_p*; deactivate node r if it should no longer be active; then **goto** *continue* if a line from r to *cur_p* is infeasible, otherwise record a new feasible break 851* ⟩ ≡

```
   begin artificial_demerits ← false;
   shortfall ← line_width − cur_active_width[1]; { we're this much too short }
   if shortfall > 0 then
      ⟨ Set the value of b to the badness for stretching the line, and compute the corresponding fit_class 852* ⟩
   else ⟨ Set the value of b to the badness for shrinking the line, and compute the corresponding fit_class 853 ⟩;
   if do_last_line_fit then ⟨ Adjust the additional data for last line 1584* ⟩;
found: if (b > inf_bad) ∨ (pi = eject_penalty) then ⟨ Prepare to deactivate node r, and goto deactivate
            unless there is a reason to consider lines of text from r to cur_p 854 ⟩
   else begin prev_r ← r;
      if b > threshold then goto continue;
      node_r_stays_active ← true;
      end;
   ⟨ Record a new feasible break 855* ⟩;
   if node_r_stays_active then goto continue; { prev_r has been set to r }
deactivate: ⟨ Deactivate node r 860 ⟩;
   end
```

This code is used in section 829*.

852* When a line must stretch, the available stretchability can be found in the subarray *cur_active_width*[2 . . 5], in units of points, fil, fill, and filll.

The present section is part of TEX's inner loop, and it is most often performed when the badness is infinite; therefore it is worth while to make a quick test for large width excess and small stretchability, before calling the *badness* subroutine.

```
⟨ Set the value of $b$ to the badness for stretching the line, and compute the corresponding fit_class 852* ⟩ ≡
  if ($cur\_active\_width[3] \neq 0$) ∨ ($cur\_active\_width[4] \neq 0$) ∨ ($cur\_active\_width[5] \neq 0$) then
    begin if do_last_line_fit then
      begin if $cur\_p = null$ then   { the last line of a paragraph }
        ⟨ Perform computations for last line and goto found 1581* ⟩;
      $shortfall \gets 0$;
      end;
    $b \gets 0$; $fit\_class \gets decent\_fit$;   { infinite stretch }
    end
  else begin if $shortfall > 7230584$ then
      if $cur\_active\_width[2] < 1663497$ then
        begin $b \gets inf\_bad$; $fit\_class \gets very\_loose\_fit$; goto done1;
        end;
    $b \gets badness(shortfall, cur\_active\_width[2])$;
    if $b > 12$ then
      if $b > 99$ then $fit\_class \gets very\_loose\_fit$
      else $fit\_class \gets loose\_fit$
    else $fit\_class \gets decent\_fit$;
  done1: end
```

This code is used in section 851*.

855* When we get to this part of the code, the line from r to *cur_p* is feasible, its badness is b, and its fitness classification is *fit_class*. We don't want to make an active node for this break yet, but we will compute the total demerits and record them in the *minimal_demerits* array, if such a break is the current champion among all ways to get to *cur_p* in a given line-number class and fitness class.

```
⟨ Record a new feasible break 855* ⟩ ≡
  if artificial_demerits then $d \gets 0$
  else ⟨ Compute the demerits, $d$, from $r$ to cur_p 859 ⟩;
  stat if $tracing\_paragraphs > 0$ then ⟨ Print a symbolic description of this feasible break 856 ⟩;
  tats
  $d \gets d + total\_demerits(r)$;   { this is the minimum total demerits from the beginning to cur_p via $r$ }
  if $d \leq minimal\_demerits[fit\_class]$ then
    begin $minimal\_demerits[fit\_class] \gets d$; $best\_place[fit\_class] \gets break\_node(r)$; $best\_pl\_line[fit\_class] \gets l$;
    if do_last_line_fit then ⟨ Store additional data for this feasible break 1585* ⟩;
    if $d < minimum\_demerits$ then $minimum\_demerits \gets d$;
    end
```

This code is used in section 851*.

863* The '**loop**' in the following code is performed at most thrice per call of *line_break*, since it is actually a pass over the entire paragraph.

```
⟨ Find optimal breakpoints 863* ⟩ ≡
  threshold ← pretolerance;
  if threshold ≥ 0 then
    begin stat if tracing_paragraphs > 0 then
      begin begin_diagnostic; print_nl("@firstpass"); end; tats
    second_pass ← false; final_pass ← false;
    end
  else begin threshold ← tolerance; second_pass ← true; final_pass ← (emergency_stretch ≤ 0);
    stat if tracing_paragraphs > 0 then begin_diagnostic;
    tats
    end;
  loop begin if threshold > inf_bad then threshold ← inf_bad;
    if second_pass then ⟨ Initialize for hyphenating a paragraph 891* ⟩;
    ⟨ Create an active breakpoint representing the beginning of the paragraph 864* ⟩;
    cur_p ← link(temp_head); auto_breaking ← true;
    prev_p ← cur_p;  { glue at beginning is not a legal breakpoint }
    while (cur_p ≠ null) ∧ (link(active) ≠ last_active) do ⟨ Call try_break if cur_p is a legal breakpoint;
          on the second pass, also try to hyphenate the next word, if cur_p is a glue node; then advance
          cur_p to the next node of the paragraph that could possibly be a legal breakpoint 866* ⟩;
    if cur_p = null then ⟨ Try the final line break at the end of the paragraph, and goto done if the
          desired breakpoints have been found 873 ⟩;
    ⟨ Clean up the memory by removing the break nodes 865 ⟩;
    if ¬second_pass then
      begin stat if tracing_paragraphs > 0 then print_nl("@secondpass"); tats
      threshold ← tolerance; second_pass ← true; final_pass ← (emergency_stretch ≤ 0);
      end  { if at first you don't succeed, . . . }
    else begin stat if tracing_paragraphs > 0 then print_nl("@emergencypass"); tats
      background[2] ← background[2] + emergency_stretch; final_pass ← true;
      end;
    end;
done: stat if tracing_paragraphs > 0 then
    begin end_diagnostic(true); normalize_selector;
    end;
  tats
  if do_last_line_fit then ⟨ Adjust the final line of the paragraph 1588* ⟩;
```

This code is used in section 815*.

864* The active node that represents the starting point does not need a corresponding passive node.

```
  define store_background(#) ≡ active_width[#] ← background[#]

⟨ Create an active breakpoint representing the beginning of the paragraph 864* ⟩ ≡
  q ← get_node(active_node_size); type(q) ← unhyphenated; fitness(q) ← decent_fit; link(q) ← last_active;
  break_node(q) ← null; line_number(q) ← prev_graf + 1; total_demerits(q) ← 0; link(active) ← q;
  if do_last_line_fit then ⟨ Initialize additional fields of the first active node 1580* ⟩;
  do_all_six(store_background);
  passive ← null; printed_node ← temp_head; pass_number ← 0; font_in_short_display ← null_font
```

This code is used in section 863*.

866* Here is the main switch in the *line_break* routine, where legal breaks are determined. As we move through the hlist, we need to keep the *active_width* array up to date, so that the badness of individual lines is readily calculated by *try_break*. It is convenient to use the short name *act_width* for the component of active width that represents real width as opposed to glue.

```
define act_width ≡ active_width[1]   { length from first active node to current node }
define kern_break ≡
            begin if ¬is_char_node(link(cur_p)) ∧ auto_breaking then
              if type(link(cur_p)) = glue_node then try_break(0, unhyphenated);
            act_width ← act_width + width(cur_p);
            end
```

⟨ Call *try_break* if *cur_p* is a legal breakpoint; on the second pass, also try to hyphenate the next word, if *cur_p* is a glue node; then advance *cur_p* to the next node of the paragraph that could possibly be a legal breakpoint 866* ⟩ ≡

```
  begin if is_char_node(cur_p) then
    ⟨ Advance cur_p to the node following the present string of characters 867 ⟩;
  case type(cur_p) of
  hlist_node, vlist_node, rule_node: act_width ← act_width + width(cur_p);
  whatsit_node: ⟨ Advance past a whatsit node in the line_break loop 1362* ⟩;
  glue_node: begin ⟨ If node cur_p is a legal breakpoint, call try_break; then update the active widths by
          including the glue in glue_ptr(cur_p) 868 ⟩;
    if second_pass ∧ auto_breaking then ⟨ Try to hyphenate the following word 894 ⟩;
    end;
  kern_node: if subtype(cur_p) = explicit then kern_break
    else act_width ← act_width + width(cur_p);
  ligature_node: begin f ← font(lig_char(cur_p));
    act_width ← act_width + char_width(f)(char_info(f)(character(lig_char(cur_p))));
    end;
  disc_node: ⟨ Try to break after a discretionary fragment, then goto done5 869 ⟩;
  math_node: begin if subtype(cur_p) < L_code then auto_breaking ← odd(subtype(cur_p));
    kern_break;
    end;
  penalty_node: try_break(penalty(cur_p), unhyphenated);
  mark_node, ins_node, adjust_node: do_nothing;
  othercases confusion("paragraph")
  endcases;
  prev_p ← cur_p; cur_p ← link(cur_p);
done5: end
```

This code is used in section 863*.

876* Once the best sequence of breakpoints has been found (hurray), we call on the procedure *post_line_break* to finish the remainder of the work. (By introducing this subprocedure, we are able to keep *line_break* from getting extremely long.)

⟨ Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list 876* ⟩ ≡

```
  post_line_break(d)
```

This code is used in section 815*.

877*. The total number of lines that will be set by *post_line_break* is *best_line* − *prev_graf* − 1. The last breakpoint is specified by *break_node*(*best_bet*), and this passive node points to the other breakpoints via the *prev_break* links. The finishing-up phase starts by linking the relevant passive nodes in forward order, changing *prev_break* to *next_break*. (The *next_break* fields actually reside in the same memory space as the *prev_break* fields did, but we give them a new name because of their new significance.) Then the lines are justified, one by one.

```
  define next_break ≡ prev_break  { new name for prev_break after links are reversed }
⟨ Declare subprocedures for line_break 826 ⟩ +≡
procedure post_line_break(d : boolean);
  label done, done1;
  var q, r, s: pointer;  { temporary registers for list manipulation }
    disc_break: boolean;  { was the current break at a discretionary node? }
    post_disc_break: boolean;  { and did it have a nonempty post-break part? }
    cur_width: scaled;  { width of line number cur_line }
    cur_indent: scaled;  { left margin of line number cur_line }
    t: quarterword;  { used for replacement counts in discretionary nodes }
    pen: integer;  { use when calculating penalties between lines }
    cur_line: halfword;  { the current line number being justified }
    LR_ptr: pointer;  { stack of LR codes }
  begin LR_ptr ← LR_save;
  ⟨ Reverse the links of the relevant passive nodes, setting cur_p to the first breakpoint 878 ⟩;
  cur_line ← prev_graf + 1;
  repeat ⟨ Justify the line ending at breakpoint cur_p, and append it to the current vertical list, together
        with associated penalties and other insertions 880* ⟩;
    incr(cur_line); cur_p ← next_break(cur_p);
    if cur_p ≠ null then
      if ¬post_disc_break then ⟨ Prune unwanted nodes at the beginning of the next line 879* ⟩;
  until cur_p = null;
  if (cur_line ≠ best_line) ∨ (link(temp_head) ≠ null) then confusion("line␣breaking");
  prev_graf ← best_line − 1; LR_save ← LR_ptr;
  end;
```

879*. Glue and penalty and kern and math nodes are deleted at the beginning of a line, except in the anomalous case that the node to be deleted is actually one of the chosen breakpoints. Otherwise the pruning done here is designed to match the lookahead computation in *try_break*, where the *break_width* values are computed for non-discretionary breakpoints.

```
⟨Prune unwanted nodes at the beginning of the next line 879*⟩ ≡
  begin r ← temp_head;
  loop begin q ← link(r);
    if q = cur_break(cur_p) then goto done1; { cur_break(cur_p) is the next breakpoint }
        { now q cannot be null }
    if is_char_node(q) then goto done1;
    if non_discardable(q) then goto done1;
    if type(q) = kern_node then
      if subtype(q) ≠ explicit then goto done1;
    r ← q; { now type(q) = glue_node, kern_node, math_node or penalty_node }
    if type(q) = math_node then
      if TeXXeT_en then ⟨Adjust the LR stack for the post_line_break routine 1439*⟩;
    end;
done1: if r ≠ temp_head then
    begin link(r) ← null; flush_node_list(link(temp_head)); link(temp_head) ← q;
    end;
  end
```

This code is used in section 877*.

880*. The current line to be justified appears in a horizontal list starting at *link*(*temp_head*) and ending at *cur_break*(*cur_p*). If *cur_break*(*cur_p*) is a glue node, we reset the glue to equal the *right_skip* glue; otherwise we append the *right_skip* glue at the right. If *cur_break*(*cur_p*) is a discretionary node, we modify the list so that the discretionary break is compulsory, and we set *disc_break* to *true*. We also append the *left_skip* glue at the left of the line, unless it is zero.

```
⟨Justify the line ending at breakpoint cur_p, and append it to the current vertical list, together with
      associated penalties and other insertions 880*⟩ ≡
  if TeXXeT_en then ⟨Insert LR nodes at the beginning of the current line and adjust the LR stack based
        on LR nodes in this line 1438*⟩;
  ⟨Modify the end of the line to reflect the nature of the break and to include \rightskip; also set the
      proper value of disc_break 881*⟩;
  if TeXXeT_en then ⟨Insert LR nodes at the end of the current line 1440*⟩;
  ⟨Put the \leftskip glue at the left and detach this line 887⟩;
  ⟨Call the packaging subroutine, setting just_box to the justified box 889⟩;
  ⟨Append the new box to the current vertical list, followed by the list of special nodes taken out of the
      box by the packager 888⟩;
  ⟨Append a penalty node, if a nonzero penalty is appropriate 890*⟩
```

This code is used in section 877*.

881* At the end of the following code, q will point to the final node on the list about to be justified.

```
⟨Modify the end of the line to reflect the nature of the break and to include \rightskip; also set the
        proper value of disc_break 881*⟩ ≡
  q ← cur_break(cur_p); disc_break ← false; post_disc_break ← false;
  if q ≠ null then   {q cannot be a char_node}
    if type(q) = glue_node then
      begin delete_glue_ref(glue_ptr(q)); glue_ptr(q) ← right_skip; subtype(q) ← right_skip_code + 1;
      add_glue_ref(right_skip); goto done;
      end
    else begin if type(q) = disc_node then
        ⟨Change discretionary to compulsory and set disc_break ← true 882⟩
      else if type(q) = kern_node then width(q) ← 0
        else if type(q) = math_node then
            begin width(q) ← 0;
            if TeXXeT_en then ⟨Adjust the LR stack for the post_line_break routine 1439*⟩;
            end;
      end
  else begin q ← temp_head;
    while link(q) ≠ null do q ← link(q);
    end;
  ⟨Put the \rightskip glue after node q 886⟩;
done:
```

This code is used in section 880*.

890* Penalties between the lines of a paragraph come from club and widow lines, from the *inter_line_penalty* parameter, and from lines that end at discretionary breaks. Breaking between lines of a two-line paragraph gets both club-line and widow-line penalties. The local variable *pen* will be set to the sum of all relevant penalties for the current line, except that the final line is never penalized.

```
⟨ Append a penalty node, if a nonzero penalty is appropriate 890* ⟩ ≡
  if $cur\_line + 1 \neq best\_line$ then
    begin $q \gets inter\_line\_penalties\_ptr$;
    if $q \neq null$ then
      begin $r \gets cur\_line$;
      if $r > penalty(q)$ then $r \gets penalty(q)$;
      $pen \gets penalty(q + r)$;
      end
    else $pen \gets inter\_line\_penalty$;
    $q \gets club\_penalties\_ptr$;
    if $q \neq null$ then
      begin $r \gets cur\_line - prev\_graf$;
      if $r > penalty(q)$ then $r \gets penalty(q)$;
      $pen \gets pen + penalty(q + r)$;
      end
    else if $cur\_line = prev\_graf + 1$ then $pen \gets pen + club\_penalty$;
    if $d$ then $q \gets display\_widow\_penalties\_ptr$
    else $q \gets widow\_penalties\_ptr$;
    if $q \neq null$ then
      begin $r \gets best\_line - cur\_line - 1$;
      if $r > penalty(q)$ then $r \gets penalty(q)$;
      $pen \gets pen + penalty(q + r)$;
      end
    else if $cur\_line + 2 = best\_line$ then
        if $d$ then $pen \gets pen + display\_widow\_penalty$
        else $pen \gets pen + widow\_penalty$;
    if $disc\_break$ then $pen \gets pen + broken\_penalty$;
    if $pen \neq 0$ then
      begin $r \gets new\_penalty(pen)$; $link(tail) \gets r$; $tail \gets r$;
      end;
    end
```

This code is used in section 880*.

891*. Pre-hyphenation. When the line-breaking routine is unable to find a feasible sequence of breakpoints, it makes a second pass over the paragraph, attempting to hyphenate the hyphenatable words. The goal of hyphenation is to insert discretionary material into the paragraph so that there are more potential places to break.

The general rules for hyphenation are somewhat complex and technical, because we want to be able to hyphenate words that are preceded or followed by punctuation marks, and because we want the rules to work for languages other than English. We also must contend with the fact that hyphens might radically alter the ligature and kerning structure of a word.

A sequence of characters will be considered for hyphenation only if it belongs to a "potentially hyphenatable part" of the current paragraph. This is a sequence of nodes $p_0p_1\ldots p_m$ where p_0 is a glue node, $p_1\ldots p_{m-1}$ are either character or ligature or whatsit or implicit kern or text direction nodes, and p_m is a glue or penalty or insertion or adjust or mark or whatsit or explicit kern node. (Therefore hyphenation is disabled by boxes, math formulas, and discretionary nodes already inserted by the user.) The ligature nodes among $p_1\ldots p_{m-1}$ are effectively expanded into the original non-ligature characters; the kern nodes and whatsits are ignored. Each character c is now classified as either a nonletter (if $lc\_code(c)=0$), a lowercase letter (if $lc\_code(c)=c$), or an uppercase letter (otherwise); an uppercase letter is treated as if it were $lc\_code(c)$ for purposes of hyphenation. The characters generated by $p_1\ldots p_{m-1}$ may begin with nonletters; let c_1 be the first letter that is not in the middle of a ligature. Whatsit nodes preceding c_1 are ignored; a whatsit found after c_1 will be the terminating node p_m. All characters that do not have the same font as c_1 will be treated as nonletters. The *hyphen_char* for that font must be between 0 and 255, otherwise hyphenation will not be attempted. TEX looks ahead for as many consecutive letters $c_1\ldots c_n$ as possible; however, n must be less than 64, so a character that would otherwise be c_{64} is effectively not a letter. Furthermore c_n must not be in the middle of a ligature. In this way we obtain a string of letters $c_1\ldots c_n$ that are generated by nodes $p_a\ldots p_b$, where $1\le a\le b+1\le m$. If $n\ge l\_hyf+r\_hyf$, this string qualifies for hyphenation; however, *uc_hyph* must be positive, if c_1 is uppercase.

The hyphenation process takes place in three stages. First, the candidate sequence $c_1\ldots c_n$ is found; then potential positions for hyphens are determined by referring to hyphenation tables; and finally, the nodes $p_a\ldots p_b$ are replaced by a new sequence of nodes that includes the discretionary breaks found.

Fortunately, we do not have to do all this calculation very often, because of the way it has been taken out of TEX's inner loop. For example, when the second edition of the author's 700-page book *Seminumerical Algorithms* was typeset by TEX, only about 1.2 hyphenations needed to be tried per paragraph, since the line breaking algorithm needed to use two passes on only about 5 per cent of the paragraphs.

```
⟨Initialize for hyphenating a paragraph 891*⟩ ≡
  begin init if trie_not_ready then init_trie;
  tini
  cur_lang ← init_cur_lang; l_hyf ← init_l_hyf; r_hyf ← init_r_hyf; set_hyph_index;
  end
```

This code is used in section 863*.

896*. The first thing we need to do is find the node *ha* just before the first letter.

⟨ Skip to node *ha*, or **goto** *done1* if no hyphenation should be attempted 896* ⟩ ≡

```
  loop begin if is_char_node(s) then
        begin c ← qo(character(s)); hf ← font(s);
        end
     else if type(s) = ligature_node then
           if lig_ptr(s) = null then goto continue
           else begin q ← lig_ptr(s); c ← qo(character(q)); hf ← font(q);
              end
        else if (type(s) = kern_node) ∧ (subtype(s) = normal) then goto continue
           else if (type(s) = math_node) ∧ (subtype(s) ≥ L_code) then goto continue
              else if type(s) = whatsit_node then
                    begin ⟨ Advance past a whatsit node in the pre-hyphenation loop 1363 ⟩;
                    goto continue;
                    end
                 else goto done1;
     set_lc_code(c);
     if hc[0] ≠ 0 then
        if (hc[0] = c) ∨ (uc_hyph > 0) then goto done2
        else goto done1;
  continue: prev_s ← s; s ← link(prev_s);
     end;
done2: hyf_char ← hyphen_char[hf];
  if hyf_char < 0 then goto done1;
  if hyf_char > 255 then goto done1;
  ha ← prev_s
```

This code is used in section 894.

897*. The word to be hyphenated is now moved to the *hu* and *hc* arrays.

⟨ Skip to node *hb*, putting letters into *hu* and *hc* 897* ⟩ ≡

```
  hn ← 0;
  loop begin if is_char_node(s) then
        begin if font(s) ≠ hf then goto done3;
        hyf_bchar ← character(s); c ← qo(hyf_bchar); set_lc_code(c);
        if hc[0] = 0 then goto done3;
        if hn = 63 then goto done3;
        hb ← s; incr(hn); hu[hn] ← c; hc[hn] ← hc[0]; hyf_bchar ← non_char;
        end
     else if type(s) = ligature_node then ⟨ Move the characters of a ligature node to hu and hc; but goto
              done3 if they are not all letters 898* ⟩
        else if (type(s) = kern_node) ∧ (subtype(s) = normal) then
              begin hb ← s; hyf_bchar ← font_bchar[hf];
              end
           else goto done3;
     s ← link(s);
     end;
done3:
```

This code is used in section 894.

898* We let j be the index of the character being stored when a ligature node is being expanded, since we do not want to advance *hn* until we are sure that the entire ligature consists of letters. Note that it is possible to get to *done3* with $hn = 0$ and *hb* not set to any value.

```
⟨Move the characters of a ligature node to hu and hc; but goto done3 if they are not all letters 898*⟩ ≡
  begin if $font(lig\_char(s)) \neq hf$ then goto done3;
  $j \leftarrow hn$; $q \leftarrow lig\_ptr(s)$; if $q > null$ then $hyf\_bchar \leftarrow character(q)$;
  while $q > null$ do
    begin $c \leftarrow qo(character(q))$; $set\_lc\_code(c)$;
    if $hc[0] = 0$ then goto done3;
    if $j = 63$ then goto done3;
    $incr(j)$; $hu[j] \leftarrow c$; $hc[j] \leftarrow hc[0]$;
    $q \leftarrow link(q)$;
    end;
  $hb \leftarrow s$; $hn \leftarrow j$;
  if $odd(subtype(s))$ then $hyf\_bchar \leftarrow font\_bchar[hf]$ else $hyf\_bchar \leftarrow non\_char$;
  end
```

This code is used in section 897*.

899* ⟨Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto** *done1* 899*⟩ ≡

```
  if $hn < l\_hyf + r\_hyf$ then goto done1; { l_hyf and r_hyf are $\geq 1$ }
  loop begin if $\neg(is\_char\_node(s))$ then
      case $type(s)$ of
      ligature_node: do_nothing;
      kern_node: if $subtype(s) \neq normal$ then goto done4;
      whatsit_node, glue_node, penalty_node, ins_node, adjust_node, mark_node: goto done4;
      math_node: if $subtype(s) \geq L\_code$ then goto done4 else goto done1;
      othercases goto done1
      endcases;
    $s \leftarrow link(s)$;
    end;
done4:
```

This code is used in section 894.

934* We have now completed the hyphenation routine, so the *line_break* procedure is finished at last. Since the hyphenation exception table is fresh in our minds, it's a good time to deal with the routine that adds new entries to it.

When TEX has scanned '`\hyphenation`', it calls on a procedure named *new_hyph_exceptions* to do the right thing.

```
  define set_cur_lang ≡
            if language ≤ 0 then cur_lang ← 0
            else if language > 255 then cur_lang ← 0
              else cur_lang ← language
procedure new_hyph_exceptions; { enters new exceptions }
  label reswitch, exit, found, not_found, not_found1;
  var n: 0 .. 64; { length of current word; not always a small_number }
    j: 0 .. 64; { an index into hc }
    h: hyph_pointer; { an index into hyph_word and hyph_list }
    k: str_number; { an index into str_start }
    p: pointer; { head of a list of hyphen positions }
    q: pointer; { used when creating a new node for list p }
    s, t: str_number; { strings being compared or stored }
    u, v: pool_pointer; { indices into str_pool }
  begin scan_left_brace; { a left brace must follow \hyphenation }
  set_cur_lang;
  init if trie_not_ready then
    begin hyph_index ← 0; goto not_found1;
    end;
  tini
  set_hyph_index;
not_found1: ⟨Enter as many hyphenation exceptions as are listed, until coming to a right brace; then
      return 935⟩;
exit: end;
```

937* ⟨Append a new letter or hyphen 937*⟩ ≡

```
  if cur_chr = "-" then ⟨Append the value n to list p 938⟩
  else begin set_lc_code(cur_chr);
    if hc[0] = 0 then
      begin print_err("Not␣a␣letter");
      help2("Letters␣in␣\hyphenation␣words␣must␣have␣\lccode>0.")
      ("Proceed;␣I´ll␣ignore␣the␣character␣I␣just␣read."); error;
      end
    else if n < 63 then
        begin incr(n); hc[n] ← hc[0];
        end;
    end
```

This code is used in section 935.

952* Here is how the trie-compression data structures are initialized. If storage is tight, it would be possible to overlap *trie_op_hash*, *trie_op_lang*, and *trie_op_val* with *trie*, *trie_hash*, and *trie_taken*, because we finish with the former just before we need the latter.

```
⟨ Get ready to compress the trie 952* ⟩ ≡
  ⟨ Sort the hyphenation op tables into proper order 945 ⟩;
  for $p \gets 0$ to $trie\_size$ do $trie\_hash[p] \gets 0$;
  $hyph\_root \gets compress\_trie(hyph\_root)$; $trie\_root \gets compress\_trie(trie\_root)$;
      { identify equivalent subtries }
  for $p \gets 0$ to $trie\_ptr$ do $trie\_ref[p] \gets 0$;
  for $p \gets 0$ to 255 do $trie\_min[p] \gets p + 1$;
  $trie\_link(0) \gets 1$; $trie\_max \gets 0$
```

This code is used in section 966*.

958* When the whole trie has been allocated into the sequential table, we must go through it once again so that *trie* contains the correct information. Null pointers in the linked trie will be represented by the value 0, which properly implements an "empty" family.

```
⟨ Move the data into trie 958* ⟩ ≡
  $h.rh \gets 0$; $h.b0 \gets min\_quarterword$; $h.b1 \gets min\_quarterword$;
      { $trie\_link \gets 0$, $trie\_op \gets min\_quarterword$, $trie\_char \gets qi(0)$ }
  if $trie\_max = 0$ then   { no patterns were given }
    begin for $r \gets 0$ to 256 do $trie[r] \gets h$;
    $trie\_max \gets 256$;
    end
  else begin if $hyph\_root > 0$ then $trie\_fix(hyph\_root)$;
    if $trie\_root > 0$ then $trie\_fix(trie\_root)$;   { this fixes the non-holes in trie }
    $r \gets 0$;   { now we will zero out all the holes }
    repeat $s \gets trie\_link(r)$; $trie[r] \gets h$; $r \gets s$;
    until $r > trie\_max$;
    end;
  $trie\_char(0) \gets qi("?")$;   { make $trie\_char(c) \neq c$ for all $c$ }
```

This code is used in section 966*.

960* Now let's go back to the easier problem, of building the linked trie. When `INITEX` has scanned the '`\patterns`' control sequence, it calls on *new_patterns* to do the right thing.

```
⟨ Declare procedures for preprocessing hyphenation patterns 944 ⟩ +≡
procedure new_patterns; { initializes the hyphenation pattern data }
  label done, done1;
  var k, l: 0 . . 64; { indices into hc and hyf; not always in small_number range }
    digit_sensed: boolean; { should the next digit be treated as a letter? }
    v: quarterword; { trie op code }
    p, q: trie_pointer; { nodes of trie traversed during insertion }
    first_child: boolean; { is p = trie_l[q]? }
    c: ASCII_code; { character being inserted }
  begin if trie_not_ready then
    begin set_cur_lang; scan_left_brace; { a left brace must follow \patterns }
    ⟨ Enter all of the patterns into a linked trie, until coming to a right brace 961 ⟩;
    if saving_hyph_codes > 0 then ⟨ Store hyphenation codes for current language 1590* ⟩;
    end
  else begin print_err("Too␣late␣for␣"); print_esc("patterns");
    help1("All␣patterns␣must␣be␣given␣before␣typesetting␣begins."); error;
    link(garbage) ← scan_toks(false, false); flush_list(def_ref);
    end;
  end;
```

966* Finally we put everything together: Here is how the trie gets to its final, efficient form. The following packing routine is rigged so that the root of the linked tree gets mapped into location 1 of *trie*, as required by the hyphenation algorithm. This happens because the first call of *first_fit* will "take" location 1.

```
⟨ Declare procedures for preprocessing hyphenation patterns 944 ⟩ +≡
procedure init_trie;
  var p: trie_pointer; { pointer for initialization }
    j, k, t: integer; { all-purpose registers for initialization }
    r, s: trie_pointer; { used to clean up the packed trie }
    h: two_halves; { template used to zero out trie's holes }
  begin ⟨ Get ready to compress the trie 952* ⟩;
  if trie_root ≠ 0 then
    begin first_fit(trie_root); trie_pack(trie_root);
    end;
  if hyph_root ≠ 0 then ⟨ Pack all stored hyph_codes 1592* ⟩;
  ⟨ Move the data into trie 958* ⟩;
  trie_not_ready ← false;
  end;
```

968*. A subroutine called *prune_page_top* takes a pointer to a vlist and returns a pointer to a modified vlist in which all glue, kern, and penalty nodes have been deleted before the first box or rule node. However, the first box or rule is actually preceded by a newly created glue node designed so that the topmost baseline will be at distance *split_top_skip* from the top, whenever this is possible without backspacing.

When the second argument s is *false* the deleted nodes are destroyed, otherwise they are collected in a list starting at *split_disc*.

In this routine and those that follow, we make use of the fact that a vertical list contains no character nodes, hence the *type* field exists for each node in the list.

```
function prune_page_top(p : pointer; s : boolean): pointer;   { adjust top after page break }
  var prev_p: pointer;   { lags one step behind p }
    q, r: pointer;   { temporary variables for list manipulation }
  begin prev_p ← temp_head; link(temp_head) ← p;
  while p ≠ null do
    case type(p) of
    hlist_node, vlist_node, rule_node: ⟨Insert glue for split_top_skip and set p ← null 969⟩;
    whatsit_node, mark_node, ins_node: begin prev_p ← p; p ← link(prev_p);
      end;
    glue_node, kern_node, penalty_node: begin q ← p; p ← link(q); link(q) ← null; link(prev_p) ← p;
      if s then
        begin if split_disc = null then split_disc ← q else link(r) ← q;
        r ← q;
        end
      else flush_node_list(q);
      end;
    othercases confusion("pruning")
    endcases;
  prune_page_top ← link(temp_head);
  end;
```

977.* Now we are ready to consider *vsplit* itself. Most of its work is accomplished by the two subroutines that we have just considered.

Given the number of a vlist box n, and given a desired page height h, the *vsplit* function finds the best initial segment of the vlist and returns a box for a page of height h. The remainder of the vlist, if any, replaces the original box, after removing glue and penalties and adjusting for *split_top_skip*. Mark nodes in the split-off box are used to set the values of *split_first_mark* and *split_bot_mark*; we use the fact that *split_first_mark* = *null* if and only if *split_bot_mark* = *null*.

The original box becomes "void" if and only if it has been entirely extracted. The extracted box is "void" if and only if the original box was void (or if it was, erroneously, an hlist box).

```
⟨Declare the function called do_marks 1560*⟩
function vsplit(n : halfword; h : scaled): pointer;  {extracts a page of height h from box n}
  label exit, done;
  var v: pointer;  {the box to be split}
    p: pointer;  {runs through the vlist}
    q: pointer;  {points to where the break occurs}
  begin cur_val ← n; fetch_box(v); flush_node_list(split_disc); split_disc ← null;
  if sa_mark ≠ null then
    if do_marks(vsplit_init, 0, sa_mark) then sa_mark ← null;
  if split_first_mark ≠ null then
    begin delete_token_ref(split_first_mark); split_first_mark ← null; delete_token_ref(split_bot_mark);
    split_bot_mark ← null;
    end;
  ⟨Dispense with trivial cases of void or bad boxes 978⟩;
  q ← vert_break(list_ptr(v), h, split_max_depth);
  ⟨Look at all the marks in nodes before the break, and set the final link to null at the break 979*⟩;
  q ← prune_page_top(q, saving_vdiscards > 0); p ← list_ptr(v); free_node(v, box_node_size);
  if q ≠ null then q ← vpack(q, natural);
  change_box(q);  {the eq_level of the box stays the same}
  vsplit ← vpackage(p, h, exactly, split_max_depth);
exit: end;
```

979.* It's possible that the box begins with a penalty node that is the "best" break, so we must be careful to handle this special case correctly.

```
⟨Look at all the marks in nodes before the break, and set the final link to null at the break 979*⟩ ≡
  p ← list_ptr(v);
  if p = q then list_ptr(v) ← null
  else loop begin if type(p) = mark_node then
        if mark_class(p) ≠ 0 then ⟨Update the current marks for vsplit 1562*⟩
        else if split_first_mark = null then
            begin split_first_mark ← mark_ptr(p); split_bot_mark ← split_first_mark;
            token_ref_count(split_first_mark) ← token_ref_count(split_first_mark) + 2;
            end
          else begin delete_token_ref(split_bot_mark); split_bot_mark ← mark_ptr(p);
            add_token_ref(split_bot_mark);
            end;
      if link(p) = q then
        begin link(p) ← null; goto done;
        end;
      p ← link(p);
      end;
done:
```

This code is used in section 977*.

982* An array *page_so_far* records the heights and depths of everything on the current page. This array contains six *scaled* numbers, like the similar arrays already considered in *line_break* and *vert_break*; and it also contains *page_goal* and *page_depth*, since these values are all accessible to the user via *set_page_dimen* commands. The value of *page_so_far*[1] is also called *page_total*. The stretch and shrink components of the `\skip` corrections for each insertion are included in *page_so_far*, but the natural space components of these corrections are not, since they have been subtracted from *page_goal*.

The variable *page_depth* records the depth of the current page; it has been adjusted so that it is at most *page_max_depth*. The variable *last_glue* points to the glue specification of the most recent node contributed from the contribution list, if this was a glue node; otherwise *last_glue* = *max_halfword*. (If the contribution list is nonempty, however, the value of *last_glue* is not necessarily accurate.) The variables *last_penalty*, *last_kern*, and *last_node_type* are similar. And finally, *insert_penalties* holds the sum of the penalties associated with all split and floating insertions.

```
  define page_goal ≡ page_so_far[0]   { desired height of information on page being built }
  define page_total ≡ page_so_far[1]   { height of the current page }
  define page_shrink ≡ page_so_far[6]   { shrinkability of the current page }
  define page_depth ≡ page_so_far[7]   { depth of the current page }
⟨ Global variables 13 ⟩ +≡
page_so_far: array [0 . . 7] of scaled;   { height and glue of the current page }
last_glue: pointer;   { used to implement \lastskip }
last_penalty: integer;   { used to implement \lastpenalty }
last_kern: scaled;   { used to implement \lastkern }
last_node_type: integer;   { used to implement \lastnodetype }
insert_penalties: integer;   { sum of the penalties for held-over insertions }
```

991* The page builder is ready to start a fresh page if we initialize the following state variables. (However, the page insertion list is initialized elsewhere.)

```
⟨ Start a new current page 991* ⟩ ≡
  page_contents ← empty; page_tail ← page_head; link(page_head) ← null;
  last_glue ← max_halfword; last_penalty ← 0; last_kern ← 0; last_node_type ← −1; page_depth ← 0;
  page_max_depth ← 0
```

This code is used in sections 215* and 1017.

996* ⟨ Update the values of *last_glue*, *last_penalty*, and *last_kern* 996* ⟩ ≡

```
  if last_glue ≠ max_halfword then delete_glue_ref(last_glue);
  last_penalty ← 0; last_kern ← 0; last_node_type ← type(p) + 1;
  if type(p) = glue_node then
    begin last_glue ← glue_ptr(p); add_glue_ref(last_glue);
    end
  else begin last_glue ← max_halfword;
    if type(p) = penalty_node then last_penalty ← penalty(p)
    else if type(p) = kern_node then last_kern ← width(p);
    end
```

This code is used in section 994.

999* ⟨Recycle node *p* 999*⟩ ≡

```
link(contrib_head) ← link(p); link(p) ← null;
if saving_vdiscards > 0 then
   begin if page_disc = null then page_disc ← p else link(tail_page_disc) ← p;
   tail_page_disc ← p;
   end
else flush_node_list(p)
```

This code is used in section 997.

1012* When the page builder has looked at as much material as could appear before the next page break, it makes its decision. The break that gave minimum badness will be used to put a completed "page" into box 255, with insertions appended to their other boxes.

We also set the values of *top_mark*, *first_mark*, and *bot_mark*. The program uses the fact that $bot\_mark \neq null$ implies $first\_mark \neq null$; it also knows that $bot\_mark = null$ implies $top\_mark = first\_mark = null$.

The *fire_up* subroutine prepares to output the current page at the best place; then it fires up the user's output routine, if there is one, or it simply ships out the page. There is one parameter, *c*, which represents the node that was being contributed to the page when the decision to force an output was made.

```
⟨Declare the procedure called fire_up 1012*⟩ ≡
procedure fire_up(c : pointer);
  label exit;
  var p, q, r, s: pointer; {nodes being examined and/or changed}
    prev_p: pointer; {predecessor of p}
    n: min_quarterword .. 255; {insertion box number}
    wait: boolean; {should the present insertion be held over?}
    save_vbadness: integer; {saved value of vbadness}
    save_vfuzz: scaled; {saved value of vfuzz}
    save_split_top_skip: pointer; {saved value of split_top_skip}
  begin ⟨Set the value of output_penalty 1013⟩;
  if sa_mark ≠ null then
    if do_marks(fire_up_init, 0, sa_mark) then sa_mark ← null;
  if bot_mark ≠ null then
    begin if top_mark ≠ null then delete_token_ref(top_mark);
    top_mark ← bot_mark; add_token_ref(top_mark); delete_token_ref(first_mark); first_mark ← null;
    end;
  ⟨Put the optimal current page into box 255, update first_mark and bot_mark, append insertions to their
        boxes, and put the remaining nodes back on the contribution list 1014*⟩;
  if sa_mark ≠ null then
    if do_marks(fire_up_done, 0, sa_mark) then sa_mark ← null;
  if (top_mark ≠ null) ∧ (first_mark = null) then
    begin first_mark ← top_mark; add_token_ref(top_mark);
    end;
  if output_routine ≠ null then
    if dead_cycles ≥ max_dead_cycles then
      ⟨Explain that too many dead cycles have occurred in a row 1024⟩
    else ⟨Fire up the user's output routine and return 1025⟩;
  ⟨Perform the default output routine 1023*⟩;
exit: end;
```

This code is used in section 994.

1014* As the page is finally being prepared for output, pointer *p* runs through the vlist, with *prev_p* trailing behind; pointer *q* is the tail of a list of insertions that are being held over for a subsequent page.

```
⟨ Put the optimal current page into box 255, update first_mark and bot_mark, append insertions to their
        boxes, and put the remaining nodes back on the contribution list 1014* ⟩ ≡
  if c = best_page_break then best_page_break ← null;  { c not yet linked in }
  ⟨ Ensure that box 255 is empty before output 1015 ⟩;
  insert_penalties ← 0;  { this will count the number of insertions held over }
  save_split_top_skip ← split_top_skip;
  if holding_inserts ≤ 0 then ⟨ Prepare all the boxes involved in insertions to act as queues 1018 ⟩;
  q ← hold_head; link(q) ← null; prev_p ← page_head; p ← link(prev_p);
  while p ≠ best_page_break do
    begin if type(p) = ins_node then
      begin if holding_inserts ≤ 0 then ⟨ Either insert the material specified by node p into the
            appropriate box, or hold it for the next page; also delete node p from the current page 1020 ⟩;
      end
    else if type(p) = mark_node then
        if mark_class(p) ≠ 0 then ⟨ Update the current marks for fire_up 1565* ⟩
        else ⟨ Update the values of first_mark and bot_mark 1016 ⟩;
    prev_p ← p; p ← link(prev_p);
    end;
  split_top_skip ← save_split_top_skip; ⟨ Break the current page at node p, put it in box 255, and put the
        remaining nodes on the contribution list 1017 ⟩;
  ⟨ Delete the page-insertion nodes 1019 ⟩
```

This code is used in section 1012*.

1021*

```
⟨ Wrap up the box specified by node r, splitting node p if called for; set wait ← true if node p
        holds a remainder after splitting 1021* ⟩ ≡
  begin if type(r) = split_up then
    if (broken_ins(r) = p) ∧ (broken_ptr(r) ≠ null) then
      begin while link(s) ≠ broken_ptr(r) do s ← link(s);
      link(s) ← null; split_top_skip ← split_top_ptr(p); ins_ptr(p) ← prune_page_top(broken_ptr(r), false);
      if ins_ptr(p) ≠ null then
        begin temp_ptr ← vpack(ins_ptr(p), natural); height(p) ← height(temp_ptr) + depth(temp_ptr);
        free_node(temp_ptr, box_node_size); wait ← true;
        end;
      end;
  best_ins_ptr(r) ← null; n ← qo(subtype(r)); temp_ptr ← list_ptr(box(n));
  free_node(box(n), box_node_size); box(n) ← vpack(temp_ptr, natural);
  end
```

This code is used in section 1020.

1023* The list of heldover insertions, running from *link*(*page_head*) to *page_tail*, must be moved to the contribution list when the user has specified no output routine.

```
⟨ Perform the default output routine 1023* ⟩ ≡
  begin if link(page_head) ≠ null then
    begin if link(contrib_head) = null then
      if nest_ptr = 0 then tail ← page_tail else contrib_tail ← page_tail
    else link(page_tail) ← link(contrib_head);
    link(contrib_head) ← link(page_head); link(page_head) ← null; page_tail ← page_head;
    end;
  flush_node_list(page_disc); page_disc ← null; ship_out(box(255)); box(255) ← null;
  end
```

This code is used in section 1012*.

1026* When the user's output routine finishes, it has constructed a vlist in internal vertical mode, and TEX will do the following:

```
⟨ Resume the page builder after an output routine has come to an end 1026* ⟩ ≡
  begin if (loc ≠ null) ∨ ((token_type ≠ output_text) ∧ (token_type ≠ backed_up)) then
    ⟨ Recover from an unbalanced output routine 1027 ⟩;
  end_token_list; { conserve stack space in case more outputs are triggered }
  end_graf; unsave; output_active ← false; insert_penalties ← 0;
  ⟨ Ensure that box 255 is empty after output 1028 ⟩;
  if tail ≠ head then { current list goes after heldover insertions }
    begin link(page_tail) ← link(head); page_tail ← tail;
    end;
  if link(page_head) ≠ null then { and both go before heldover contributions }
    begin if link(contrib_head) = null then contrib_tail ← page_tail;
    link(page_tail) ← link(contrib_head); link(contrib_head) ← link(page_head); link(page_head) ← null;
    page_tail ← page_head;
    end;
  flush_node_list(page_disc); page_disc ← null; pop_nest; build_page;
  end
```

This code is used in section 1100.

1070*. Here is where we clear the parameters that are supposed to revert to their default values after every paragraph and when internal vertical mode is entered.

```
⟨ Declare action procedures for use by main_control 1043 ⟩ +≡
procedure normal_paragraph;
  begin if looseness $\neq 0$ then eq_word_define(int_base + looseness_code, 0);
  if hang_indent $\neq 0$ then eq_word_define(dimen_base + hang_indent_code, 0);
  if hang_after $\neq 1$ then eq_word_define(int_base + hang_after_code, 1);
  if par_shape_ptr $\neq$ null then eq_define(par_shape_loc, shape_ref, null);
  if inter_line_penalties_ptr $\neq$ null then eq_define(inter_line_penalties_loc, shape_ref, null);
  end;
```

1071*. Now let's turn to the question of how `\hbox` is treated. We actually need to consider also a slightly larger context, since constructions like '`\setbox3=\hbox...`' and '`\leaders\hbox...`' and '`\lower3.8pt\hbox...`' are supposed to invoke quite different actions after the box has been packaged. Conversely, constructions like '`\setbox3=`' can be followed by a variety of different kinds of boxes, and we would like to encode such things in an efficient way.

In other words, there are two problems: to represent the context of a box, and to represent its type.

The first problem is solved by putting a "context code" on the *save_stack*, just below the two entries that give the dimensions produced by *scan_spec*. The context code is either a (signed) shift amount, or it is a large integer ≥ *box_flag*, where *box_flag* = 2^{30}. Codes *box_flag* through *global_box_flag* − 1 represent '`\setbox0`' through '`\setbox32767`'; codes *global_box_flag* through *ship_out_flag* − 1 represent '`\global\setbox0`' through '`\global\setbox32767`'; code *ship_out_flag* represents '`\shipout`'; and codes *leader_flag* through *leader_flag* + 2 represent '`\leaders`', '`\cleaders`', and '`\xleaders`'.

The second problem is solved by giving the command code *make_box* to all control sequences that produce a box, and by using the following *chr_code* values to distinguish between them: *box_code*, *copy_code*, *last_box_code*, *vsplit_code*, *vtop_code*, *vtop_code* + *vmode*, and *vtop_code* + *hmode*, where the latter two are used to denote `\vbox` and `\hbox`, respectively.

```
  define box_flag ≡ ´10000000000   { context code for '\setbox0' }
  define global_box_flag ≡ ´10000100000   { context code for '\global\setbox0' }
  define ship_out_flag ≡ ´10000200000   { context code for '\shipout' }
  define leader_flag ≡ ´10000200001   { context code for '\leaders' }
  define box_code = 0   { chr_code for '\box' }
  define copy_code = 1   { chr_code for '\copy' }
  define last_box_code = 2   { chr_code for '\lastbox' }
  define vsplit_code = 3   { chr_code for '\vsplit' }
  define vtop_code = 4   { chr_code for '\vtop' }
```

```
⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive("moveleft", hmove, 1); primitive("moveright", hmove, 0);
  primitive("raise", vmove, 1); primitive("lower", vmove, 0);
  primitive("box", make_box, box_code); primitive("copy", make_box, copy_code);
  primitive("lastbox", make_box, last_box_code); primitive("vsplit", make_box, vsplit_code);
  primitive("vtop", make_box, vtop_code);
  primitive("vbox", make_box, vtop_code + vmode); primitive("hbox", make_box, vtop_code + hmode);
  primitive("shipout", leader_ship, a_leaders − 1);   { ship_out_flag = leader_flag − 1 }
  primitive("leaders", leader_ship, a_leaders); primitive("cleaders", leader_ship, c_leaders);
  primitive("xleaders", leader_ship, x_leaders);
```

1075* The *box_end* procedure does the right thing with *cur_box*, if *box_context* represents the context as explained above.

```
⟨ Declare action procedures for use by main_control 1043 ⟩ +≡
procedure box_end(box_context : integer);
  var p: pointer;  { ord_noad for new box in math mode }
    a: small_number;  { global prefix }
  begin if box_context < box_flag then
    ⟨ Append box cur_box to the current list, shifted by box_context 1076 ⟩
  else if box_context < ship_out_flag then ⟨ Store cur_box in a box register 1077* ⟩
    else if cur_box ≠ null then
        if box_context > ship_out_flag then ⟨ Append a new leader node that uses cur_box 1078 ⟩
        else ship_out(cur_box);
  end;
```

1077* ⟨ Store *cur_box* in a box register 1077* ⟩ ≡

```
  begin if box_context < global_box_flag then
    begin cur_val ← box_context − box_flag; a ← 0;
    end
  else begin cur_val ← box_context − global_box_flag; a ← 4;
    end;
  if cur_val < 256 then define(box_base + cur_val, box_ref, cur_box)
  else sa_def_box;
  end
```

This code is used in section 1075*.

1079* Now that we can see what eventually happens to boxes, we can consider the first steps in their creation. The *begin_box* routine is called when *box_context* is a context specification, *cur_chr* specifies the type of box desired, and *cur_cmd* = *make_box*.

```
⟨ Declare action procedures for use by main_control 1043 ⟩ +≡
procedure begin_box(box_context : integer);
  label exit, done;
  var p, q: pointer;  { run through the current list }
    r: pointer;  { running behind p }
    fm: boolean;  { a final \beginM \endM node pair? }
    tx: pointer;  { effective tail node }
    m: quarterword;  { the length of a replacement list }
    k: halfword;  { 0 or vmode or hmode }
    n: halfword;  { a box number }
  begin case cur_chr of
  box_code: begin scan_register_num; fetch_box(cur_box); change_box(null);
        { the box becomes void, at the same level }
    end;
  copy_code: begin scan_register_num; fetch_box(q); cur_box ← copy_node_list(q);
    end;
  last_box_code: ⟨ If the current list ends with a box node, delete it from the list and make cur_box point to
        it; otherwise set cur_box ← null 1080* ⟩;
  vsplit_code: ⟨ Split off part of a vertical box, make cur_box point to it 1082* ⟩;
  othercases ⟨ Initiate the construction of an hbox or vbox, then return 1083 ⟩
  endcases;
  box_end(box_context);  { in simple cases, we use the box immediately }
exit: end;
```

1080* Note that the condition $\neg is\_char\_node(tail)$ implies that $head \neq tail$, since *head* is a one-word node.

```
define fetch_effective_tail_eTeX(#) ≡ { extract tx, drop \beginM \endM pair }
      q ← head; p ← null;
      repeat r ← p; p ← q; fm ← false;
        if ¬is_char_node(q) then
          if type(q) = disc_node then
            begin for m ← 1 to replace_count(q) do p ← link(p);
            if p = tx then #;
            end
          else if (type(q) = math_node) ∧ (subtype(q) = begin_M_code) then fm ← true;
        q ← link(p);
      until q = tx; { found r. .p. .q = tx }
      q ← link(tx); link(p) ← q; link(tx) ← null;
      if q = null then
        if fm then confusion("tail1")
        else tail ← p
      else if fm then { r. .p = begin_M . .q = end_M }
          begin tail ← r; link(r) ← null; flush_node_list(p); end
define check_effective_tail(#) ≡ find_effective_tail_eTeX
define fetch_effective_tail ≡ fetch_effective_tail_eTeX
```

⟨ If the current list ends with a box node, delete it from the list and make *cur_box* point to it; otherwise set *cur_box* ← *null* 1080* ⟩ ≡

```
  begin cur_box ← null;
  if abs(mode) = mmode then
    begin you_cant; help1("Sorry;␣this␣\lastbox␣will␣be␣void."); error;
    end
  else if (mode = vmode) ∧ (head = tail) then
      begin you_cant; help2("Sorry...I␣usually␣can´t␣take␣things␣from␣the␣current␣page.")
      ("This␣\lastbox␣will␣therefore␣be␣void."); error;
      end
    else begin check_effective_tail(goto done);
      if ¬is_char_node(tx) then
        if (type(tx) = hlist_node) ∨ (type(tx) = vlist_node) then
          ⟨ Remove the last box, unless it's part of a discretionary 1081* ⟩;
    done: end;
  end
```

This code is used in section 1079*.

1081* ⟨ Remove the last box, unless it's part of a discretionary 1081* ⟩ ≡

```
  begin fetch_effective_tail(goto done); cur_box ← tx; shift_amount(cur_box) ← 0;
  end
```

This code is used in section 1080*.

1082* Here we deal with things like '`\vsplit 13 to 100pt`'.

```
⟨ Split off part of a vertical box, make cur_box point to it 1082* ⟩ ≡
  begin scan_register_num; n ← cur_val;
  if ¬scan_keyword("to") then
    begin print_err("Missing␣`to´␣inserted");
    help2("I´m␣working␣on␣`\vsplit<box␣number>␣to␣<dimen>´;")
    ("will␣look␣for␣the␣<dimen>␣next."); error;
    end;
  scan_normal_dimen; cur_box ← vsplit(n, cur_val);
  end
```

This code is used in section 1079*.

1096* ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
procedure end_graf;
  begin if mode = hmode then
    begin if head = tail then pop_nest   { null paragraphs are ignored }
    else line_break(false);
    if LR_save ≠ null then
      begin flush_list(LR_save); LR_save ← null;
      end;
    normal_paragraph; error_count ← 0;
    end;
  end;
```

1101* ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
procedure make_mark;
  var p: pointer;   { new node }
    c: halfword;   { the mark class }
  begin if cur_chr = 0 then c ← 0
  else begin scan_register_num; c ← cur_val;
    end;
  p ← scan_toks(false, true); p ← get_node(small_node_size); mark_class(p) ← c; type(p) ← mark_node;
  subtype(p) ← 0;   { the subtype is not used }
  mark_ptr(p) ← def_ref; link(tail) ← p; tail ← p;
  end;
```

1105* When *delete_last* is called, *cur_chr* is the *type* of node that will be deleted, if present.

```
⟨ Declare action procedures for use by main_control 1043 ⟩ +≡
procedure delete_last;
  label exit;
  var p, q: pointer; { run through the current list }
    r: pointer; { running behind p }
    fm: boolean; { a final \beginM \endM node pair? }
    tx: pointer; { effective tail node }
    m: quarterword; { the length of a replacement list }
  begin if (mode = vmode) ∧ (tail = head) then
    ⟨ Apologize for inability to do the operation now, unless \unskip follows non-glue 1106 ⟩
  else begin check_effective_tail(return);
    if ¬is_char_node(tx) then
      if type(tx) = cur_chr then
        begin fetch_effective_tail(return); flush_node_list(tx);
        end;
    end;
exit: end;
```

1108* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
remove_item: if chr_code = glue_node then print_esc("unskip")
  else if chr_code = kern_node then print_esc("unkern")
    else print_esc("unpenalty");
un_hbox: if chr_code = copy_code then print_esc("unhcopy")
  else print_esc("unhbox");
un_vbox: if chr_code = copy_code then print_esc("unvcopy") ⟨ Cases of un_vbox for print_cmd_chr 1597* ⟩
  else print_esc("unvbox");
```

1110* ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
procedure unpackage;
  label done, exit;
  var p: pointer; { the box }
    c: box_code .. copy_code; { should we copy? }
  begin if cur_chr > copy_code then ⟨ Handle saved items and goto done 1598* ⟩;
  c ← cur_chr; scan_register_num; fetch_box(p);
  if p = null then return;
  if (abs(mode) = mmode) ∨ ((abs(mode) = vmode) ∧ (type(p) ≠ vlist_node)) ∨
        ((abs(mode) = hmode) ∧ (type(p) ≠ hlist_node)) then
    begin print_err("Incompatible␣list␣can´t␣be␣unboxed");
    help3("Sorry,␣Pandora.␣(You␣sneaky␣devil.)")
    ("I␣refuse␣to␣unbox␣an␣\hbox␣in␣vertical␣mode␣or␣vice␣versa.")
    ("And␣I␣can´t␣open␣any␣boxes␣in␣math␣mode.");
    error; return;
    end;
  if c = copy_code then link(tail) ← copy_node_list(list_ptr(p))
  else begin link(tail) ← list_ptr(p); change_box(null); free_node(p, box_node_size);
    end;
done: while link(tail) ≠ null do tail ← link(tail);
exit: end;
```

1130* We've now covered most of the abuses of `\halign` and `\valign`. Let's take a look at what happens when they are used correctly.

```
⟨ Cases of main_control that build boxes and lists 1056 ⟩ +≡
vmode + halign: init_align;
hmode + valign: ⟨ Cases of main_control for hmode + valign 1434* ⟩
  init_align;
mmode + halign: if privileged then
    if cur_group = math_shift_group then init_align
    else off_save;
vmode + endv, hmode + endv: do_endv;
```

1138* ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
⟨ Declare subprocedures for init_math 1468* ⟩
procedure init_math;
  label reswitch, found, not_found, done;
  var w: scaled;  { new or partial pre_display_size }
    j: pointer;  { prototype box for display }
    x: integer;  { new pre_display_direction }
    l: scaled;  { new display_width }
    s: scaled;  { new display_indent }
    p: pointer;  { current node when calculating pre_display_size }
    q: pointer;  { glue specification when calculating pre_display_size }
    f: internal_font_number;  { font in current char_node }
    n: integer;  { scope of paragraph shape specification }
    v: scaled;  { w plus possible glue amount }
    d: scaled;  { increment to v }
  begin get_token;  { get_x_token would fail on \ifmmode! }
  if $(cur\_cmd = math\_shift) \land (mode > 0)$ then ⟨ Go into display math mode 1145* ⟩
  else begin back_input; ⟨ Go into ordinary math mode 1139 ⟩;
    end;
  end;
```

1145* When we enter display math mode, we need to call *line_break* to process the partial paragraph that has just been interrupted by the display. Then we can set the proper values of *display_width* and *display_indent* and *pre_display_size*.

```
⟨ Go into display math mode 1145* ⟩ ≡
  begin $j \leftarrow null$; $w \leftarrow -max\_dimen$;
  if $head = tail$ then  { '\noindent$$' or '$$ $$' }
    ⟨ Prepare for display after an empty paragraph 1467* ⟩
  else begin line_break(true);
    ⟨ Calculate the natural width, w, by which the characters of the final line extend to the right of the
          reference point, plus two ems; or set $w \leftarrow max\_dimen$ if the non-blank information on that line is
          affected by stretching or shrinking 1146* ⟩;
    end;  { now we are in vertical mode, working on the list that will contain the display }
  ⟨ Calculate the length, l, and the shift amount, s, of the display lines 1149 ⟩;
  push_math(math_shift_group); $mode \leftarrow mmode$; eq_word_define($int\_base + cur\_fam\_code, -1$);
  eq_word_define($dimen\_base + pre\_display\_size\_code, w$); $LR\_box \leftarrow j$;
  if eTeX_ex then eq_word_define($int\_base + pre\_display\_direction\_code, x$);
  eq_word_define($dimen\_base + display\_width\_code, l$); eq_word_define($dimen\_base + display\_indent\_code, s$);
  if $every\_display \neq null$ then begin_token_list(every_display, every_display_text);
  if $nest\_ptr = 1$ then build_page;
  end
```

This code is used in section 1138*.

1146*. ⟨Calculate the natural width, w, by which the characters of the final line extend to the right of the reference point, plus two ems; or set $w \leftarrow max\_dimen$ if the non-blank information on that line is affected by stretching or shrinking 1146*⟩ ≡

```
  ⟨Prepare for display after a non-empty paragraph 1469*⟩;
  while $p \neq null$ do
    begin ⟨Let $d$ be the natural width of node $p$; if the node is “visible,” goto found; if the node is glue
        that stretches or shrinks, set $v \leftarrow max\_dimen$ 1147*⟩;
    if $v < max\_dimen$ then $v \leftarrow v + d$;
    goto not_found;
  found: if $v < max\_dimen$ then
      begin $v \leftarrow v + d$; $w \leftarrow v$;
      end
    else begin $w \leftarrow max\_dimen$; goto done;
      end;
  not_found: $p \leftarrow link(p)$;
    end;
done: ⟨Finish the natural width computation 1470*⟩
```

This code is used in section 1145*.

1147*. ⟨Let d be the natural width of node p; if the node is “visible,” **goto** *found*; if the node is glue that stretches or shrinks, set $v \leftarrow max\_dimen$ 1147*⟩ ≡

```
reswitch: if is_char_node(p) then
    begin $f \leftarrow font(p)$; $d \leftarrow char\_width(f)(char\_info(f)(character(p)))$; goto found;
    end;
  case type(p) of
  hlist_node, vlist_node, rule_node: begin $d \leftarrow width(p)$; goto found;
    end;
  ligature_node: ⟨Make node $p$ look like a char_node and goto reswitch 652⟩;
  kern_node: $d \leftarrow width(p)$;
  ⟨Cases of ‘Let $d$ be the natural width’ that need special treatment 1471*⟩
  glue_node: ⟨Let $d$ be the natural width of this glue; if stretching or shrinking, set $v \leftarrow max\_dimen$; goto
      found in the case of leaders 1148⟩;
  whatsit_node: ⟨Let $d$ be the width of the whatsit $p$ 1361⟩;
  othercases $d \leftarrow 0$
  endcases
```

This code is used in section 1146*.

1185*. ⟨Compleat the incompleat noad 1185*⟩ ≡

```
  begin $math\_type(denominator(incompleat\_noad)) \leftarrow sub\_mlist$;
  $info(denominator(incompleat\_noad)) \leftarrow link(head)$;
  if $p = null$ then $q \leftarrow incompleat\_noad$
  else begin $q \leftarrow info(numerator(incompleat\_noad))$;
    if $(type(q) \neq left\_noad) \lor (delim\_ptr = null)$ then confusion("right");
    $info(numerator(incompleat\_noad)) \leftarrow link(delim\_ptr)$; $link(delim\_ptr) \leftarrow incompleat\_noad$;
    $link(incompleat\_noad) \leftarrow p$;
    end;
  end
```

This code is used in section 1184.

1189*. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
left_right: if chr_code = left_noad then print_esc("left")
  ⟨ Cases of left_right for print_cmd_chr 1429* ⟩
else print_esc("right");
```

1191*. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
procedure math_left_right;
  var t: small_number;  { left_noad or right_noad }
    p: pointer;  { new noad }
    q: pointer;  { resulting mlist }
  begin t ← cur_chr;
  if (t ≠ left_noad) ∧ (cur_group ≠ math_left_group) then ⟨ Try to recover from mismatched \right 1192* ⟩
  else begin p ← new_noad; type(p) ← t; scan_delimiter(delimiter(p), false);
    if t = middle_noad then
      begin type(p) ← right_noad; subtype(p) ← middle_noad;
      end;
    if t = left_noad then q ← p
    else begin q ← fin_mlist(p); unsave;  { end of math_left_group }
      end;
    if t ≠ right_noad then
      begin push_math(math_left_group); link(head) ← q; tail ← p; delim_ptr ← p;
      end
    else begin tail_append(new_noad); type(tail) ← inner_noad; math_type(nucleus(tail)) ← sub_mlist;
      info(nucleus(tail)) ← q;
      end;
    end;
  end;
```

1192*. ⟨ Try to recover from mismatched \right 1192* ⟩ ≡

```
  begin if cur_group = math_shift_group then
    begin scan_delimiter(garbage, false); print_err("Extra␣");
    if t = middle_noad then
      begin print_esc("middle"); help1("I´m␣ignoring␣a␣\middle␣that␣had␣no␣matching␣\left.");
      end
    else begin print_esc("right"); help1("I´m␣ignoring␣a␣\right␣that␣had␣no␣matching␣\left.");
      end;
    error;
    end
  else off_save;
  end
```

This code is used in section 1191*.

```
1194.* ⟨Declare action procedures for use by main_control 1043⟩ +≡
⟨Declare subprocedures for after_math 1479*⟩
procedure after_math;
  var l: boolean;  {‘\leqno’ instead of ‘\eqno’}
    danger: boolean;  {not enough symbol fonts are present}
    m: integer;  {mmode or −mmode}
    p: pointer;  {the formula}
    a: pointer;  {box containing equation number}
    ⟨Local variables for finishing a displayed formula 1198⟩
  begin danger ← false; ⟨Retrieve the prototype box 1477*⟩;
  ⟨Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set
      danger ← true 1195⟩;
  m ← mode; l ← false; p ← fin_mlist(null);  {this pops the nest}
  if mode = −m then  {end of equation number}
    begin ⟨Check that another $ follows 1197⟩;
    cur_mlist ← p; cur_style ← text_style; mlist_penalties ← false; mlist_to_hlist;
    a ← hpack(link(temp_head), natural); set_box_lr(a)(dlist); unsave; decr(save_ptr);
        {now cur_group = math_shift_group}
    if saved(0) = 1 then l ← true;
    danger ← false; ⟨Retrieve the prototype box 1477*⟩;
    ⟨Check that the necessary fonts for math symbols are present; if not, flush the current math lists and
        set danger ← true 1195⟩;
    m ← mode; p ← fin_mlist(null);
    end
  else a ← null;
  if m < 0 then ⟨Finish math in text 1196⟩
  else begin if a = null then ⟨Check that another $ follows 1197⟩;
    ⟨Finish displayed math 1199*⟩;
    end;
  end;
```

1199*. At this time p points to the mlist for the formula; a is either *null* or it points to a box containing the equation number; and we are in vertical mode (or internal vertical mode).

```
⟨ Finish displayed math 1199* ⟩ ≡
  $cur\_mlist \gets p$; $cur\_style \gets display\_style$; $mlist\_penalties \gets false$; $mlist\_to\_hlist$; $p \gets link(temp\_head)$;
  $adjust\_tail \gets adjust\_head$; $b \gets hpack(p, natural)$; $p \gets list\_ptr(b)$; $t \gets adjust\_tail$; $adjust\_tail \gets null$;
  $w \gets width(b)$; $z \gets display\_width$; $s \gets display\_indent$;
  if $pre\_display\_direction < 0$ then $s \gets -s - z$;
  if $(a = null) \lor danger$ then
    begin $e \gets 0$; $q \gets 0$;
    end
  else begin $e \gets width(a)$; $q \gets e + math\_quad(text\_size)$;
    end;
  if $w + q > z$ then ⟨ Squeeze the equation as much as possible; if there is an equation number that should
        go on a separate line by itself, set $e \gets 0$ 1201 ⟩;
  ⟨ Determine the displacement, $d$, of the left edge of the equation, with respect to the line size $z$, assuming
      that $l = false$ 1202* ⟩;
  ⟨ Append the glue or equation number preceding the display 1203* ⟩;
  ⟨ Append the display and perhaps also the equation number 1204* ⟩;
  ⟨ Append the glue or equation number following the display 1205* ⟩;
  ⟨ Flush the prototype box 1478* ⟩;
  $resume\_after\_display$
```

This code is used in section 1194*.

1202*. We try first to center the display without regard to the existence of the equation number. If that would make it too close (where "too close" means that the space between display and equation number is less than the width of the equation number), we either center it in the remaining space or move it as far from the equation number as possible. The latter alternative is taken only if the display begins with glue, since we assume that the user put glue there to control the spacing precisely.

```
⟨ Determine the displacement, $d$, of the left edge of the equation, with respect to the line size $z$, assuming
      that $l = false$ 1202* ⟩ ≡
  $set\_box\_lr(b)(dlist)$; $d \gets half(z - w)$;
  if $(e > 0) \land (d < 2 * e)$ then   { too close }
    begin $d \gets half(z - w - e)$;
    if $p \neq null$ then
      if $\lnot is\_char\_node(p)$ then
        if $type(p) = glue\_node$ then $d \gets 0$;
    end
```

This code is used in section 1199*.

1203* If the equation number is set on a line by itself, either before or after the formula, we append an infinite penalty so that no page break will separate the display from its number; and we use the same size and displacement for all three potential lines of the display, even though '`\parshape`' may specify them differently.

```
⟨ Append the glue or equation number preceding the display 1203* ⟩ ≡
  tail_append(new_penalty(pre_display_penalty));
  if $(d + s \le pre\_display\_size) \lor l$ then   { not enough clearance }
    begin g1 ← above_display_skip_code; g2 ← below_display_skip_code;
    end
  else begin g1 ← above_display_short_skip_code; g2 ← below_display_short_skip_code;
    end;
  if $l \land (e = 0)$ then   { it follows that type(a) = hlist_node }
    begin app_display(j, a, 0); tail_append(new_penalty(inf_penalty));
    end
  else tail_append(new_param_glue(g1))
```

This code is used in section 1199*.

```
1204*  ⟨ Append the display and perhaps also the equation number 1204* ⟩ ≡
  if $e \ne 0$ then
    begin r ← new_kern($z - w - e - d$);
    if l then
      begin link(a) ← r; link(r) ← b; b ← a; d ← 0;
      end
    else begin link(b) ← r; link(r) ← a;
      end;
    b ← hpack(b, natural);
    end;
  app_display(j, b, d)
```

This code is used in section 1199*.

```
1205*  ⟨ Append the glue or equation number following the display 1205* ⟩ ≡
  if $(a \ne null) \land (e = 0) \land \neg l$ then
    begin tail_append(new_penalty(inf_penalty)); app_display(j, a, $z - width(a)$); g2 ← 0;
    end;
  if $t \ne adjust\_head$ then   { migrating material comes after equation number }
    begin link(tail) ← link(adjust_head); tail ← t;
    end;
  tail_append(new_penalty(post_display_penalty));
  if $g2 > 0$ then tail_append(new_param_glue(g2))
```

This code is used in section 1199*.

1206* When `\halign` appears in a display, the alignment routines operate essentially as they do in vertical mode. Then the following program is activated, with p and q pointing to the beginning and end of the resulting list, and with *aux_save* holding the *prev_depth* value.

```
⟨ Finish an alignment in a display 1206* ⟩ ≡
  begin do_assignments;
  if cur_cmd ≠ math_shift then ⟨ Pontificate about improper alignment in display 1207 ⟩
  else ⟨ Check that another $ follows 1197 ⟩;
  flush_node_list(LR_box); pop_nest; tail_append(new_penalty(pre_display_penalty));
  tail_append(new_param_glue(above_display_skip_code)); link(tail) ← p;
  if p ≠ null then tail ← q;
  tail_append(new_penalty(post_display_penalty)); tail_append(new_param_glue(below_display_skip_code));
  prev_depth ← aux_save.sc; resume_after_display;
  end
```

This code is used in section 812.

1208*. Mode-independent processing. The long *main_control* procedure has now been fully specified, except for certain activities that are independent of the current mode. These activities do not change the current vlist or hlist or mlist; if they change anything, it is the value of a parameter or the meaning of a control sequence.

Assignments to values in *eqtb* can be global or local. Furthermore, a control sequence can be defined to be '`\long`', '`\protected`', or '`\outer`', and it might or might not be expanded. The prefixes '`\global`', '`\long`', '`\protected`', and '`\outer`' can occur in any order. Therefore we assign binary numeric codes, making it possible to accumulate the union of all specified prefixes by adding the corresponding codes. (Pascal's **set** operations could also have been used.)

```
⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive("long", prefix, 1); primitive("outer", prefix, 2); primitive("global", prefix, 4);
  primitive("def", def, 0); primitive("gdef", def, 1); primitive("edef", def, 2); primitive("xdef", def, 3);
```

1209*. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
prefix: if $chr\_code = 1$ then print_esc("long")
  else if $chr\_code = 2$ then print_esc("outer")
  ⟨ Cases of prefix for print_cmd_chr 1506* ⟩
else print_esc("global");
def: if $chr\_code = 0$ then print_esc("def")
  else if $chr\_code = 1$ then print_esc("gdef")
    else if $chr\_code = 2$ then print_esc("edef")
      else print_esc("xdef");
```

1211*. If the user says, e.g., '`\global\global`', the redundancy is silently accepted.

```
⟨ Declare action procedures for use by main_control 1043 ⟩ +≡
⟨ Declare subprocedures for prefixed_command 1215 ⟩
procedure prefixed_command;
  label done, exit;
  var a: small_number; { accumulated prefix codes so far }
    f: internal_font_number; { identifies a font }
    j: halfword; { index into a \parshape specification }
    k: font_index; { index into font_info }
    p, q: pointer; { for temporary short-term use }
    n: integer; { ditto }
    e: boolean; { should a definition be expanded? or was \let not done? }
  begin $a \leftarrow 0$;
  while $cur\_cmd = prefix$ do
    begin if $\neg odd(a \mathbin{\bf div} cur\_chr)$ then $a \leftarrow a + cur\_chr$;
    ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
    if $cur\_cmd \le max\_non\_prefixed\_command$ then ⟨ Discard erroneous prefixes and return 1212* ⟩;
    if $tracing\_commands > 2$ then
      if eTeX_ex then show_cur_cmd_chr;
    end;
  ⟨ Discard the prefixes \long and \outer if they are irrelevant 1213* ⟩;
  ⟨ Adjust for the setting of \globaldefs 1214 ⟩;
  case cur_cmd of
  ⟨ Assignments 1217 ⟩
  othercases confusion("prefix")
  endcases;
done: ⟨ Insert a token saved by \afterassignment, if any 1269 ⟩;
exit: end;
```

1212* ⟨Discard erroneous prefixes and **return** 1212*⟩ ≡

```
begin print_err("You␣can´t␣use␣a␣prefix␣with␣`"); print_cmd_chr(cur_cmd, cur_chr);
print_char("´"); help1("I´ll␣pretend␣you␣didn´t␣say␣\long␣or␣\outer␣or␣\global.");
if eTeX_ex then
  help_line[0] ← "I´ll␣pretend␣you␣didn´t␣say␣\long␣or␣\outer␣or␣\global␣or␣\protected.";
back_error; return;
end
```

This code is used in section 1211*.

1213* ⟨Discard the prefixes `\long` and `\outer` if they are irrelevant 1213*⟩ ≡

```
if $a \ge 8$ then
  begin j ← protected_token; a ← a − 8;
  end
else j ← 0;
if $(cur\_cmd \ne def) \land ((a \bmod 4 \ne 0) \lor (j \ne 0))$ then
  begin print_err("You␣can´t␣use␣`"); print_esc("long"); print("´␣or␣`"); print_esc("outer");
  help1("I´ll␣pretend␣you␣didn´t␣say␣\long␣or␣\outer␣here.");
  if eTeX_ex then
    begin help_line[0] ← "I´ll␣pretend␣you␣didn´t␣say␣\long␣or␣\outer␣or␣\protected␣here.";
    print("´␣or␣`"); print_esc("protected");
    end;
  print("´␣with␣`"); print_cmd_chr(cur_cmd, cur_chr); print_char("´"); error;
  end
```

This code is used in section 1211*.

1218* When a *def* command has been scanned, *cur_chr* is odd if the definition is supposed to be global, and $cur\_chr \ge 2$ if the definition is supposed to be expanded.

⟨Assignments 1217⟩ +≡

```
def: begin if $odd(cur\_chr) \land \neg global \land (global\_defs \ge 0)$ then a ← a + 4;
  e ← (cur_chr ≥ 2); get_r_token; p ← cur_cs; q ← scan_toks(true, e);
  if $j \ne 0$ then
    begin q ← get_avail; info(q) ← j; link(q) ← link(def_ref); link(def_ref) ← q;
    end;
  define(p, call + (a mod 4), def_ref);
  end;
```

1221* ⟨ Assignments 1217 ⟩ +≡

```
let: begin n ← cur_chr; get_r_token; p ← cur_cs;
  if n = normal then
    begin repeat get_token;
    until cur_cmd ≠ spacer;
    if cur_tok = other_token + "=" then
      begin get_token;
      if cur_cmd = spacer then get_token;
      end;
    end
  else begin get_token; q ← cur_tok; get_token; back_input; cur_tok ← q; back_input;
        { look ahead, then back up }
    end; { note that back_input doesn't affect cur_cmd, cur_chr }
  if cur_cmd ≥ call then add_token_ref(cur_chr)
  else if (cur_cmd = register) ∨ (cur_cmd = toks_register) then
      if (cur_chr < mem_bot) ∨ (cur_chr > lo_mem_stat_max) then add_sa_ref(cur_chr);
  define(p, cur_cmd, cur_chr);
  end;
```

1224* We temporarily define *p* to be *relax*, so that an occurrence of *p* while scanning the definition will simply stop the scanning instead of producing an "undefined control sequence" error or expanding the previous meaning. This allows, for instance, '`\chardef\foo=123\foo`'.

⟨ Assignments 1217 ⟩ +≡

```
shorthand_def: begin n ← cur_chr; get_r_token; p ← cur_cs; define(p, relax, 256); scan_optional_equals;
  case n of
  char_def_code: begin scan_char_num; define(p, char_given, cur_val);
    end;
  math_char_def_code: begin scan_fifteen_bit_int; define(p, math_given, cur_val);
    end;
  othercases begin scan_register_num;
    if cur_val > 255 then
      begin j ← n − count_def_code; { int_val .. box_val }
      if j > mu_val then j ← tok_val; { int_val .. mu_val or tok_val }
      find_sa_element(j, cur_val, true); add_sa_ref(cur_ptr);
      if j = tok_val then j ← toks_register else j ← register;
      define(p, j, cur_ptr);
      end
    else case n of
      count_def_code: define(p, assign_int, count_base + cur_val);
      dimen_def_code: define(p, assign_dimen, scaled_base + cur_val);
      skip_def_code: define(p, assign_glue, skip_base + cur_val);
      mu_skip_def_code: define(p, assign_mu_glue, mu_skip_base + cur_val);
      toks_def_code: define(p, assign_toks, toks_base + cur_val);
      end; { there are no other cases }
    end
  endcases;
  end;
```

1225* ⟨ Assignments 1217 ⟩ +≡

```
read_to_cs: begin j ← cur_chr; scan_int; n ← cur_val;
  if ¬scan_keyword("to") then
    begin print_err("Missing␣`to´␣inserted");
    help2("You␣should␣have␣said␣`\read<number>␣to␣\cs´.")
    ("I´m␣going␣to␣look␣for␣the␣\cs␣now."); error;
    end;
  get_r_token; p ← cur_cs; read_toks(n, p, j); define(p, call, cur_val);
  end;
```

1226* The token-list parameters, `\output` and `\everypar`, etc., receive their values in the following way. (For safety's sake, we place an enclosing pair of braces around an `\output` list.)

```
⟨ Assignments 1217 ⟩ +≡
toks_register, assign_toks: begin q ← cur_cs; e ← false;
      { just in case, will be set true for sparse array elements }
  if cur_cmd = toks_register then
    if cur_chr = mem_bot then
      begin scan_register_num;
      if cur_val > 255 then
        begin find_sa_element(tok_val, cur_val, true); cur_chr ← cur_ptr; e ← true;
        end
      else cur_chr ← toks_base + cur_val;
      end
    else e ← true;
  p ← cur_chr;   { p = every_par_loc or output_routine_loc or ... }
  scan_optional_equals; ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
  if cur_cmd ≠ left_brace then ⟨ If the right-hand side is a token parameter or token register, finish the
        assignment and goto done 1227* ⟩;
  back_input; cur_cs ← q; q ← scan_toks(false, false);
  if link(def_ref) = null then   { empty list: revert to the default }
    begin sa_define(p, null)(p, undefined_cs, null); free_avail(def_ref);
    end
  else begin if (p = output_routine_loc) ∧ ¬e then   { enclose in curlies }
      begin link(q) ← get_avail; q ← link(q); info(q) ← right_brace_token + "}"; q ← get_avail;
      info(q) ← left_brace_token + "{"; link(q) ← link(def_ref); link(def_ref) ← q;
      end;
    sa_define(p, def_ref)(p, call, def_ref);
    end;
  end;
```

1227* ⟨ If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1227* ⟩ ≡

```
if (cur_cmd = toks_register) ∨ (cur_cmd = assign_toks) then
  begin if cur_cmd = toks_register then
    if cur_chr = mem_bot then
      begin scan_register_num;
      if cur_val < 256 then q ← equiv(toks_base + cur_val)
      else begin find_sa_element(tok_val, cur_val, false);
        if cur_ptr = null then q ← null
        else q ← sa_ptr(cur_ptr);
        end;
      end
    else q ← sa_ptr(cur_chr)
  else q ← equiv(cur_chr);
  if q = null then sa_define(p, null)(p, undefined_cs, null)
  else begin add_token_ref(q); sa_define(p, q)(p, call, q);
    end;
  goto done;
  end
```

This code is used in section 1226*.

1236* We use the fact that *register* < *advance* < *multiply* < *divide*.

⟨ Declare subprocedures for *prefixed_command* 1215 ⟩ +≡

```
procedure do_register_command(a : small_number);
  label found, exit;
  var l, q, r, s: pointer; { for list manipulation }
    p: int_val .. mu_val; { type of register involved }
    e: boolean; { does l refer to a sparse array element? }
    w: integer; { integer or dimen value of l }
  begin q ← cur_cmd; e ← false; { just in case, will be set true for sparse array elements }
  ⟨ Compute the register location l and its type p; but return if invalid 1237* ⟩;
  if q = register then scan_optional_equals
  else if scan_keyword("by") then do_nothing; { optional `by' }
  arith_error ← false;
  if q < multiply then ⟨ Compute result of register or advance, put it in cur_val 1238* ⟩
  else ⟨ Compute result of multiply or divide, put it in cur_val 1240* ⟩;
  if arith_error then
    begin print_err("Arithmetic␣overflow");
    help2("I␣can´t␣carry␣out␣that␣multiplication␣or␣division,")
    ("since␣the␣result␣is␣out␣of␣range.");
    if p ≥ glue_val then delete_glue_ref(cur_val);
    error; return;
    end;
  if p < glue_val then sa_word_define(l, cur_val)
  else begin trap_zero_glue; sa_define(l, cur_val)(l, glue_ref, cur_val);
    end;
exit: end;
```

1237* Here we use the fact that the consecutive codes *int_val* .. *mu_val* and *assign_int* .. *assign_mu_glue* correspond to each other nicely.

```
⟨ Compute the register location l and its type p; but return if invalid 1237* ⟩ ≡
  begin if $q \neq register$ then
    begin get_x_token;
    if $(cur\_cmd \geq assign\_int) \land (cur\_cmd \leq assign\_mu\_glue)$ then
      begin $l \gets cur\_chr$; $p \gets cur\_cmd - assign\_int$; goto found;
      end;
    if $cur\_cmd \neq register$ then
      begin print_err("You␣can´t␣use␣`"); print_cmd_chr(cur_cmd, cur_chr); print("´␣after␣");
      print_cmd_chr(q, 0); help1("I´m␣forgetting␣what␣you␣said␣and␣not␣changing␣anything.");
      error; return;
      end;
    end;
  if $(cur\_chr < mem\_bot) \lor (cur\_chr > lo\_mem\_stat\_max)$ then
    begin $l \gets cur\_chr$; $p \gets sa\_type(l)$; $e \gets true$;
    end
  else begin $p \gets cur\_chr - mem\_bot$; scan_register_num;
    if $cur\_val > 255$ then
      begin find_sa_element(p, cur_val, true); $l \gets cur\_ptr$; $e \gets true$;
      end
    else case p of
      int_val: $l \gets cur\_val + count\_base$;
      dimen_val: $l \gets cur\_val + scaled\_base$;
      glue_val: $l \gets cur\_val + skip\_base$;
      mu_val: $l \gets cur\_val + mu\_skip\_base$;
      end; { there are no other cases }
    end;
  end;
found: if $p < glue\_val$ then if e then $w \gets sa\_int(l)$ else $w \gets eqtb[l].int$
  else if e then $s \gets sa\_ptr(l)$ else $s \gets equiv(l)$
```

This code is used in section 1236*.

1238* ⟨ Compute result of *register* or *advance*, put it in *cur_val* 1238* ⟩ ≡

```
  if $p < glue\_val$ then
    begin if $p = int\_val$ then scan_int else scan_normal_dimen;
    if $q = advance$ then $cur\_val \gets cur\_val + w$;
    end
  else begin scan_glue(p);
    if $q = advance$ then ⟨ Compute the sum of two glue specs 1239* ⟩;
    end
```

This code is used in section 1236*.

1239* ⟨Compute the sum of two glue specs 1239*⟩ ≡

```
begin q ← new_spec(cur_val); r ← s; delete_glue_ref(cur_val); width(q) ← width(q) + width(r);
if stretch(q) = 0 then stretch_order(q) ← normal;
if stretch_order(q) = stretch_order(r) then stretch(q) ← stretch(q) + stretch(r)
else if (stretch_order(q) < stretch_order(r)) ∧ (stretch(r) ≠ 0) then
    begin stretch(q) ← stretch(r); stretch_order(q) ← stretch_order(r);
    end;
if shrink(q) = 0 then shrink_order(q) ← normal;
if shrink_order(q) = shrink_order(r) then shrink(q) ← shrink(q) + shrink(r)
else if (shrink_order(q) < shrink_order(r)) ∧ (shrink(r) ≠ 0) then
    begin shrink(q) ← shrink(r); shrink_order(q) ← shrink_order(r);
    end;
cur_val ← q;
end
```

This code is used in section 1238*.

1240* ⟨Compute result of *multiply* or *divide*, put it in *cur_val* 1240*⟩ ≡

```
begin scan_int;
if p < glue_val then
  if q = multiply then
    if p = int_val then cur_val ← mult_integers(w, cur_val)
    else cur_val ← nx_plus_y(w, cur_val, 0)
  else cur_val ← x_over_n(w, cur_val)
else begin r ← new_spec(s);
  if q = multiply then
    begin width(r) ← nx_plus_y(width(s), cur_val, 0); stretch(r) ← nx_plus_y(stretch(s), cur_val, 0);
    shrink(r) ← nx_plus_y(shrink(s), cur_val, 0);
    end
  else begin width(r) ← x_over_n(width(s), cur_val); stretch(r) ← x_over_n(stretch(s), cur_val);
    shrink(r) ← x_over_n(shrink(s), cur_val);
    end;
  cur_val ← r;
  end;
end
```

This code is used in section 1236*.

1241* The processing of boxes is somewhat different, because we may need to scan and create an entire box before we actually change the value of the old one.

⟨Assignments 1217⟩ +≡

```
set_box: begin scan_register_num;
  if global then n ← global_box_flag + cur_val else n ← box_flag + cur_val;
  scan_optional_equals;
  if set_box_allowed then scan_box(n)
  else begin print_err("Improper␣"); print_esc("setbox");
    help2("Sorry,␣\setbox␣is␣not␣allowed␣after␣\halign␣in␣a␣display,")
    ("or␣between␣\accent␣and␣an␣accented␣character."); error;
    end;
  end;
```

1246* ⟨ Declare subprocedures for *prefixed_command* 1215 ⟩ +≡

```
procedure alter_integer;
  var c: small_number;   { 0 for \deadcycles, 1 for \insertpenalties, etc. }
  begin c ← cur_chr; scan_optional_equals; scan_int;
  if c = 0 then dead_cycles ← cur_val
  ⟨ Cases for alter_integer 1427* ⟩
else insert_penalties ← cur_val;
  end;
```

1247* ⟨ Declare subprocedures for *prefixed_command* 1215 ⟩ +≡

```
procedure alter_box_dimen;
  var c: small_number;   { width_offset or height_offset or depth_offset }
    b: pointer;   { box register }
  begin c ← cur_chr; scan_register_num; fetch_box(b); scan_optional_equals; scan_normal_dimen;
  if b ≠ null then mem[b + c].sc ← cur_val;
  end;
```

1248* Paragraph shapes are set up in the obvious way.

```
⟨ Assignments 1217 ⟩ +≡
set_shape: begin q ← cur_chr; scan_optional_equals; scan_int; n ← cur_val;
  if n ≤ 0 then p ← null
  else if q > par_shape_loc then
      begin n ← (cur_val div 2) + 1; p ← get_node(2 ∗ n + 1); info(p) ← n; n ← cur_val;
      mem[p + 1].int ← n;   { number of penalties }
      for j ← p + 2 to p + n + 1 do
        begin scan_int; mem[j].int ← cur_val;   { penalty values }
        end;
      if ¬odd(n) then mem[p + n + 2].int ← 0;   { unused }
      end
    else begin p ← get_node(2 ∗ n + 1); info(p) ← n;
      for j ← 1 to n do
        begin scan_normal_dimen; mem[p + 2 ∗ j − 1].sc ← cur_val;   { indentation }
        scan_normal_dimen; mem[p + 2 ∗ j].sc ← cur_val;   { width }
        end;
      end;
  define(q, shape_ref, p);
  end;
```

1257* ⟨ Declare subprocedures for *prefixed_command* 1215 ⟩ +≡

```
procedure new_font(a : small_number);
  label common_ending;
  var u: pointer;  { user's font identifier }
    s: scaled;  { stated "at" size, or negative of scaled magnification }
    f: internal_font_number;  { runs through existing fonts }
    t: str_number;  { name for the frozen font identifier }
    old_setting: 0 .. max_selector;  { holds selector setting }
    flushable_string: str_number;  { string not yet referenced }
  begin if $job\_name = 0$ then open_log_file;  { avoid confusing texput with the font name }
  get_r_token; $u \gets cur\_cs$;
  if $u \ge hash\_base$ then $t \gets text(u)$
  else if $u \ge single\_base$ then
      if $u = null\_cs$ then $t \gets$ "FONT" else $t \gets u - single\_base$
    else begin $old\_setting \gets selector$; $selector \gets new\_string$; print("FONT"); print($u - active\_base$);
      $selector \gets old\_setting$; str_room(1); $t \gets make\_string$;
      end;
  define(u, set_font, null_font); scan_optional_equals; scan_file_name;
  ⟨ Scan the font size specification 1258 ⟩;
  ⟨ If this font has already been loaded, set f to the internal font number and goto common_ending 1260 ⟩;
  $f \gets read\_font\_info(u, cur\_name, cur\_area, s)$;
common_ending: define(u, set_font, f); $eqtb[font\_id\_base + f] \gets eqtb[u]$; $font\_id\_text(f) \gets t$;
  end;
```

1292* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
xray: case chr_code of
  show_box_code: print_esc("showbox");
  show_the_code: print_esc("showthe");
  show_lists: print_esc("showlists");
    ⟨ Cases of xray for print_cmd_chr 1407* ⟩
  othercases print_esc("show")
  endcases;
```

1293* ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
procedure show_whatever;
  label common_ending;
  var p: pointer;  { tail of a token list to show }
    t: small_number;  { type of conditional being shown }
    m: normal .. or_code;  { upper bound on fi_or_else codes }
    l: integer;  { line where that conditional began }
    n: integer;  { level of \if...\fi nesting }
  begin case cur_chr of
  show_lists: begin begin_diagnostic; show_activities;
    end;
  show_box_code: ⟨ Show the current contents of a box 1296* ⟩;
  show_code: ⟨ Show the current meaning of a token, then goto common_ending 1294 ⟩;
    ⟨ Cases for show_whatever 1408* ⟩
  othercases ⟨ Show the current value of some parameter or register, then goto common_ending 1297 ⟩
  endcases;
  ⟨ Complete a potentially long \show command 1298 ⟩;
common_ending: if interaction < error_stop_mode then
    begin help0; decr(error_count);
    end
  else if tracing_online > 0 then
      begin
      help3("This␣isn´t␣an␣error␣message;␣I´m␣just␣\showing␣something.")
      ("Type␣`I\show...´␣to␣show␣more␣(e.g.,␣\show\cs,")
      ("\showthe\count10,␣\showbox255,␣\showlists).");
      end
    else begin
      help5("This␣isn´t␣an␣error␣message;␣I´m␣just␣\showing␣something.")
      ("Type␣`I\show...´␣to␣show␣more␣(e.g.,␣\show\cs,")
      ("\showthe\count10,␣\showbox255,␣\showlists).")
      ("And␣type␣`I\tracingonline=1\show...´␣to␣show␣boxes␣and")
      ("lists␣on␣your␣terminal␣as␣well␣as␣in␣the␣transcript␣file.");
      end;
  error;
  end;
```

1295* ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

```
undefined_cs: print("undefined");
call, long_call, outer_call, long_outer_call: begin n ← cmd − call;
  if info(link(chr_code)) = protected_token then n ← n + 4;
  if odd(n div 4) then print_esc("protected");
  if odd(n) then print_esc("long");
  if odd(n div 2) then print_esc("outer");
  if n > 0 then print_char("␣");
  print("macro");
  end;
end_template: print_esc("outer␣endtemplate");
```

1296* ⟨Show the current contents of a box 1296*⟩ ≡

```
begin scan_register_num; fetch_box(p); begin_diagnostic; print_nl(">␣\box"); print_int(cur_val);
print_char("=");
if p = null then print("void") else show_box(p);
end
```

This code is used in section 1293*.

1307*. The next few sections of the program should make it clear how we use the dump/undump macros.

```
⟨ Dump constants for consistency check 1307* ⟩ ≡
  dump_int(@$);
  ⟨ Dump the ε-TeX state 1385* ⟩
  dump_int(mem_bot);
  dump_int(mem_top);
  dump_int(eqtb_size);
  dump_int(hash_prime);
  dump_int(hyph_size)
```

This code is used in section 1302.

1308*. Sections of a `WEB` program that are "commented out" still contribute strings to the string pool; therefore `INITEX` and TeX will have the same strings. (And it is, of course, a good thing that they do.)

```
⟨ Undump constants for consistency check 1308* ⟩ ≡
  x ← fmt_file↑.int;
  if x ≠ @$ then goto bad_fmt;   { check that strings are the same }
  ⟨ Undump the ε-TeX state 1386* ⟩
  undump_int(x);
  if x ≠ mem_bot then goto bad_fmt;
  undump_int(x);
  if x ≠ mem_top then goto bad_fmt;
  undump_int(x);
  if x ≠ eqtb_size then goto bad_fmt;
  undump_int(x);
  if x ≠ hash_prime then goto bad_fmt;
  undump_int(x);
  if x ≠ hyph_size then goto bad_fmt
```

This code is used in section 1303.

1311* By sorting the list of available spaces in the variable-size portion of *mem*, we are usually able to get by without having to dump very much of the dynamic memory.

We recompute *var_used* and *dyn_used*, so that `INITEX` dumps valid information even when it has not been gathering statistics.

```
⟨ Dump the dynamic memory 1311* ⟩ ≡
  sort_avail; var_used ← 0; dump_int(lo_mem_max); dump_int(rover);
  if eTeX_ex then
    for k ← int_val to tok_val do dump_int(sa_root[k]);
  p ← mem_bot; q ← rover; x ← 0;
  repeat for k ← p to q + 1 do dump_wd(mem[k]);
    x ← x + q + 2 − p; var_used ← var_used + q − p; p ← q + node_size(q); q ← rlink(q);
  until q = rover;
  var_used ← var_used + lo_mem_max − p; dyn_used ← mem_end + 1 − hi_mem_min;
  for k ← p to lo_mem_max do dump_wd(mem[k]);
  x ← x + lo_mem_max + 1 − p; dump_int(hi_mem_min); dump_int(avail);
  for k ← hi_mem_min to mem_end do dump_wd(mem[k]);
  x ← x + mem_end + 1 − hi_mem_min; p ← avail;
  while p ≠ null do
    begin decr(dyn_used); p ← link(p);
    end;
  dump_int(var_used); dump_int(dyn_used); print_ln; print_int(x);
  print("␣memory␣locations␣dumped;␣current␣usage␣is␣"); print_int(var_used); print_char("&");
  print_int(dyn_used)
```

This code is used in section 1302.

1312* ⟨ Undump the dynamic memory 1312* ⟩ ≡

```
  undump(lo_mem_stat_max + 1000)(hi_mem_stat_min − 1)(lo_mem_max);
  undump(lo_mem_stat_max + 1)(lo_mem_max)(rover);
  if eTeX_ex then
    for k ← int_val to tok_val do undump(null)(lo_mem_max)(sa_root[k]);
  p ← mem_bot; q ← rover;
  repeat for k ← p to q + 1 do undump_wd(mem[k]);
    p ← q + node_size(q);
    if (p > lo_mem_max) ∨ ((q ≥ rlink(q)) ∧ (rlink(q) ≠ rover)) then goto bad_fmt;
    q ← rlink(q);
  until q = rover;
  for k ← p to lo_mem_max do undump_wd(mem[k]);
  if mem_min < mem_bot − 2 then   { make more low memory available }
    begin p ← llink(rover); q ← mem_min + 1; link(mem_min) ← null; info(mem_min) ← null;
        { we don't use the bottom word }
    rlink(p) ← q; llink(rover) ← q;
    rlink(q) ← rover; llink(q) ← p; link(q) ← empty_flag; node_size(q) ← mem_bot − q;
    end;
  undump(lo_mem_max + 1)(hi_mem_stat_min)(hi_mem_min); undump(null)(mem_top)(avail);
  mem_end ← mem_top;
  for k ← hi_mem_min to mem_end do undump_wd(mem[k]);
  undump_int(var_used); undump_int(dyn_used)
```

This code is used in section 1303.

1324* ⟨ Dump the hyphenation tables 1324* ⟩ ≡

```
dump_int(hyph_count);
for k ← 0 to hyph_size do
  if hyph_word[k] ≠ 0 then
    begin dump_int(k); dump_int(hyph_word[k]); dump_int(hyph_list[k]);
    end;
print_ln; print_int(hyph_count); print("␣hyphenation␣exception");
if hyph_count ≠ 1 then print_char("s");
if trie_not_ready then init_trie;
dump_int(trie_max); dump_int(hyph_start);
for k ← 0 to trie_max do dump_hh(trie[k]);
dump_int(trie_op_ptr);
for k ← 1 to trie_op_ptr do
  begin dump_int(hyf_distance[k]); dump_int(hyf_num[k]); dump_int(hyf_next[k]);
  end;
print_nl("Hyphenation␣trie␣of␣length␣"); print_int(trie_max); print("␣has␣");
print_int(trie_op_ptr); print("␣op");
if trie_op_ptr ≠ 1 then print_char("s");
print("␣out␣of␣"); print_int(trie_op_size);
for k ← 255 downto 0 do
  if trie_used[k] > min_quarterword then
    begin print_nl("␣␣"); print_int(qo(trie_used[k])); print("␣for␣language␣"); print_int(k);
    dump_int(k); dump_int(qo(trie_used[k]));
    end
```

This code is used in section 1302.

1325* Only "nonempty" parts of *op_start* need to be restored.

⟨ Undump the hyphenation tables 1325* ⟩ ≡

```
undump(0)(hyph_size)(hyph_count);
for k ← 1 to hyph_count do
  begin undump(0)(hyph_size)(j); undump(0)(str_ptr)(hyph_word[j]);
  undump(min_halfword)(max_halfword)(hyph_list[j]);
  end;
undump_size(0)(trie_size)(´trie␣size´)(j); init trie_max ← j; tiniundump(0)(j)(hyph_start);
for k ← 0 to j do undump_hh(trie[k]);
undump_size(0)(trie_op_size)(´trie␣op␣size´)(j); init trie_op_ptr ← j; tini
for k ← 1 to j do
  begin undump(0)(63)(hyf_distance[k]);   { a small_number }
  undump(0)(63)(hyf_num[k]); undump(min_quarterword)(max_quarterword)(hyf_next[k]);
  end;
init for k ← 0 to 255 do trie_used[k] ← min_quarterword;
tini
k ← 256;
while j > 0 do
  begin undump(0)(k − 1)(k); undump(1)(j)(x); init trie_used[k] ← qi(x); tini
  j ← j − x; op_start[k] ← qo(j);
  end;
init trie_not_ready ← false tini
```

This code is used in section 1303.

1335* We get to the *final_cleanup* routine when `\end` or `\dump` has been scanned and *its_all_over*.

```
⟨Last-minute procedures 1333⟩ +≡
procedure final_cleanup;
  label exit;
  var c: small_number;  {0 for \end, 1 for \dump}
  begin c ← cur_chr;
  if job_name = 0 then open_log_file;
  while input_ptr > 0 do
    if state = token_list then end_token_list else end_file_reading;
  while open_parens > 0 do
    begin print("␣)"); decr(open_parens);
    end;
  if cur_level > level_one then
    begin print_nl("("); print_esc("end␣occurred␣"); print("inside␣a␣group␣at␣level␣");
    print_int(cur_level − level_one); print_char(")");
    if eTeX_ex then show_save_groups;
    end;
  while cond_ptr ≠ null do
    begin print_nl("("); print_esc("end␣occurred␣"); print("when␣"); print_cmd_chr(if_test, cur_if);
    if if_line ≠ 0 then
      begin print("␣on␣line␣"); print_int(if_line);
      end;
    print("␣was␣incomplete)"); if_line ← if_line_field(cond_ptr); cur_if ← subtype(cond_ptr);
    temp_ptr ← cond_ptr; cond_ptr ← link(cond_ptr); free_node(temp_ptr, if_node_size);
    end;
  if history ≠ spotless then
    if ((history = warning_issued) ∨ (interaction < error_stop_mode)) then
      if selector = term_and_log then
        begin selector ← term_only;
        print_nl("(see␣the␣transcript␣file␣for␣additional␣information)");
        selector ← term_and_log;
        end;
  if c = 1 then
    begin init for c ← top_mark_code to split_bot_mark_code do
      if cur_mark[c] ≠ null then delete_token_ref(cur_mark[c]);
    if sa_mark ≠ null then
      if do_marks(destroy_marks, 0, sa_mark) then sa_mark ← null;
    for c ← last_box_code to vsplit_code do flush_node_list(disc_ptr[c]);
    if last_glue ≠ max_halfword then delete_glue_ref(last_glue);
    store_fmt_file; return; tini
    print_nl("(\dump␣is␣performed␣only␣by␣INITEX)"); return;
    end;
exit: end;
```

1336*

```
⟨Last-minute procedures 1333⟩ +≡
  init procedure init_prim;  {initialize all the primitives}
  begin no_new_control_sequence ← false; first ← 0;
  ⟨Put each of TEX's primitives into the hash table 226⟩;
  no_new_control_sequence ← true;
  end;
  tini
```

1337* When we begin the following code, TEX's tables may still contain garbage; the strings might not even be present. Thus we must proceed cautiously to get bootstrapped in.

But when we finish this part of the program, TEX is ready to call on the *main_control* routine to do its work.

```
⟨ Get the first line of input and prepare to start 1337* ⟩ ≡
  begin ⟨ Initialize the input routines 331* ⟩;
  ⟨ Enable ε-TEX, if requested 1379* ⟩
  if (format_ident = 0) ∨ (buffer[loc] = "&") then
    begin if format_ident ≠ 0 then initialize;  { erase preloaded format }
    if ¬open_fmt_file then goto final_end;
    if ¬load_fmt_file then
      begin w_close(fmt_file); goto final_end;
      end;
    w_close(fmt_file);
    while (loc < limit) ∧ (buffer[loc] = "␣") do incr(loc);
    end;
  if eTeX_ex then wterm_ln('entering␣extended␣mode');
  if end_line_char_inactive then decr(limit)
  else buffer[limit] ← end_line_char;
  fix_date_and_time;
  ⟨ Compute the magic offset 765 ⟩;
  ⟨ Initialize the print selector based on interaction 75 ⟩;
  if (loc < limit) ∧ (cat_code(buffer[loc]) ≠ escape) then start_input;  { \input assumed }
  end
```

This code is used in section 1332.

```
1362*  define adv_past(#) ≡ if subtype(#) = language_node then
          begin cur_lang ← what_lang(#); l_hyf ← what_lhm(#); r_hyf ← what_rhm(#); set_hyph_index;
          end
```

⟨ Advance past a whatsit node in the *line_break* loop 1362* ⟩ ≡ *adv_past*(*cur_p*)

This code is used in section 866*.

1379*. The extended features of ε-TEX. The program has two modes of operation: (1) In TEX compatibility mode it fully deserves the name TEX and there are neither extended features nor additional primitive commands. There are, however, a few modifications that would be legitimate in any implementation of TEX such as, e.g., preventing inadequate results of the glue to `DVI` unit conversion during *ship_out*. (2) In extended mode there are additional primitive commands and the extended features of ε-TEX are available.

The distinction between these two modes of operation initially takes place when a 'virgin' `eINITEX` starts without reading a format file. Later on the values of all ε-TEX state variables are inherited when `eVIRTEX` (or `eINITEX`) reads a format file.

The code below is designed to work for cases where '**init** . . . **tini**' is a run-time switch.

```
⟨Enable ε-TEX, if requested 1379*⟩ ≡
  init if (buffer[loc] = "*") ∧ (format_ident = "␣(INITEX)") then
    begin no_new_control_sequence ← false; ⟨Generate all ε-TEX primitives 1380*⟩
    incr(loc); eTeX_mode ← 1;   { enter extended mode }
    ⟨Initialize variables for ε-TEX extended mode 1548*⟩
    end;
  tini
  if ¬no_new_control_sequence then   { just entered extended mode ? }
    no_new_control_sequence ← true else
```

This code is used in section 1337*.

1380*. The ε-TEX features available in extended mode are grouped into two categories: (1) Some of them are permanently enabled and have no semantic effect as long as none of the additional primitives are executed. (2) The remaining ε-TEX features are optional and can be individually enabled and disabled. For each optional feature there is an ε-TEX state variable named `\...state`; the feature is enabled, resp. disabled by assigning a positive, resp. non-positive value to that integer.

```
  define eTeX_state_base = int_base + eTeX_state_code
  define eTeX_state(#) ≡ eqtb[eTeX_state_base + #].int   { an ε-TEX state variable }
  define eTeX_version_code = eTeX_int   { code for \eTeXversion }

⟨Generate all ε-TEX primitives 1380*⟩ ≡
  primitive("lastnodetype", last_item, last_node_type_code);
  primitive("eTeXversion", last_item, eTeX_version_code);
  primitive("eTeXrevision", convert, eTeX_revision_code);
```

See also sections 1388*, 1394*, 1397*, 1400*, 1403*, 1406*, 1415*, 1417*, 1420*, 1423*, 1428*, 1432*, 1482*, 1494*, 1497*, 1505*, 1513*, 1536*, 1540*, 1544*, 1596*, and 1599*.

This code is used in section 1379*.

1381*.

```
⟨Cases of last_item for print_cmd_chr 1381*⟩ ≡
last_node_type_code: print_esc("lastnodetype");
eTeX_version_code: print_esc("eTeXversion");
```

See also sections 1395*, 1398*, 1401*, 1404*, 1514*, 1537*, and 1541*.

This code is used in section 417*.

1382*.

```
⟨Cases for fetching an integer value 1382*⟩ ≡
eTeX_version_code: cur_val ← eTeX_version;
```

See also sections 1396*, 1399*, and 1538*.

This code is used in section 424*.

1383*.

```
  define eTeX_ex ≡ (eTeX_mode = 1)   { is this extended mode? }

⟨Global variables 13⟩ +≡
eTeX_mode: 0 . . 1;   { identifies compatibility and extended mode }
```

1384* ⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡

```
eTeX_mode ← 0;  { initially we are in compatibility mode }
⟨ Initialize variables for ε-TeX compatibility mode 1547* ⟩
```

1385* ⟨ Dump the ε-TeX state 1385* ⟩ ≡

```
dump_int(eTeX_mode);
for j ← 0 to eTeX_states − 1 do eTeX_state(j) ← 0;  { disable all enhancements }
```

See also section 1493*.

This code is used in section 1307*.

1386* ⟨ Undump the ε-TeX state 1386* ⟩ ≡

```
undump(0)(1)(eTeX_mode);
if eTeX_ex then
  begin ⟨ Initialize variables for ε-TeX extended mode 1548* ⟩
  end
else begin ⟨ Initialize variables for ε-TeX compatibility mode 1547* ⟩
  end;
```

This code is used in section 1308*.

1387* The *eTeX_enabled* function simply returns its first argument as result. This argument is *true* if an optional ε-TeX feature is currently enabled; otherwise, if the argument is *false*, the function gives an error message.

⟨ Declare ε-TeX procedures for use by *main_control* 1387* ⟩ ≡

```
function eTeX_enabled(b : boolean; j : quarterword; k : halfword): boolean;
  begin if ¬b then
    begin print_err("Improper␣"); print_cmd_chr(j, k);
    help1("Sorry,␣this␣optional␣e-TeX␣feature␣has␣been␣disabled."); error;
    end;
  eTeX_enabled ← b;
  end;
```

See also sections 1410* and 1426*.

This code is used in section 815*.

1388* First we implement the additional ε-TeX parameters in the table of equivalents.

⟨ Generate all ε-TeX primitives 1380* ⟩ +≡

```
primitive("everyeof", assign_toks, every_eof_loc);
primitive("tracingassigns", assign_int, int_base + tracing_assigns_code);
primitive("tracinggroups", assign_int, int_base + tracing_groups_code);
primitive("tracingifs", assign_int, int_base + tracing_ifs_code);
primitive("tracingscantokens", assign_int, int_base + tracing_scan_tokens_code);
primitive("tracingnesting", assign_int, int_base + tracing_nesting_code);
primitive("predisplaydirection", assign_int, int_base + pre_display_direction_code);
primitive("lastlinefit", assign_int, int_base + last_line_fit_code);
primitive("savingvdiscards", assign_int, int_base + saving_vdiscards_code);
primitive("savinghyphcodes", assign_int, int_base + saving_hyph_codes_code);
```

1389* **define** *every_eof* ≡ *equiv*(*every_eof_loc*)

⟨ Cases of *assign_toks* for *print_cmd_chr* 1389* ⟩ ≡

```
every_eof_loc: print_esc("everyeof");
```

This code is used in section 231*.

1390* ⟨ Cases for *print_param* 1390* ⟩ ≡
tracing_assigns_code: *print_esc*(`"tracingassigns"`);
tracing_groups_code: *print_esc*(`"tracinggroups"`);
tracing_ifs_code: *print_esc*(`"tracingifs"`);
tracing_scan_tokens_code: *print_esc*(`"tracingscantokens"`);
tracing_nesting_code: *print_esc*(`"tracingnesting"`);
pre_display_direction_code: *print_esc*(`"predisplaydirection"`);
last_line_fit_code: *print_esc*(`"lastlinefit"`);
saving_vdiscards_code: *print_esc*(`"savingvdiscards"`);
saving_hyph_codes_code: *print_esc*(`"savinghyphcodes"`);

See also section 1431*.

This code is used in section 237*.

1391* In order to handle `\everyeof` we need an array *eof_seen* of boolean variables.

⟨ Global variables 13 ⟩ +≡
eof_seen: **array** [1 . . *max_in_open*] **of** *boolean*; { has eof been seen? }

1392* The *print_group* procedure prints the current level of grouping and the name corresponding to *cur_group*.

```
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
procedure print_group(e : boolean);
  label exit;
  begin case cur_group of
  bottom_level: begin print("bottom␣level"); return;
    end;
  simple_group, semi_simple_group: begin if cur_group = semi_simple_group then print("semi␣");
    print("simple");
    end;
  hbox_group, adjusted_hbox_group: begin if cur_group = adjusted_hbox_group then print("adjusted␣");
    print("hbox");
    end;
  vbox_group: print("vbox");
  vtop_group: print("vtop");
  align_group, no_align_group: begin if cur_group = no_align_group then print("no␣");
    print("align");
    end;
  output_group: print("output");
  disc_group: print("disc");
  insert_group: print("insert");
  vcenter_group: print("vcenter");
  math_group, math_choice_group, math_shift_group, math_left_group: begin print("math");
    if cur_group = math_choice_group then print("␣choice")
    else if cur_group = math_shift_group then print("␣shift")
      else if cur_group = math_left_group then print("␣left");
    end;
  end;   { there are no other cases }
  print("␣group␣(level␣"); print_int(qo(cur_level)); print_char(")");
  if saved(−1) ≠ 0 then
    begin if e then print("␣entered␣at␣line␣")
    else print("␣at␣line␣");
    print_int(saved(−1));
    end;
exit: end;
```

1393* The *group_trace* procedure is called when a new level of grouping begins ($e = false$) or ends ($e = true$) with *saved*(−1) containing the line number.

```
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
  stat procedure group_trace(e : boolean);
  begin begin_diagnostic; print_char("{");
  if e then print("leaving␣")
  else print("entering␣");
  print_group(e); print_char("}"); end_diagnostic(false);
  end;
  tats
```

1394* The \currentgrouplevel and \currentgrouptype commands return the current level of grouping and the type of the current group respectively.

```
define current_group_level_code = eTeX_int + 1   { code for \currentgrouplevel }
define current_group_type_code = eTeX_int + 2   { code for \currentgrouptype }
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("currentgrouplevel", last_item, current_group_level_code);
  primitive("currentgrouptype", last_item, current_group_type_code);
```

1395*
```
⟨ Cases of last_item for print_cmd_chr 1381* ⟩ +≡
current_group_level_code: print_esc("currentgrouplevel");
current_group_type_code: print_esc("currentgrouptype");
```

1396*
```
⟨ Cases for fetching an integer value 1382* ⟩ +≡
current_group_level_code: cur_val ← cur_level − level_one;
current_group_type_code: cur_val ← cur_group;
```

1397* The \currentiflevel, \currentiftype, and \currentifbranch commands return the current level of conditionals and the type and branch of the current conditional.

```
define current_if_level_code = eTeX_int + 3   { code for \currentiflevel }
define current_if_type_code = eTeX_int + 4   { code for \currentiftype }
define current_if_branch_code = eTeX_int + 5   { code for \currentifbranch }
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("currentiflevel", last_item, current_if_level_code);
  primitive("currentiftype", last_item, current_if_type_code);
  primitive("currentifbranch", last_item, current_if_branch_code);
```

1398*
```
⟨ Cases of last_item for print_cmd_chr 1381* ⟩ +≡
current_if_level_code: print_esc("currentiflevel");
current_if_type_code: print_esc("currentiftype");
current_if_branch_code: print_esc("currentifbranch");
```

1399*
```
⟨ Cases for fetching an integer value 1382* ⟩ +≡
current_if_level_code: begin q ← cond_ptr; cur_val ← 0;
  while q ≠ null do
    begin incr(cur_val); q ← link(q);
    end;
  end;
current_if_type_code: if cond_ptr = null then cur_val ← 0
  else if cur_if < unless_code then cur_val ← cur_if + 1
    else cur_val ← −(cur_if − unless_code + 1);
current_if_branch_code: if (if_limit = or_code) ∨ (if_limit = else_code) then cur_val ← 1
  else if if_limit = fi_code then cur_val ← −1
    else cur_val ← 0;
```

1400* The `\fontcharwd`, `\fontcharht`, `\fontchardp`, and `\fontcharic` commands return information about a character in a font.

```
define font_char_wd_code = eTeX_dim   { code for \fontcharwd }
define font_char_ht_code = eTeX_dim + 1   { code for \fontcharht }
define font_char_dp_code = eTeX_dim + 2   { code for \fontchardp }
define font_char_ic_code = eTeX_dim + 3   { code for \fontcharic }
```

⟨ Generate all ε-TEX primitives 1380* ⟩ +≡

```
primitive("fontcharwd", last_item, font_char_wd_code);
primitive("fontcharht", last_item, font_char_ht_code);
primitive("fontchardp", last_item, font_char_dp_code);
primitive("fontcharic", last_item, font_char_ic_code);
```

1401* ⟨ Cases of *last_item* for *print_cmd_chr* 1381* ⟩ +≡

```
font_char_wd_code: print_esc("fontcharwd");
font_char_ht_code: print_esc("fontcharht");
font_char_dp_code: print_esc("fontchardp");
font_char_ic_code: print_esc("fontcharic");
```

1402* ⟨ Cases for fetching a dimension value 1402* ⟩ ≡

```
font_char_wd_code, font_char_ht_code, font_char_dp_code, font_char_ic_code: begin scan_font_ident;
  q ← cur_val; scan_char_num;
  if (font_bc[q] ≤ cur_val) ∧ (font_ec[q] ≥ cur_val) then
    begin i ← char_info(q)(qi(cur_val));
    case m of
    font_char_wd_code: cur_val ← char_width(q)(i);
    font_char_ht_code: cur_val ← char_height(q)(height_depth(i));
    font_char_dp_code: cur_val ← char_depth(q)(height_depth(i));
    font_char_ic_code: cur_val ← char_italic(q)(i);
    end;   { there are no other cases }
    end
  else cur_val ← 0;
  end;
```

See also sections 1405* and 1539*.

This code is used in section 424*.

1403* The `\parshapedimen`, `\parshapeindent`, and `\parshapelength` commands return the indent and length parameters of the current `\parshape` specification.

```
define par_shape_length_code = eTeX_dim + 4   { code for \parshapelength }
define par_shape_indent_code = eTeX_dim + 5   { code for \parshapeindent }
define par_shape_dimen_code = eTeX_dim + 6   { code for \parshapedimen }
```

⟨ Generate all ε-TEX primitives 1380* ⟩ +≡

```
primitive("parshapelength", last_item, par_shape_length_code);
primitive("parshapeindent", last_item, par_shape_indent_code);
primitive("parshapedimen", last_item, par_shape_dimen_code);
```

1404* ⟨ Cases of *last_item* for *print_cmd_chr* 1381* ⟩ +≡

```
par_shape_length_code: print_esc("parshapelength");
par_shape_indent_code: print_esc("parshapeindent");
par_shape_dimen_code: print_esc("parshapedimen");
```

1405*. ⟨Cases for fetching a dimension value 1402*⟩ +≡

```
par_shape_length_code, par_shape_indent_code, par_shape_dimen_code: begin
      q ← cur_chr − par_shape_length_code; scan_int;
  if (par_shape_ptr = null) ∨ (cur_val ≤ 0) then cur_val ← 0
  else begin if q = 2 then
      begin q ← cur_val mod 2; cur_val ← (cur_val + q) div 2;
      end;
    if cur_val > info(par_shape_ptr) then cur_val ← info(par_shape_ptr);
    cur_val ← mem[par_shape_ptr + 2 ∗ cur_val − q].sc;
    end;
  cur_val_level ← dimen_val;
  end;
```

1406*. The `\showgroups` command displays all currently active grouping levels.

```
  define show_groups = 4  { \showgroups }
⟨Generate all ε-TEX primitives 1380*⟩ +≡
  primitive("showgroups", xray, show_groups);
```

1407*. ⟨Cases of *xray* for *print_cmd_chr* 1407*⟩ ≡

```
show_groups: print_esc("showgroups");
```

See also sections 1416* and 1421*.

This code is used in section 1292*.

1408*. ⟨Cases for *show_whatever* 1408*⟩ ≡

```
show_groups: begin begin_diagnostic; show_save_groups;
  end;
```

See also section 1422*.

This code is used in section 1293*.

1409*. ⟨Types in the outer block 18⟩ +≡

```
  save_pointer = 0 .. save_size;  { index into save_stack }
```

1410* The modifications of TEX required for the display produced by the *show_save_groups* procedure were first discussed by Donald E. Knuth in *TUGboat* **11**, 165–170 and 499–511, 1990.

In order to understand a group type we also have to know its mode. Since unrestricted horizontal modes are not associated with grouping, they are skipped when traversing the semantic nest.

```
⟨ Declare ε-TEX procedures for use by main_control 1387* ⟩ +≡
procedure show_save_groups;
  label found1, found2, found, done;
  var p: 0 .. nest_size;  { index into nest }
    m: −mmode .. mmode;  { mode }
    v: save_pointer;  { saved value of save_ptr }
    l: quarterword;  { saved value of cur_level }
    c: group_code;  { saved value of cur_group }
    a: −1 .. 1;  { to keep track of alignments }
    i: integer; j: quarterword; s: str_number;
  begin p ← nest_ptr; nest[p] ← cur_list;  { put the top level into the array }
  v ← save_ptr; l ← cur_level; c ← cur_group; save_ptr ← cur_boundary; decr(cur_level);
  a ← 1; print_nl(""); print_ln;
  loop begin print_nl("###␣"); print_group(true);
    if cur_group = bottom_level then goto done;
    repeat m ← nest[p].mode_field;
      if p > 0 then decr(p)
      else m ← vmode;
    until m ≠ hmode;
    print("␣(");
    case cur_group of
    simple_group: begin incr(p); goto found2;
      end;
    hbox_group, adjusted_hbox_group: s ← "hbox";
    vbox_group: s ← "vbox";
    vtop_group: s ← "vtop";
    align_group: if a = 0 then
        begin if m = −vmode then s ← "halign"
        else s ← "valign";
        a ← 1; goto found1;
        end
      else begin if a = 1 then print("align␣entry")
        else print_esc("cr");
        if p ≥ a then p ← p − a;
        a ← 0; goto found;
        end;
    no_align_group: begin incr(p); a ← −1; print_esc("noalign"); goto found2;
      end;
    output_group: begin print_esc("output"); goto found;
      end;
    math_group: goto found2;
    disc_group, math_choice_group: begin if cur_group = disc_group then print_esc("discretionary")
      else print_esc("mathchoice");
      for i ← 1 to 3 do
        if i ≤ saved(−2) then print("{}");
      goto found2;
      end;
    insert_group: begin if saved(−2) = 255 then print_esc("vadjust")
```

```
      else begin print_esc("insert"); print_int(saved(−2));
        end;
      goto found2;
      end;
    vcenter_group: begin s ← "vcenter"; goto found1;
      end;
    semi_simple_group: begin incr(p); print_esc("begingroup"); goto found;
      end;
    math_shift_group: begin if m = mmode then print_char("$")
      else if nest[p].mode_field = mmode then
          begin print_cmd_chr(eq_no, saved(−2)); goto found;
          end;
      print_char("$"); goto found;
      end;
    math_left_group: begin if type(nest[p + 1].eTeX_aux_field) = left_noad then print_esc("left")
      else print_esc("middle");
      goto found;
      end;
    end; { there are no other cases }
    ⟨Show the box context 1412*⟩;
  found1: print_esc(s); ⟨Show the box packaging info 1411*⟩;
  found2: print_char("{");
  found: print_char(")"); decr(cur_level); cur_group ← save_level(save_ptr);
    save_ptr ← save_index(save_ptr)
    end;
done: save_ptr ← v; cur_level ← l; cur_group ← c;
  end;
```

1411* ⟨Show the box packaging info 1411*⟩ ≡

```
  if saved(−2) ≠ 0 then
    begin print_char("␣");
    if saved(−3) = exactly then print("to")
    else print("spread");
    print_scaled(saved(−2)); print("pt");
    end
```

This code is used in section 1410*.

1412* ⟨ Show the box context 1412* ⟩ ≡

```
i ← saved(−4);
if i ≠ 0 then
  if i < box_flag then
    begin if abs(nest[p].mode_field) = vmode then j ← hmove
    else j ← vmove;
    if i > 0 then print_cmd_chr(j, 0)
    else print_cmd_chr(j, 1);
    print_scaled(abs(i)); print("pt");
    end
  else if i < ship_out_flag then
      begin if i ≥ global_box_flag then
        begin print_esc("global"); i ← i − (global_box_flag − box_flag);
        end;
      print_esc("setbox"); print_int(i − box_flag); print_char("=");
      end
    else print_cmd_chr(leader_ship, i − (leader_flag − a_leaders))
```

This code is used in section 1410*.

1413* The *scan_general_text* procedure is much like *scan_toks*(*false*, *false*), but will be invoked via *expand*, i.e., recursively.

⟨ Declare ε-TEX procedures for scanning 1413* ⟩ ≡
procedure *scan_general_text*; *forward*;

See also sections 1507*, 1516*, and 1521*.

This code is used in section 409*.

1414* The token list (balanced text) created by *scan_general_text* begins at *link*(*temp_head*) and ends at *cur_val*. (If *cur_val* = *temp_head*, the list is empty.)

```
⟨ Declare ε-TEX procedures for token lists 1414* ⟩ ≡
procedure scan_general_text;
  label found;
  var s: normal .. absorbing;   { to save scanner_status }
    w: pointer;   { to save warning_index }
    d: pointer;   { to save def_ref }
    p: pointer;   { tail of the token list being built }
    q: pointer;   { new node being added to the token list via store_new_token }
    unbalance: halfword;   { number of unmatched left braces }
  begin s ← scanner_status; w ← warning_index; d ← def_ref; scanner_status ← absorbing;
  warning_index ← cur_cs; def_ref ← get_avail; token_ref_count(def_ref) ← null; p ← def_ref;
  scan_left_brace;   { remove the compulsory left brace }
  unbalance ← 1;
  loop begin get_token;
    if cur_tok < right_brace_limit then
      if cur_cmd < right_brace then incr(unbalance)
      else begin decr(unbalance);
        if unbalance = 0 then goto found;
        end;
    store_new_token(cur_tok);
    end;
found: q ← link(def_ref); free_avail(def_ref);   { discard reference count }
  if q = null then cur_val ← temp_head else cur_val ← p;
  link(temp_head) ← q; scanner_status ← s; warning_index ← w; def_ref ← d;
  end;
```

See also section 1488*.

This code is used in section 464*.

1415* The `\showtokens` command displays a token list.

```
  define show_tokens = 5   { \showtokens , must be odd! }
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("showtokens", xray, show_tokens);
```

1416*
```
⟨ Cases of xray for print_cmd_chr 1407* ⟩ +≡
show_tokens: print_esc("showtokens");
```

1417* The `\unexpanded` primitive prevents expansion of tokens much as the result from `\the` applied to a token variable. The `\detokenize` primitive converts a token list into a list of character tokens much as if the token list were written to a file. We use the fact that the command modifiers for `\unexpanded` and `\detokenize` are odd whereas those for `\the` and `\showthe` are even.

```
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("unexpanded", the, 1);
  primitive("detokenize", the, show_tokens);
```

1418*
```
⟨ Cases of the for print_cmd_chr 1418* ⟩ ≡
else if chr_code = 1 then print_esc("unexpanded")
  else print_esc("detokenize")
```

This code is used in section 266*.

1419* ⟨ Handle `\unexpanded` or `\detokenize` and **return** 1419* ⟩ ≡

```
if odd(cur_chr) then
  begin c ← cur_chr; scan_general_text;
  if c = 1 then the_toks ← cur_val
  else begin old_setting ← selector; selector ← new_string; b ← pool_ptr; p ← get_avail;
    link(p) ← link(temp_head); token_show(p); flush_list(p); selector ← old_setting;
    the_toks ← str_toks(b);
    end;
  return;
  end
```

This code is used in section 465*.

1420* The `\showifs` command displays all currently active conditionals.

```
define show_ifs = 6   { \showifs }
```

⟨ Generate all ε-TEX primitives 1380* ⟩ +≡

```
primitive("showifs", xray, show_ifs);
```

1421* ⟨ Cases of *xray* for *print_cmd_chr* 1407* ⟩ +≡

```
show_ifs: print_esc("showifs");
```

1422*

```
define print_if_line(#) ≡
      if # ≠ 0 then
        begin print("␣entered␣on␣line␣"); print_int(#);
        end
```

⟨ Cases for *show_whatever* 1408* ⟩ +≡

```
show_ifs: begin begin_diagnostic; print_nl(""); print_ln;
  if cond_ptr = null then
    begin print_nl("###␣"); print("no␣active␣conditionals");
    end
  else begin p ← cond_ptr; n ← 0;
    repeat incr(n); p ← link(p); until p = null;
    p ← cond_ptr; t ← cur_if; l ← if_line; m ← if_limit;
    repeat print_nl("###␣level␣"); print_int(n); print(":␣"); print_cmd_chr(if_test, t);
      if m = fi_code then print_esc("else");
      print_if_line(l); decr(n); t ← subtype(p); l ← if_line_field(p); m ← type(p); p ← link(p);
    until p = null;
    end;
  end;
```

1423* The `\interactionmode` primitive allows to query and set the interaction mode.

⟨ Generate all ε-TEX primitives 1380* ⟩ +≡

```
primitive("interactionmode", set_page_int, 2);
```

1424* ⟨ Cases of *set_page_int* for *print_cmd_chr* 1424* ⟩ ≡

```
else if chr_code = 2 then print_esc("interactionmode")
```

This code is used in section 417*.

1425* ⟨ Cases for 'Fetch the *dead_cycles* or the *insert_penalties*' 1425* ⟩ ≡

```
else if m = 2 then cur_val ← interaction
```

This code is used in section 419*.

1426* ⟨ Declare ε-TEX procedures for use by *main_control* 1387* ⟩ +≡

```
procedure new_interaction; forward;
```

1427* ⟨ Cases for *alter_integer* 1427* ⟩ ≡

```
else if $c = 2$ then
    begin if $(cur\_val < batch\_mode) \lor (cur\_val > error\_stop\_mode)$ then
      begin print_err("Bad␣interaction␣mode");
      help2("Modes␣are␣0=batch,␣1=nonstop,␣2=scroll,␣and")
      ("3=errorstop.␣Proceed,␣and␣I´ll␣ignore␣this␣case."); int_error(cur_val);
      end
    else begin cur_chr ← cur_val; new_interaction;
      end;
    end
```

This code is used in section 1246*.

1428* The *middle* feature of ε-TEX allows one ore several `\middle` delimiters to appear between `\left` and `\right`.

⟨ Generate all ε-TEX primitives 1380* ⟩ +≡

```
  primitive("middle", left_right, middle_noad);
```

1429* ⟨ Cases of *left_right* for *print_cmd_chr* 1429* ⟩ ≡

```
else if chr_code = middle_noad then print_esc("middle")
```

This code is used in section 1189*.

1430* In constructions such as

```
\hbox to \hsize{
     \hskip 0pt plus 0.0001fil
     ...
     \hfil\penalty-200\hfilneg
     ...}
```

the stretch components of `\hfil` and `\hfilneg` compensate; they may, however, get modified in order to prevent arithmetic overflow during *hlist_out* when each of them is multiplied by a large *glue_set* value.

Since this "glue rounding" depends on state variables *cur_g* and *cur_glue* and TeX--XeT is supposed to emulate the behaviour of TeX-XeT (plus a suitable postprocessor) as close as possible the glue rounding cannot be postponed until (segments of) an hlist has been reversed.

The code below is invoked after the effective width, *rule_wd*, of a glue node has been computed. The glue node is either converted into a kern node or, for leaders, the glue specification is replaced by an equivalent rigid one; the subtype of the glue node remains unchanged.

```
⟨ Handle a glue node for mixed direction typesetting 1430* ⟩ ≡
  if $(((g\_sign = stretching) \land (stretch\_order(g) = g\_order)) \lor ((g\_sign = shrinking) \land (shrink\_order(g) =$
        $g\_order)))$ then
    begin $fast\_delete\_glue\_ref(g)$;
    if $subtype(p) < a\_leaders$ then
      begin $type(p) \gets kern\_node$; $width(p) \gets rule\_wd$;
      end
    else begin $g \gets get\_node(glue\_spec\_size)$;
      $stretch\_order(g) \gets filll + 1$; $shrink\_order(g) \gets filll + 1$;   { will never match }
      $width(g) \gets rule\_wd$; $stretch(g) \gets 0$; $shrink(g) \gets 0$; $glue\_ptr(p) \gets g$;
      end;
    end
```

This code is used in sections 625* and 1461*.

1431* The optional *TeXXeT* feature of ε-TeX contains the code for mixed left-to-right and right-to-left typesetting. This code is inspired by but different from TeX-XeT as presented by Donald E. Knuth and Pierre MacKay in *TUGboat* **8**, 14–25, 1987.

In order to avoid confusion with TeX-XeT the present implementation of mixed direction typesetting is called TeX--XeT. It differs from TeX-XeT in several important aspects: (1) Right-to-left text is reversed explicitly by the *ship_out* routine and is written to a normal `DVI` file without any *begin_reflect* or *end_reflect* commands; (2) a *math_node* is (ab)used instead of a *whatsit_node* to record the `\beginL`, `\endL`, `\beginR`, and `\endR` text direction primitives in order to keep the influence on the line breaking algorithm for pure left-to-right text as small as possible; (3) right-to-left text interrupted by a displayed equation is automatically resumed after that equation; and (4) the *valign* command code with a non-zero command modifier is (ab)used for the text direction primitives.

Nevertheless there is a subtle difference between TeX and TeX--XeT that may influence the line breaking algorithm for pure left-to-right text. When a paragraph containing math mode material is broken into lines TeX may generate lines where math mode material is not enclosed by properly nested `\mathon` and `\mathoff` nodes. Unboxing such lines as part of a new paragraph may have the effect that hyphenation is attempted for 'words' originating from math mode or that hyphenation is inhibited for words originating from horizontal mode.

In TeX--XeT additional `\beginM`, resp. `\endM` math nodes are supplied at the start, resp. end of lines such that math mode material inside a horizontal list always starts with either `\mathon` or `\beginM` and ends with `\mathoff` or `\endM`. These additional nodes are transparent to operations such as `\unskip`, `\lastpenalty`, or `\lastbox` but they do have the effect that hyphenation is never attempted for 'words' originating from math mode and is never inhibited for words originating from horizontal mode.

```
  define TeXXeT_state ≡ eTeX_state(TeXXeT_code)
  define TeXXeT_en ≡ (TeXXeT_state > 0)   { is TeX--XeT enabled? }
⟨ Cases for print_param 1390* ⟩ +≡
eTeX_state_code + TeXXeT_code: print_esc("TeXXeTstate");
```

1432* ⟨ Generate all ε-TeX primitives 1380* ⟩ +≡

```
  primitive("TeXXeTstate", assign_int, eTeX_state_base + TeXXeT_code);
  primitive("beginL", valign, begin_L_code); primitive("endL", valign, end_L_code);
  primitive("beginR", valign, begin_R_code); primitive("endR", valign, end_R_code);
```

1433* ⟨ Cases of *valign* for *print_cmd_chr* 1433* ⟩ ≡

```
else case chr_code of
  begin_L_code: print_esc("beginL");
  end_L_code: print_esc("endL");
  begin_R_code: print_esc("beginR");
  othercases print_esc("endR")
  endcases
```

This code is used in section 266*.

1434* ⟨ Cases of *main_control* for *hmode* + *valign* 1434* ⟩ ≡

```
  if cur_chr > 0 then
    begin if eTeX_enabled(TeXXeT_en, cur_cmd, cur_chr) then tail_append(new_math(0, cur_chr));
    end
  else
```

This code is used in section 1130*.

1435* An hbox with subtype dlist will never be reversed, even when embedded in right-to-left text.

```
⟨ Display if this box is never to be reversed 1435* ⟩ ≡
  if $(type(p) = hlist\_node) \land (box\_lr(p) = dlist)$ then $print($",␣display"$)$
```

This code is used in section 184*.

1436* A number of routines are based on a stack of one-word nodes whose *info* fields contain *end_M_code*, *end_L_code*, or *end_R_code*. The top of the stack is pointed to by *LR_ptr*.

When the stack manipulation macros of this section are used below, variable *LR_ptr* might be the global variable declared here for *hpack* and *ship_out*, or might be local to *post_line_break*.

```
  define $put\_LR(\#) \equiv$
          begin $temp\_ptr \gets get\_avail$; $info(temp\_ptr) \gets \#$; $link(temp\_ptr) \gets LR\_ptr$;
          $LR\_ptr \gets temp\_ptr$;
          end
  define $push\_LR(\#) \equiv put\_LR(end\_LR\_type(\#))$
  define $pop\_LR \equiv$
          begin $temp\_ptr \gets LR\_ptr$; $LR\_ptr \gets link(temp\_ptr)$; $free\_avail(temp\_ptr)$;
          end

⟨ Global variables 13 ⟩ +≡
$LR\_ptr$: $pointer$;  { stack of LR codes for hpack, ship_out, and init_math }
$LR\_problems$: $integer$;  { counts missing begins and ends }
$cur\_dir$: $small\_number$;  { current text direction }
```

1437* ⟨ Set initial values of key variables 21 ⟩ +≡

```
  $LR\_ptr \gets null$; $LR\_problems \gets 0$; $cur\_dir \gets left\_to\_right$;
```

1438* ⟨ Insert LR nodes at the beginning of the current line and adjust the LR stack based on LR nodes in this line 1438* ⟩ ≡

```
  begin $q \gets link(temp\_head)$;
  if $LR\_ptr \neq null$ then
    begin $temp\_ptr \gets LR\_ptr$; $r \gets q$;
    repeat $s \gets new\_math(0, begin\_LR\_type(info(temp\_ptr)))$; $link(s) \gets r$; $r \gets s$;
      $temp\_ptr \gets link(temp\_ptr)$;
    until $temp\_ptr = null$;
    $link(temp\_head) \gets r$;
    end;
  while $q \neq cur\_break(cur\_p)$ do
    begin if $\neg is\_char\_node(q)$ then
      if $type(q) = math\_node$ then ⟨ Adjust the LR stack for the post_line_break routine 1439* ⟩;
    $q \gets link(q)$;
    end;
  end
```

This code is used in section 880*.

1439* ⟨ Adjust the LR stack for the *post_line_break* routine 1439* ⟩ ≡

```
  if $end\_LR(q)$ then
    begin if $LR\_ptr \neq null$ then
      if $info(LR\_ptr) = end\_LR\_type(q)$ then $pop\_LR$;
    end
  else $push\_LR(q)$
```

This code is used in sections 879*, 881*, and 1438*.

1440* We use the fact that q now points to the node with `\rightskip` glue.

```
⟨Insert LR nodes at the end of the current line 1440*⟩ ≡
  if LR_ptr ≠ null then
    begin s ← temp_head; r ← link(s);
    while r ≠ q do
      begin s ← r; r ← link(s);
      end;
    r ← LR_ptr;
    while r ≠ null do
      begin temp_ptr ← new_math(0, info(r)); link(s) ← temp_ptr; s ← temp_ptr; r ← link(r);
      end;
    link(s) ← q;
    end
```

This code is used in section 880*.

1441*

```
⟨Initialize the LR stack 1441*⟩ ≡
  put_LR(before)   { this will never match }
```

This code is used in sections 649*, 1445*, and 1469*.

1442*

```
⟨Adjust the LR stack for the hpack routine 1442*⟩ ≡
  if end_LR(p) then
    if info(LR_ptr) = end_LR_type(p) then pop_LR
    else begin incr(LR_problems); type(p) ← kern_node; subtype(p) ← explicit;
      end
  else push_LR(p)
```

This code is used in section 651*.

1443*

```
⟨Check for LR anomalies at the end of hpack 1443*⟩ ≡
  begin if info(LR_ptr) ≠ before then
    begin while link(q) ≠ null do q ← link(q);
    repeat temp_ptr ← q; q ← new_math(0, info(LR_ptr)); link(temp_ptr) ← q;
      LR_problems ← LR_problems + 10000; pop_LR;
    until info(LR_ptr) = before;
    end;
  if LR_problems > 0 then
    begin ⟨Report LR problems 1444*⟩;
    goto common_ending;
    end;
  pop_LR;
  if LR_ptr ≠ null then confusion("LR1");
  end
```

This code is used in section 649*.

1444*

```
⟨Report LR problems 1444*⟩ ≡
  begin print_ln; print_nl("\endL␣or␣\endR␣problem␣(");
  print_int(LR_problems div 10000); print("␣missing,␣");
  print_int(LR_problems mod 10000); print("␣extra");
  LR_problems ← 0;
  end
```

This code is used in sections 1443* and 1465*.

1445*. ⟨ Initialize *hlist_out* for mixed direction typesetting 1445* ⟩ ≡

```
if eTeX_ex then
  begin ⟨ Initialize the LR stack 1441* ⟩;
  if box_lr(this_box) = dlist then
    if cur_dir = right_to_left then
      begin cur_dir ← left_to_right; cur_h ← cur_h − width(this_box);
      end
    else set_box_lr(this_box)(0);
  if (cur_dir = right_to_left) ∧ (box_lr(this_box) ≠ reversed) then
    ⟨ Reverse the complete hlist and set the subtype to reversed 1452* ⟩;
  end
```

This code is used in section 619*.

1446*. ⟨ Finish *hlist_out* for mixed direction typesetting 1446* ⟩ ≡

```
if eTeX_ex then
  begin ⟨ Check for LR anomalies at the end of hlist_out 1449* ⟩;
  if box_lr(this_box) = dlist then cur_dir ← right_to_left;
  end
```

This code is used in section 619*.

1447*. ⟨ Handle a math node in *hlist_out* 1447* ⟩ ≡

```
begin if eTeX_ex then ⟨ Adjust the LR stack for the hlist_out routine; if necessary reverse an hlist
        segment and goto reswitch 1448* ⟩;
cur_h ← cur_h + width(p);
end
```

This code is used in section 622*.

1448*. Breaking a paragraph into lines while TEX--XƎT is disabled may result in lines whith unpaired math nodes. Such hlists are silently accepted in the absence of text direction directives.

define *LR_dir*(#) ≡ (*subtype*(#) **div** *R_code*) { text direction of a 'math node' }

⟨ Adjust the LR stack for the *hlist_out* routine; if necessary reverse an hlist segment and **goto** *reswitch* 1448* ⟩ ≡

```
begin if end_LR(p) then
  if info(LR_ptr) = end_LR_type(p) then pop_LR
  else begin if subtype(p) > L_code then incr(LR_problems);
    end
else begin push_LR(p);
  if LR_dir(p) ≠ cur_dir then ⟨ Reverse an hlist segment and goto reswitch 1453* ⟩;
  end;
type(p) ← kern_node;
end
```

This code is used in section 1447*.

1449*. ⟨ Check for LR anomalies at the end of *hlist_out* 1449* ⟩ ≡

```
begin while info(LR_ptr) ≠ before do
  begin if info(LR_ptr) > L_code then LR_problems ← LR_problems + 10000;
  pop_LR;
  end;
pop_LR;
end
```

This code is used in section 1446*.

1450* **define** *edge_node* = *style_node* { a *style_node* does not occur in hlists }

```
  define edge_node_size = style_node_size   { number of words in an edge node }
  define edge_dist(#) ≡ depth(#)
            { new left_edge position relative to cur_h (after width has been taken into account) }
⟨ Declare procedures needed in hlist_out, vlist_out 1368 ⟩ +≡
function new_edge(s : small_number; w : scaled): pointer;   { create an edge node }
  var p: pointer;   { the new node }
  begin p ← get_node(edge_node_size); type(p) ← edge_node; subtype(p) ← s; width(p) ← w;
  edge_dist(p) ← 0;   { the edge_dist field will be set later }
  new_edge ← p;
  end;
```

1451* ⟨ Cases of *hlist_out* that arise in mixed direction text only 1451* ⟩ ≡

```
edge_node: begin cur_h ← cur_h + width(p); left_edge ← cur_h + edge_dist(p); cur_dir ← subtype(p);
  end;
```

This code is used in section 622*.

1452* We detach the hlist, start a new one consisting of just one kern node, append the reversed list, and set the width of the kern node.

```
⟨ Reverse the complete hlist and set the subtype to reversed 1452* ⟩ ≡
  begin save_h ← cur_h; temp_ptr ← p; p ← new_kern(0); link(prev_p) ← p; cur_h ← 0;
  link(p) ← reverse(this_box, null, cur_g, cur_glue); width(p) ← −cur_h; cur_h ← save_h;
  set_box_lr(this_box)(reversed);
  end
```

This code is used in section 1445*.

1453* We detach the remainder of the hlist, replace the math node by an edge node, and append the reversed hlist segment to it; the tail of the reversed segment is another edge node and the remainder of the original list is attached to it.

```
⟨ Reverse an hlist segment and goto reswitch 1453* ⟩ ≡
  begin save_h ← cur_h; temp_ptr ← link(p); rule_wd ← width(p); free_node(p, small_node_size);
  cur_dir ← reflected; p ← new_edge(cur_dir, rule_wd); link(prev_p) ← p;
  cur_h ← cur_h − left_edge + rule_wd; link(p) ← reverse(this_box, new_edge(reflected, 0), cur_g, cur_glue);
  edge_dist(p) ← cur_h; cur_dir ← reflected; cur_h ← save_h; goto reswitch;
  end
```

This code is used in section 1448*.

1454* OLD VERSION. The *reverse* function defined here is responsible to reverse the nodes of an hlist (segment). The first parameter *this_box* is the enclosing hlist node, the second parameter *t* is to become the tail of the reversed list, and the global variable *temp_ptr* is the head of the list to be reversed. Finally *cur_g* and *cur_glue* are the current glue rounding state variables, to be updated by this function. We remove nodes from the original list and add them to the head of the new one.

```
⟨Declare procedures needed in hlist_out, vlist_out 1368⟩ +≡
function reverse(this_box, t : pointer; var cur_g : scaled; var cur_glue : real): pointer;
  label reswitch, next_p, done;
  var l: pointer; { the new list }
    p: pointer; { the current node }
    q: pointer; { the next node }
    g_order: glue_ord; { applicable order of infinity for glue }
    g_sign: normal .. shrinking; { selects type of glue }
    glue_temp: real; { glue value before rounding }
    m, n: halfword; { count of unmatched math nodes }
  begin g_order ← glue_order(this_box); g_sign ← glue_sign(this_box); l ← t; p ← temp_ptr;
  m ← min_halfword; n ← min_halfword;
  loop begin while $p \neq null$ do ⟨Move node p to the new list and go to the next node; or goto done if
          the end of the reflected segment has been reached 1459*⟩;
    if $(t = null) \land (m = min\_halfword) \land (n = min\_halfword)$ then goto done;
    p ← new_math(0, info(LR_ptr)); LR_problems ← LR_problems + 10000;
        { manufacture one missing math node }
    end;
done: reverse ← l;
  end;
```

1455* NEW VERSION. The *reverse* function defined here is responsible to reverse (parts of) the nodes of an hlist. The first parameter *this_box* is the enclosing hlist node, the second parameter *t* is to become the tail of the reversed list, and the global variable *temp_ptr* is the head of the list to be reversed. Finally *cur_g* and *cur_glue* are the current glue rounding state variables, to be updated by this function.

```
⟨ Declare procedures needed in hlist_out, vlist_out 1368 ⟩ +≡
  @{Declare subprocedures for reverse 1456* ⟩
function reverse(this_box, t : pointer; var cur_g : scaled; var cur_glue : real): pointer;
  label reswitch, next_p, done;
  var l: pointer;   { the new list }
    p: pointer;   { the current node }
    q: pointer;   { the next node }
    g_order: glue_ord;   { applicable order of infinity for glue }
    g_sign: normal .. shrinking;   { selects type of glue }
    glue_temp: real;   { glue value before rounding }
    m, n: halfword;   { count of unmatched math nodes }
  begin g_order ← glue_order(this_box); g_sign ← glue_sign(this_box);
  ⟨ Build a list of segments and determine their widths 1457* ⟩;
  l ← t; p ← temp_ptr; m ← min_halfword; n ← min_halfword;
  loop begin while p ≠ null do ⟨ Move node p to the new list and go to the next node; or goto done if
            the end of the reflected segment has been reached 1459* ⟩;
    if (t = null) ∧ (m = min_halfword) ∧ (n = min_halfword) then goto done;
    p ← new_math(0, info(LR_ptr)); LR_problems ← LR_problems + 10000;
        { manufacture one missing math node }
    end;
done: reverse ← l;
  end; @}
```

1456* We cannot simply remove nodes from the original list and add them to the head of the new one; this might reverse the order of whatsit nodes such that, e.g., a *write_node* for a stream appears before the *open_node* and/or after the *close_node* for that stream.

All whatsit nodes as well as hlist and vlist nodes containing such nodes must not be permuted. A sequence of hlist and vlist nodes not containing whatsit nodes as well as char, ligature, rule, kern, and glue nodes together with math nodes not changing the text direction can be explicitly reversed. Embedded sections of left-to-right text are treated as a unit and all remaining nodes are irrelevant and can be ignored.

In a first step we determine the width of various segments of the hlist to be reversed: (1) embedded left-to-right text, (2) sequences of permutable or irrelevant nodes, (3) sequences of whatsit or irrelevant nodes, and (4) individual hlist and vlist nodes containing whatsit nodes.

```
  define segment_node = style_node
  define segment_node_size = style_node_size   { number of words in a segment node }
  define segment_first(#) ≡ info(# + 2)   { first node of the segment }
  define segment_last(#) ≡ link(# + 2)   { last node of the segment }
⟨ Declare subprocedures for reverse 1456* ⟩ ≡
function new_segment(s : small_number; f : pointer): pointer;   { create a segment node }
  var p: pointer;   { the new node }
  begin p ← get_node(segment_node_size); type(p) ← segment_node; subtype(p) ← s; width(p) ← 0;
      { the width field will be set later }
  segment_first(p) ← f; segment_last(p) ← f; new_segment ← p;
  end;
```

See also section 1458*.

This code is used in section 1455*.

1457* ⟨ Build a list of segments and determine their widths 1457* ⟩ ≡

```
  begin end
```

This code is used in section 1455*.

1458* Here is a recursive subroutine that determines if the hlist or vlist node p contains whatsit nodes.

```
⟨ Declare subprocedures for reverse 1456* ⟩ +≡
function has_whatsit(p : pointer): boolean;
  label exit;
  begin p ← list_ptr(p); has_whatsit ← true;
  while p ≠ null do
    begin if ¬is_char_node(p) then
      case type(p) of
      hlist_node, vlist_node: if has_whatsit(p) then goto exit;
      whatsit_node: goto exit;
      othercases do_nothing
      endcases;
    p ← link(p);
    end;
  has_whatsit ← false;
exit: end;
```

1459* ⟨ Move node p to the new list and go to the next node; or **goto** *done* if the end of the reflected segment has been reached 1459* ⟩ ≡

```
reswitch: if is_char_node(p) then
    repeat f ← font(p); c ← character(p); cur_h ← cur_h + char_width(f)(char_info(f)(c)); q ← link(p);
      link(p) ← l; l ← p; p ← q;
    until ¬is_char_node(p)
  else ⟨ Move the non-char_node p to the new list 1460* ⟩
```

This code is used in sections 1454* and 1455*.

1460* ⟨ Move the non-*char_node* p to the new list 1460* ⟩ ≡

```
  begin q ← link(p);
  case type(p) of
  hlist_node, vlist_node, rule_node, kern_node: rule_wd ← width(p);
  ⟨ Cases of reverse that need special treatment 1461* ⟩
  edge_node: confusion("LR2");
  othercases goto next_p
  endcases;
  cur_h ← cur_h + rule_wd;
next_p: link(p) ← l;
  if type(p) = kern_node then
    if (rule_wd = 0) ∨ (l = null) then
      begin free_node(p, small_node_size); p ← l;
      end;
  l ← p; p ← q;
  end
```

This code is used in section 1459*.

1461*. Here we compute the effective width of a glue node as in *hlist_out*.

```
⟨ Cases of reverse that need special treatment 1461* ⟩ ≡
glue_node: begin round_glue; ⟨ Handle a glue node for mixed direction typesetting 1430* ⟩;
  end;
```

See also sections 1462* and 1463*.

This code is used in section 1460*.

1462*. A ligature node is replaced by a char node.

```
⟨ Cases of reverse that need special treatment 1461* ⟩ +≡
ligature_node: begin flush_node_list(lig_ptr(p)); temp_ptr ← p; p ← get_avail;
  mem[p] ← mem[lig_char(temp_ptr)]; link(p) ← q; free_node(temp_ptr, small_node_size); goto reswitch;
  end;
```

1463*. Math nodes in an inner reflected segment are modified, those at the outer level are changed into kern nodes.

```
⟨ Cases of reverse that need special treatment 1461* ⟩ +≡
math_node: begin rule_wd ← width(p);
  if end_LR(p) then
    if info(LR_ptr) ≠ end_LR_type(p) then
      begin type(p) ← kern_node; incr(LR_problems);
      end
    else begin pop_LR;
      if n > min_halfword then
        begin decr(n); decr(subtype(p));  { change after into before }
        end
      else begin type(p) ← kern_node;
        if m > min_halfword then decr(m)
        else ⟨ Finish the reversed hlist segment and goto done 1464* ⟩;
        end;
      end
  else begin push_LR(p);
    if (n > min_halfword) ∨ (LR_dir(p) ≠ cur_dir) then
      begin incr(n); incr(subtype(p));  { change before into after }
      end
    else begin type(p) ← kern_node; incr(m);
      end;
    end;
  end;
```

1464*. Finally we have found the end of the hlist segment to be reversed; the final math node is released and the remaining list attached to the edge node terminating the reversed segment.

```
⟨ Finish the reversed hlist segment and goto done 1464* ⟩ ≡
  begin free_node(p, small_node_size); link(t) ← q; width(t) ← rule_wd; edge_dist(t) ← −cur_h − rule_wd;
  goto done;
  end
```

This code is used in section 1463*.

1465* ⟨ Check for LR anomalies at the end of *ship_out* 1465* ⟩ ≡

```
begin if $LR\_problems > 0$ then
  begin ⟨ Report LR problems 1444* ⟩;
  print_char(")"); print_ln;
  end;
if $(LR\_ptr \neq null) \lor (cur\_dir \neq left\_to\_right)$ then confusion("LR3");
end
```

This code is used in section 638*.

1466* Some special actions are required for displayed equation in paragraphs with mixed direction texts. First of all we have to set the text direction preceding the display.

⟨ Set the value of x to the text direction before the display 1466* ⟩ ≡

```
if $LR\_save = null$ then $x \gets 0$
else if $info(LR\_save) \geq R\_code$ then $x \gets -1$ else $x \gets 1$
```

This code is used in sections 1467* and 1469*.

1467* ⟨ Prepare for display after an empty paragraph 1467* ⟩ ≡

```
begin pop_nest; ⟨ Set the value of $x$ to the text direction before the display 1466* ⟩;
end
```

This code is used in section 1145*.

1468* When calculating the natural width, w, of the final line preceding the display, we may have to copy all or part of its hlist. We copy, however, only those parts of the original list that are relevant for the computation of *pre_display_size*.

```
⟨ Declare subprocedures for init_math 1468* ⟩ ≡
procedure just_copy(p, h, t : pointer);
  label found, not_found;
  var r: pointer;  { current node being fabricated for new list }
    words: 0 .. 5;  { number of words remaining to be copied }
  begin while p ≠ null do
    begin words ← 1;  { this setting occurs in more branches than any other }
    if is_char_node(p) then r ← get_avail
    else case type(p) of
      hlist_node, vlist_node: begin r ← get_node(box_node_size); mem[r + 6] ← mem[p + 6];
        mem[r + 5] ← mem[p + 5];  { copy the last two words }
        words ← 5; list_ptr(r) ← null;  { this affects mem[r + 5] }
        end;
      rule_node: begin r ← get_node(rule_node_size); words ← rule_node_size;
        end;
      ligature_node: begin r ← get_avail;  { only font and character are needed }
        mem[r] ← mem[lig_char(p)]; goto found;
        end;
      kern_node, math_node: begin r ← get_node(small_node_size); words ← small_node_size;
        end;
      glue_node: begin r ← get_node(small_node_size); add_glue_ref(glue_ptr(p));
        glue_ptr(r) ← glue_ptr(p); leader_ptr(r) ← null;
        end;
      whatsit_node: ⟨ Make a partial copy of the whatsit node p and make r point to it; set words to the
            number of initial words not yet copied 1357 ⟩;
      othercases goto not_found
      endcases;
    while words > 0 do
      begin decr(words); mem[r + words] ← mem[p + words];
      end;
  found: link(h) ← r; h ← r;
  not_found: p ← link(p);
    end;
  link(h) ← t;
  end;
```

See also section 1473*.

This code is used in section 1138*.

1469* When the final line ends with R-text, the value w refers to the line reflected with respect to the left edge of the enclosing vertical list.

```
⟨Prepare for display after a non-empty paragraph 1469*⟩ ≡
  if eTeX_ex then ⟨Let j be the prototype box for the display 1475*⟩;
  $v \gets$ shift_amount(just_box); ⟨Set the value of x to the text direction before the display 1466*⟩;
  if $x \ge 0$ then
    begin $p \gets$ list_ptr(just_box); link(temp_head) $\gets$ null;
    end
  else begin $v \gets -v -$ width(just_box); $p \gets$ new_math(0, begin_L_code); link(temp_head) $\gets p$;
    just_copy(list_ptr(just_box), p, new_math(0, end_L_code)); cur_dir $\gets$ right_to_left;
    end;
  $v \gets v + 2 *$ quad(cur_font);
  if TeXXeT_en then ⟨Initialize the LR stack 1441*⟩
```

This code is used in section 1146*.

1470*

```
⟨Finish the natural width computation 1470*⟩ ≡
  if TeXXeT_en then
    begin while LR_ptr $\neq$ null do pop_LR;
    if LR_problems $\neq 0$ then
      begin $w \gets$ max_dimen; LR_problems $\gets 0$;
      end;
    end;
  cur_dir $\gets$ left_to_right; flush_node_list(link(temp_head))
```

This code is used in section 1146*.

1471* In the presence of text direction directives we assume that any LR problems have been fixed by the *hpack* routine. If the final line contains, however, text direction directives while TEX--XET is disabled, then we set $w \gets$ *max_dimen*.

```
⟨Cases of 'Let d be the natural width' that need special treatment 1471*⟩ ≡
math_node: begin $d \gets$ width(p);
  if TeXXeT_en then ⟨Adjust the LR stack for the init_math routine 1472*⟩
  else if subtype(p) $\ge$ L_code then
      begin $w \gets$ max_dimen; goto done;
      end;
  end;
edge_node: begin $d \gets$ width(p); cur_dir $\gets$ subtype(p);
  end;
```

This code is used in section 1147*.

1472* ⟨ Adjust the LR stack for the *init_math* routine 1472* ⟩ ≡

```
  if end_LR(p) then
    begin if info(LR_ptr) = end_LR_type(p) then pop_LR
    else if subtype(p) > L_code then
        begin w ← max_dimen; goto done;
        end
    end
  else begin push_LR(p);
    if LR_dir(p) ≠ cur_dir then
      begin just_reverse(p); p ← temp_head;
      end;
    end
```

This code is used in section 1471*.

1473* ⟨ Declare subprocedures for *init_math* 1468* ⟩ +≡

```
procedure just_reverse(p : pointer);
  label found, done;
  var l: pointer; { the new list }
    t: pointer; { tail of reversed segment }
    q: pointer; { the next node }
    m, n: halfword; { count of unmatched math nodes }
  begin m ← min_halfword; n ← min_halfword;
  if link(temp_head) = null then
    begin just_copy(link(p), temp_head, null); q ← link(temp_head);
    end
  else begin q ← link(p); link(p) ← null; flush_node_list(link(temp_head));
    end;
  t ← new_edge(cur_dir, 0); l ← t; cur_dir ← reflected;
  while q ≠ null do
    if is_char_node(q) then
      repeat p ← q; q ← link(p); link(p) ← l; l ← p;
      until ¬is_char_node(q)
    else begin p ← q; q ← link(p);
      if type(p) = math_node then ⟨ Adjust the LR stack for the just_reverse routine 1474* ⟩;
      link(p) ← l; l ← p;
      end;
  goto done;
found: width(t) ← width(p); link(t) ← q; free_node(p, small_node_size);
done: link(temp_head) ← l;
  end;
```

1474* ⟨Adjust the LR stack for the *just_reverse* routine 1474*⟩ ≡

```
if $end\_LR(p)$ then
  if $info(LR\_ptr) \neq end\_LR\_type(p)$ then
    begin $type(p) \gets kern\_node$; $incr(LR\_problems)$;
    end
  else begin $pop\_LR$;
    if $n > min\_halfword$ then
      begin $decr(n)$; $decr(subtype(p))$; { change after into before }
      end
    else begin if $m > min\_halfword$ then $decr(m)$ else goto found;
      $type(p) \gets kern\_node$;
      end;
    end
else begin $push\_LR(p)$;
  if $(n > min\_halfword) \lor (LR\_dir(p) \neq cur\_dir)$ then
    begin $incr(n)$; $incr(subtype(p))$; { change before into after }
    end
  else begin $type(p) \gets kern\_node$; $incr(m)$;
    end;
  end
```

This code is used in section 1473*.

1475* The prototype box is an hlist node with the width, glue set, and shift amount of *just_box*, i.e., the last line preceding the display. Its hlist reflects the current `\leftskip` and `\rightskip`.

⟨Let *j* be the prototype box for the display 1475*⟩ ≡

```
begin if $right\_skip = zero\_glue$ then $j \gets new\_kern(0)$
else $j \gets new\_param\_glue(right\_skip\_code)$;
if $left\_skip = zero\_glue$ then $p \gets new\_kern(0)$
else $p \gets new\_param\_glue(left\_skip\_code)$;
$link(p) \gets j$; $j \gets new\_null\_box$; $width(j) \gets width(just\_box)$; $shift\_amount(j) \gets shift\_amount(just\_box)$;
$list\_ptr(j) \gets p$; $glue\_order(j) \gets glue\_order(just\_box)$; $glue\_sign(j) \gets glue\_sign(just\_box)$;
$glue\_set(j) \gets glue\_set(just\_box)$;
end
```

This code is used in section 1469*.

1476* At the end of a displayed equation we retrieve the prototype box.

⟨Local variables for finishing a displayed formula 1198⟩ +≡
j: *pointer*; { prototype box }

1477* ⟨Retrieve the prototype box 1477*⟩ ≡

```
if $mode = mmode$ then $j \gets LR\_box$
```

This code is used in sections 1194* and 1194*.

1478* ⟨Flush the prototype box 1478*⟩ ≡

```
$flush\_node\_list(j)$
```

This code is used in section 1199*.

1479* The *app_display* procedure used to append the displayed equation and/or equation number to the current vertical list has three parameters: the prototype box, the hbox to be appended, and the displacement of the hbox in the display line.

```
⟨ Declare subprocedures for after_math 1479* ⟩ ≡
procedure app_display(j, b : pointer; d : scaled);
  var z: scaled;  { width of the line }
    s: scaled;  { move the line right this much }
    e: scaled;  { distance from right edge of box to end of line }
    x: integer;  { pre_display_direction }
    p, q, r, t, u: pointer;  { for list manipulation }
  begin s ← display_indent; x ← pre_display_direction;
  if x = 0 then shift_amount(b) ← s + d
  else begin z ← display_width; p ← b; ⟨ Set up the hlist for the display line 1480* ⟩;
    ⟨ Package the display line 1481* ⟩;
    end;
  append_to_vlist(b);
  end;
```

This code is used in section 1194*.

1480* Here we construct the hlist for the display, starting with node p and ending with node q. We also set d and e to the amount of kerning to be added before and after the hlist (adjusted for the prototype box).

```
⟨ Set up the hlist for the display line 1480* ⟩ ≡
  if $x > 0$ then $e ← z − d − width(p)$
  else begin $e ← d$; $d ← z − e − width(p)$;
    end;
  if $j ≠ null$ then
    begin b ← copy_node_list(j); height(b) ← height(p); depth(b) ← depth(p); s ← s − shift_amount(b);
    d ← d + s; e ← e + width(b) − z − s;
    end;
  if box_lr(p) = dlist then q ← p  { display or equation number }
  else begin  { display and equation number }
    r ← list_ptr(p); free_node(p, box_node_size);
    if r = null then confusion("LR4");
    if x > 0 then
      begin p ← r;
      repeat q ← r; r ← link(r);  { find tail of list }
      until r = null;
      end
    else begin p ← null; q ← r;
      repeat t ← link(r); link(r) ← p; p ← r; r ← t;  { reverse list }
      until r = null;
      end;
    end
```

This code is used in section 1479*.

1481* In the presence of a prototype box we use its shift amount and width to adjust the values of kerning and add these values to the glue nodes inserted to cancel the `\leftskip` and `\rightskip`. If there is no prototype box (because the display is preceded by an empty paragraph), or if the skip parameters are zero, we just add kerns.

The *cancel_glue* macro creates and links a glue node that is, together with another glue node, equivalent to a given amount of kerning. We can use j as temporary pointer, since all we need is $j \neq null$.

```
define cancel_glue(#) ≡ j ← new_skip_param(#); cancel_glue_cont
define cancel_glue_cont(#) ≡ link(#) ← j; cancel_glue_cont_cont
define cancel_glue_cont_cont(#) ≡ link(j) ← #; cancel_glue_end
define cancel_glue_end(#) ≡ j ← glue_ptr(#); cancel_glue_end_end
define cancel_glue_end_end(#) ≡ stretch_order(temp_ptr) ← stretch_order(j);
      shrink_order(temp_ptr) ← shrink_order(j); width(temp_ptr) ← # − width(j);
      stretch(temp_ptr) ← −stretch(j); shrink(temp_ptr) ← −shrink(j)
```

```
⟨Package the display line 1481*⟩ ≡
  if j = null then
    begin r ← new_kern(0); t ← new_kern(0);  { the widths will be set later }
    end
  else begin r ← list_ptr(b); t ← link(r);
    end;
  u ← new_math(0, end_M_code);
  if type(t) = glue_node then  { t is \rightskip glue }
    begin cancel_glue(right_skip_code)(q)(u)(t)(e); link(u) ← t;
    end
  else begin width(t) ← e; link(t) ← u; link(q) ← t;
    end;
  u ← new_math(0, begin_M_code);
  if type(r) = glue_node then  { r is \leftskip glue }
    begin cancel_glue(left_skip_code)(u)(p)(r)(d); link(r) ← u;
    end
  else begin width(r) ← d; link(r) ← p; link(u) ← r;
    if j = null then
      begin b ← hpack(u, natural); shift_amount(b) ← s;
      end
    else list_ptr(b) ← u;
    end
```

This code is used in section 1479*.

1482* The *scan_tokens* feature of ε-TEX defines the `\scantokens` primitive.

```
⟨Generate all ε-TEX primitives 1380*⟩ +≡
  primitive("scantokens", input, 2);
```

1483* ⟨Cases of *input* for *print_cmd_chr* 1483*⟩ ≡

```
else if chr_code = 2 then print_esc("scantokens")
```

This code is used in section 377*.

1484* ⟨Cases for *input* 1484*⟩ ≡

```
else if cur_chr = 2 then pseudo_start
```

This code is used in section 378*.

1485* The global variable *pseudo_files* is used to maintain a stack of pseudo files. The *info* field of each pseudo file points to a linked list of variable size nodes representing lines not yet processed: the *info* field of the first word contains the size of this node, all the following words contain ASCII codes.

```
⟨ Global variables 13 ⟩ +≡
pseudo_files: pointer;   { stack of pseudo files }
```

1486* ⟨ Set initial values of key variables 21 ⟩ +≡

```
  pseudo_files ← null;
```

1487* The *pseudo_start* procedure initiates reading from a pseudo file.

```
⟨ Declare ε-TEX procedures for expanding 1487* ⟩ ≡
procedure pseudo_start; forward;
```

See also sections 1545*, 1550*, and 1554*.

This code is used in section 366*.

1488* ⟨ Declare ε-TEX procedures for token lists 1414* ⟩ +≡

```
procedure pseudo_start;
  var old_setting: 0 .. max_selector;   { holds selector setting }
    s: str_number;   { string to be converted into a pseudo file }
    l, m: pool_pointer;   { indices into str_pool }
    p, q, r: pointer;   { for list construction }
    w: four_quarters;   { four ASCII codes }
    nl, sz: integer;
  begin scan_general_text; old_setting ← selector; selector ← new_string; token_show(temp_head);
  selector ← old_setting; flush_list(link(temp_head)); str_room(1); s ← make_string;
  ⟨ Convert string s into a new pseudo file 1489* ⟩;
  flush_string; ⟨ Initiate input from new pseudo file 1490* ⟩;
  end;
```

1489* ⟨Convert string s into a new pseudo file 1489*⟩ ≡

```
str_pool[pool_ptr] ← si("␣"); l ← str_start[s]; nl ← si(new_line_char); p ← get_avail; q ← p;
while l < pool_ptr do
  begin m ← l;
  while (l < pool_ptr) ∧ (str_pool[l] ≠ nl) do incr(l);
  sz ← (l − m + 7) div 4;
  if sz = 1 then sz ← 2;
  r ← get_node(sz); link(q) ← r; q ← r; info(q) ← hi(sz);
  while sz > 2 do
    begin decr(sz); incr(r); w.b0 ← qi(so(str_pool[m])); w.b1 ← qi(so(str_pool[m + 1]));
    w.b2 ← qi(so(str_pool[m + 2])); w.b3 ← qi(so(str_pool[m + 3])); mem[r].qqqq ← w; m ← m + 4;
    end;
  w.b0 ← qi("␣"); w.b1 ← qi("␣"); w.b2 ← qi("␣"); w.b3 ← qi("␣");
  if l > m then
    begin w.b0 ← qi(so(str_pool[m]));
    if l > m + 1 then
      begin w.b1 ← qi(so(str_pool[m + 1]));
      if l > m + 2 then
        begin w.b2 ← qi(so(str_pool[m + 2]));
        if l > m + 3 then w.b3 ← qi(so(str_pool[m + 3]));
        end;
      end;
    end;
  mem[r + 1].qqqq ← w;
  if str_pool[l] = nl then incr(l);
  end;
info(p) ← link(p); link(p) ← pseudo_files; pseudo_files ← p
```

This code is used in section 1488*.

1490* ⟨Initiate input from new pseudo file 1490*⟩ ≡

```
begin_file_reading; { set up cur_file and new level of input }
line ← 0; limit ← start; loc ← limit + 1; { force line read }
if tracing_scan_tokens > 0 then
  begin if term_offset > max_print_line − 3 then print_ln
  else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
  name ← 19; print("(␣"); incr(open_parens); update_terminal;
  end
else name ← 18
```

This code is used in section 1488*.

1491* Here we read a line from the current pseudo file into *buffer*.

```
⟨Declare ε-TEX procedures for tracing and input 284*⟩ +≡
function pseudo_input: boolean;  { inputs the next line or returns false }
  var p: pointer;  { current line from pseudo file }
    sz: integer;  { size of node p }
    w: four_quarters;  { four ASCII codes }
    r: pointer;  { loop index }
  begin last ← first;  { cf. Matthew 19 : 30 }
  p ← info(pseudo_files);
  if p = null then pseudo_input ← false
  else begin info(pseudo_files) ← link(p); sz ← ho(info(p));
    if 4 ∗ sz − 3 ≥ buf_size − last then ⟨Report overflow of the input buffer, and abort 35⟩;
    last ← first;
    for r ← p + 1 to p + sz − 1 do
      begin w ← mem[r].qqqq; buffer[last] ← w.b0; buffer[last + 1] ← w.b1; buffer[last + 2] ← w.b2;
      buffer[last + 3] ← w.b3; last ← last + 4;
      end;
    if last ≥ max_buf_stack then max_buf_stack ← last + 1;
    while (last > first) ∧ (buffer[last − 1] = "␣") do decr(last);
    free_node(p, sz); pseudo_input ← true;
    end;
  end;
```

1492* When we are done with a pseudo file we 'close' it.

```
⟨Declare ε-TEX procedures for tracing and input 284*⟩ +≡
procedure pseudo_close;  { close the top level pseudo file }
  var p, q: pointer;
  begin p ← link(pseudo_files); q ← info(pseudo_files); free_avail(pseudo_files); pseudo_files ← p;
  while q ≠ null do
    begin p ← q; q ← link(p); free_node(p, ho(info(p)));
    end;
  end;
```

1493* ⟨Dump the ε-TEX state 1385*⟩ +≡

```
  while pseudo_files ≠ null do pseudo_close;  { flush pseudo files }
```

1494* ⟨Generate all ε-TEX primitives 1380*⟩ +≡

```
  primitive("readline", read_to_cs, 1);
```

1495* ⟨Cases of *read* for *print_cmd_chr* 1495*⟩ ≡

```
else print_esc("readline")
```

This code is used in section 266*.

1496*. ⟨ Handle \readline and **goto** *done* 1496* ⟩ ≡

```
if j = 1 then
  begin while loc ≤ limit do  { current line not yet finished }
    begin cur_chr ← buffer[loc]; incr(loc);
    if cur_chr = "␣" then cur_tok ← space_token else cur_tok ← cur_chr + other_token;
    store_new_token(cur_tok);
    end;
  goto done;
  end
```

This code is used in section 483*.

1497*. Here we define the additional conditionals of ε-TEX as well as the \unless prefix.

```
define if_def_code = 17  { '\ifdefined' }
define if_cs_code = 18  { '\ifcsname' }
define if_font_char_code = 19  { '\iffontchar' }
```

⟨ Generate all ε-TEX primitives 1380* ⟩ +≡

```
primitive("unless", expand_after, 1);
primitive("ifdefined", if_test, if_def_code); primitive("ifcsname", if_test, if_cs_code);
primitive("iffontchar", if_test, if_font_char_code);
```

1498*. ⟨ Cases of *expandafter* for *print_cmd_chr* 1498* ⟩ ≡

```
else print_esc("unless")
```

This code is used in section 266*.

1499*. ⟨ Cases of *if_test* for *print_cmd_chr* 1499* ⟩ ≡

```
if_def_code: print_esc("ifdefined");
if_cs_code: print_esc("ifcsname");
if_font_char_code: print_esc("iffontchar");
```

This code is used in section 488*.

1500*. The result of a boolean condition is reversed when the conditional is preceded by \unless.

⟨ Negate a boolean conditional and **goto** *reswitch* 1500* ⟩ ≡

```
begin get_token;
if (cur_cmd = if_test) ∧ (cur_chr ≠ if_case_code) then
  begin cur_chr ← cur_chr + unless_code; goto reswitch;
  end;
print_err("You␣can´t␣use␣`"); print_esc("unless"); print("´␣before␣`");
print_cmd_chr(cur_cmd, cur_chr); print_char("´");
help1("Continue,␣and␣I´ll␣forget␣that␣it␣ever␣happened."); back_error;
end
```

This code is used in section 367*.

1501*. The conditional \ifdefined tests if a control sequence is defined.

We need to reset *scanner_status*, since \outer control sequences are allowed, but we might be scanning a macro definition or preamble.

⟨ Cases for *conditional* 1501* ⟩ ≡

```
if_def_code: begin save_scanner_status ← scanner_status; scanner_status ← normal; get_next;
  b ← (cur_cmd ≠ undefined_cs); scanner_status ← save_scanner_status;
  end;
```

See also sections 1502* and 1504*.

This code is used in section 501*.

1502*. The conditional `\ifcsname` is equivalent to `{\expandafter }\expandafter \ifdefined \csname`, except that no new control sequence will be entered into the hash table (once all tokens preceding the mandatory `\endcsname` have been expanded).

```
⟨ Cases for conditional 1501* ⟩ +≡
if_cs_code: begin $n \gets get\_avail$; $p \gets n$;   { head of the list of characters }
  repeat get_x_token;
    if $cur\_cs = 0$ then store_new_token(cur_tok);
  until $cur\_cs \neq 0$;
  if $cur\_cmd \neq end\_cs\_name$ then ⟨ Complain about missing \endcsname 373 ⟩;
  ⟨ Look up the characters of list $n$ in the hash table, and set cur_cs 1503* ⟩;
  flush_list(n); $b \gets (eq\_type(cur\_cs) \neq undefined\_cs)$;
  end;
```

1503*.

```
⟨ Look up the characters of list $n$ in the hash table, and set cur_cs 1503* ⟩ ≡
  $m \gets first$; $p \gets link(n)$;
  while $p \neq null$ do
    begin if $m \geq max\_buf\_stack$ then
      begin $max\_buf\_stack \gets m + 1$;
      if $max\_buf\_stack = buf\_size$ then overflow("buffer␣size", buf_size);
      end;
    $buffer[m] \gets info(p)$ mod '400; incr(m); $p \gets link(p)$;
    end;
  if $m > first + 1$ then $cur\_cs \gets id\_lookup(first, m - first)$   { no_new_control_sequence is true }
  else if $m = first$ then $cur\_cs \gets null\_cs$   { the list is empty }
    else $cur\_cs \gets single\_base + buffer[first]$   { the list has length one }
```

This code is used in section 1502*.

1504*. The conditional `\iffontchar` tests the existence of a character in a font.

```
⟨ Cases for conditional 1501* ⟩ +≡
if_font_char_code: begin scan_font_ident; $n \gets cur\_val$; scan_char_num;
  if $(font\_bc[n] \leq cur\_val) \land (font\_ec[n] \geq cur\_val)$ then $b \gets char\_exists(char\_info(n)(qi(cur\_val)))$
  else $b \gets false$;
  end;
```

1505*. The *protected* feature of ε-TEX defines the `\protected` prefix command for macro definitions. Such macros are protected against expansions when lists of expanded tokens are built, e.g., for `\edef` or during `\write`.

```
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("protected", prefix, 8);
```

1506*.

```
⟨ Cases of prefix for print_cmd_chr 1506* ⟩ ≡
else if $chr\_code = 8$ then print_esc("protected")
```

This code is used in section 1209*.

1507*. The *get_x_or_protected* procedure is like *get_x_token* except that protected macros are not expanded.

```
⟨ Declare ε-TEX procedures for scanning 1413* ⟩ +≡
procedure get_x_or_protected; { sets cur_cmd, cur_chr, cur_tok, and expands non-protected macros }
  label exit;
  begin loop begin get_token;
    if cur_cmd ≤ max_command then return;
    if (cur_cmd ≥ call) ∧ (cur_cmd < end_template) then
      if info(link(cur_chr)) = protected_token then return;
    expand;
    end;
exit: end;
```

1508*. A group entered (or a conditional started) in one file may end in a different file. Such slight anomalies, although perfectly legitimate, may cause errors that are difficult to locate. In order to be able to give a warning message when such anomalies occur, ε-TEX uses the *grp_stack* and *if_stack* arrays to record the initial *cur_boundary* and *cond_ptr* values for each input file.

```
⟨ Global variables 13 ⟩ +≡
grp_stack: array [0 .. max_in_open] of save_pointer; { initial cur_boundary }
if_stack: array [0 .. max_in_open] of pointer; { initial cond_ptr }
```

1509*. When a group ends that was apparently entered in a different input file, the *group_warning* procedure is invoked in order to update the *grp_stack*. If moreover `\tracingnesting` is positive we want to give a warning message. The situation is, however, somewhat complicated by two facts: (1) There may be *grp_stack* elements without a corresponding `\input` file or `\scantokens` pseudo file (e.g., error insertions from the terminal); and (2) the relevant information is recorded in the *name_field* of the *input_stack* only loosely synchronized with the *in_open* variable indexing *grp_stack*.

```
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
procedure group_warning;
  var i: 0 .. max_in_open; { index into grp_stack }
    w: boolean; { do we need a warning? }
  begin base_ptr ← input_ptr; input_stack[base_ptr] ← cur_input; { store current state }
  i ← in_open; w ← false;
  while (grp_stack[i] = cur_boundary) ∧ (i > 0) do
    begin ⟨ Set variable w to indicate if this case should be reported 1510* ⟩;
    grp_stack[i] ← save_index(save_ptr); decr(i);
    end;
  if w then
    begin print_nl("Warning:␣end␣of␣"); print_group(true); print("␣of␣a␣different␣file"); print_ln;
    if tracing_nesting > 1 then show_context;
    if history = spotless then history ← warning_issued;
    end;
  end;
```

1510*. This code scans the input stack in order to determine the type of the current input file.

```
⟨ Set variable w to indicate if this case should be reported 1510* ⟩ ≡
  if tracing_nesting > 0 then
    begin while (input_stack[base_ptr].state_field = token_list) ∨ (input_stack[base_ptr].index_field > i) do
        decr(base_ptr);
    if input_stack[base_ptr].name_field > 17 then w ← true;
    end
```

This code is used in sections 1509* and 1511*.

1511*. When a conditional ends that was apparently started in a different input file, the *if_warning* procedure is invoked in order to update the *if_stack*. If moreover `\tracingnesting` is positive we want to give a warning message (with the same complications as above).

```
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
procedure if_warning;
  var i: 0 .. max_in_open;  { index into if_stack }
    w: boolean;  { do we need a warning? }
  begin base_ptr ← input_ptr; input_stack[base_ptr] ← cur_input;  { store current state }
  i ← in_open; w ← false;
  while if_stack[i] = cond_ptr do
    begin ⟨ Set variable w to indicate if this case should be reported 1510* ⟩;
    if_stack[i] ← link(cond_ptr); decr(i);
    end;
  if w then
    begin print_nl("Warning:␣end␣of␣"); print_cmd_chr(if_test, cur_if); print_if_line(if_line);
    print("␣of␣a␣different␣file"); print_ln;
    if tracing_nesting > 1 then show_context;
    if history = spotless then history ← warning_issued;
    end;
  end;
```

1512*. Conversely, the *file_warning* procedure is invoked when a file ends and some groups entered or conditionals started while reading from that file are still incomplete.

```
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
procedure file_warning;
  var p: pointer;  { saved value of save_ptr or cond_ptr }
    l: quarterword;  { saved value of cur_level or if_limit }
    c: quarterword;  { saved value of cur_group or cur_if }
    i: integer;  { saved value of if_line }
  begin p ← save_ptr; l ← cur_level; c ← cur_group; save_ptr ← cur_boundary;
  while grp_stack[in_open] ≠ save_ptr do
    begin decr(cur_level); print_nl("Warning:␣end␣of␣file␣when␣"); print_group(true);
    print("␣is␣incomplete");
    cur_group ← save_level(save_ptr); save_ptr ← save_index(save_ptr)
    end;
  save_ptr ← p; cur_level ← l; cur_group ← c;  { restore old values }
  p ← cond_ptr; l ← if_limit; c ← cur_if; i ← if_line;
  while if_stack[in_open] ≠ cond_ptr do
    begin print_nl("Warning:␣end␣of␣file␣when␣"); print_cmd_chr(if_test, cur_if);
    if if_limit = fi_code then print_esc("else");
    print_if_line(if_line); print("␣is␣incomplete");
    if_line ← if_line_field(cond_ptr); cur_if ← subtype(cond_ptr); if_limit ← type(cond_ptr);
    cond_ptr ← link(cond_ptr);
    end;
  cond_ptr ← p; if_limit ← l; cur_if ← c; if_line ← i;  { restore old values }
  print_ln;
  if tracing_nesting > 1 then show_context;
  if history = spotless then history ← warning_issued;
  end;
```

1513* Here are the additional ε-TEX primitives for expressions.

```
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("numexpr", last_item, eTeX_expr − int_val + int_val);
  primitive("dimexpr", last_item, eTeX_expr − int_val + dimen_val);
  primitive("glueexpr", last_item, eTeX_expr − int_val + glue_val);
  primitive("muexpr", last_item, eTeX_expr − int_val + mu_val);
```

1514* ⟨ Cases of *last_item* for *print_cmd_chr* 1381* ⟩ +≡

```
eTeX_expr − int_val + int_val: print_esc("numexpr");
eTeX_expr − int_val + dimen_val: print_esc("dimexpr");
eTeX_expr − int_val + glue_val: print_esc("glueexpr");
eTeX_expr − int_val + mu_val: print_esc("muexpr");
```

1515* This code for reducing *cur_val_level* and/or negating the result is similar to the one for all the other cases of *scan_something_internal*, with the difference that *scan_expr* has already increased the reference count of a glue specification.

```
⟨ Process an expression and return 1515* ⟩ ≡
  begin if $m < eTeX\_mu$ then
    begin case m of
      ⟨ Cases for fetching a glue value 1542* ⟩
    end; { there are no other cases }
    cur_val_level ← glue_val;
    end
  else if $m < eTeX\_expr$ then
      begin case m of
        ⟨ Cases for fetching a mu value 1543* ⟩
      end; { there are no other cases }
      cur_val_level ← mu_val;
      end
    else begin cur_val_level ← m − eTeX_expr + int_val; scan_expr;
      end;
  while $cur\_val\_level > level$ do
    begin if cur_val_level = glue_val then
      begin m ← cur_val; cur_val ← width(m); delete_glue_ref(m);
      end
    else if cur_val_level = mu_val then mu_error;
    decr(cur_val_level);
    end;
  if negative then
    if $cur\_val\_level \ge glue\_val$ then
      begin m ← cur_val; cur_val ← new_spec(m); delete_glue_ref(m);
      ⟨ Negate all three glue components of cur_val 431 ⟩;
      end
    else negate(cur_val);
  return;
  end
```

This code is used in section 424*.

1516* ⟨ Declare ε-TEX procedures for scanning 1413* ⟩ +≡

```
procedure scan_expr; forward;
```

1517* The *scan_expr* procedure scans and evaluates an expression.

```
⟨ Declare procedures needed for expressions 1517* ⟩ ≡
⟨ Declare subprocedures for scan_expr 1528* ⟩
procedure scan_expr;  { scans and evaluates an expression }
  label restart, continue, found;
  var a, b: boolean;  { saved values of arith_error }
    l: small_number;  { type of expression }
    r: small_number;  { state of expression so far }
    s: small_number;  { state of term so far }
    o: small_number;  { next operation or type of next factor }
    e: integer;  { expression so far }
    t: integer;  { term so far }
    f: integer;  { current factor }
    n: integer;  { numerator of combined multiplication and division }
    p: pointer;  { top of expression stack }
    q: pointer;  { for stack manipulations }
  begin l ← cur_val_level; a ← arith_error; b ← false; p ← null;
  ⟨ Scan and evaluate an expression e of type l 1518* ⟩;
  if b then
    begin print_err("Arithmetic␣overflow"); help2("I␣can´t␣evaluate␣this␣expression,")
    ("since␣the␣result␣is␣out␣of␣range."); error;
    if $l \ge$ glue_val then
      begin delete_glue_ref(e); e ← zero_glue; add_glue_ref(e);
      end
    else e ← 0;
    end;
  arith_error ← a; cur_val ← e; cur_val_level ← l;
  end;
```

See also section 1522*.

This code is used in section 461*.

1518* Evaluating an expression is a recursive process: When the left parenthesis of a subexpression is scanned we descend to the next level of recursion; the previous level is resumed with the matching right parenthesis.

```
  define expr_none = 0   { ( seen, or ( ⟨expr⟩ ) seen }
  define expr_add = 1   { ( ⟨expr⟩ + seen }
  define expr_sub = 2   { ( ⟨expr⟩ - seen }
  define expr_mult = 3   { ⟨term⟩ * seen }
  define expr_div = 4   { ⟨term⟩ / seen }
  define expr_scale = 5   { ⟨term⟩ * ⟨factor⟩ / seen }
⟨ Scan and evaluate an expression e of type l 1518* ⟩ ≡
restart: r ← expr_none; e ← 0; s ← expr_none; t ← 0; n ← 0;
continue: if s = expr_none then o ← l else o ← int_val;
  ⟨ Scan a factor f of type o or start a subexpression 1520* ⟩;
found: ⟨ Scan the next operator and set o 1519* ⟩;
  arith_error ← b; ⟨ Make sure that f is in the proper range 1525* ⟩;
  case s of
    ⟨ Cases for evaluation of the current term 1526* ⟩
  end;   { there are no other cases }
  if o > expr_sub then s ← o else ⟨ Evaluate the current expression 1527* ⟩;
  b ← arith_error;
  if o ≠ expr_none then goto continue;
  if p ≠ null then ⟨ Pop the expression stack and goto found 1524* ⟩
```

This code is used in section 1517*.

1519* ⟨ Scan the next operator and set o 1519* ⟩ ≡

```
  ⟨ Get the next non-blank non-call token 406 ⟩;
  if cur_tok = other_token + "+" then o ← expr_add
  else if cur_tok = other_token + "-" then o ← expr_sub
    else if cur_tok = other_token + "*" then o ← expr_mult
      else if cur_tok = other_token + "/" then o ← expr_div
        else begin o ← expr_none;
          if p = null then
            begin if cur_cmd ≠ relax then back_input;
            end
          else if cur_tok ≠ other_token + ")" then
              begin print_err("Missing␣)␣inserted␣for␣expression");
              help1("I␣was␣expecting␣to␣see␣`+',␣`-',␣`*',␣`/',␣or␣`)'.␣Didn't."); back_error;
              end;
          end
```

This code is used in section 1518*.

1520* ⟨ Scan a factor f of type o or start a subexpression 1520* ⟩ ≡

```
  ⟨ Get the next non-blank non-call token 406 ⟩;
  if cur_tok = other_token + "(" then ⟨ Push the expression stack and goto restart 1523* ⟩;
  back_input;
  if o = int_val then scan_int
  else if o = dimen_val then scan_normal_dimen
    else if o = glue_val then scan_normal_glue
      else scan_mu_glue;
  f ← cur_val
```

This code is used in section 1518*.

1521* ⟨ Declare ε-TEX procedures for scanning 1413* ⟩ +≡

```
procedure scan_normal_glue; forward;
procedure scan_mu_glue; forward;
```

1522* Here we declare two trivial procedures in order to avoid mutually recursive procedures with parameters.

⟨ Declare procedures needed for expressions 1517* ⟩ +≡

```
procedure scan_normal_glue;
  begin scan_glue(glue_val);
  end;

procedure scan_mu_glue;
  begin scan_glue(mu_val);
  end;
```

1523* Parenthesized subexpressions can be inside expressions, and this nesting has a stack. Seven local variables represent the top of the expression stack: p points to pushed-down entries, if any; l specifies the type of expression currently beeing evaluated; e is the expression so far and r is the state of its evaluation; t is the term so far and s is the state of its evaluation; finally n is the numerator for a combined multiplication and division, if any.

```
  define expr_node_size = 4  { number of words in stack entry for subexpressions }
  define expr_e_field(#) ≡ mem[# + 1].int  { saved expression so far }
  define expr_t_field(#) ≡ mem[# + 2].int  { saved term so far }
  define expr_n_field(#) ≡ mem[# + 3].int  { saved numerator }
```

⟨ Push the expression stack and **goto** *restart* 1523* ⟩ ≡

```
  begin q ← get_node(expr_node_size); link(q) ← p; type(q) ← l; subtype(q) ← 4 * s + r;
  expr_e_field(q) ← e; expr_t_field(q) ← t; expr_n_field(q) ← n; p ← q; l ← o; goto restart;
  end
```

This code is used in section 1520*.

1524* ⟨ Pop the expression stack and **goto** *found* 1524* ⟩ ≡

```
  begin f ← e; q ← p; e ← expr_e_field(q); t ← expr_t_field(q); n ← expr_n_field(q); s ← subtype(q) div 4;
  r ← subtype(q) mod 4; l ← type(q); p ← link(q); free_node(q, expr_node_size); goto found;
  end
```

This code is used in section 1518*.

1525* We want to make sure that each term and (intermediate) result is in the proper range. Integer values must not exceed *infinity* ($2^{31}-1$) in absolute value, dimensions must not exceed *max_dimen* ($2^{30}-1$). We avoid the absolute value of an integer, because this might fail for the value -2^{31} using 32-bit arithmetic.

```
define num_error(#) ≡ { clear a number or dimension and set arith_error }
        begin arith_error ← true; # ← 0;
        end
define glue_error(#) ≡ { clear a glue spec and set arith_error }
        begin arith_error ← true; delete_glue_ref(#); # ← new_spec(zero_glue);
        end
```

```
⟨ Make sure that f is in the proper range 1525* ⟩ ≡
  if (l = int_val) ∨ (s > expr_sub) then
    begin if (f > infinity) ∨ (f < −infinity) then num_error(f);
    end
  else if l = dimen_val then
      begin if abs(f) > max_dimen then num_error(f);
      end
    else begin if (abs(width(f)) > max_dimen) ∨ (abs(stretch(f)) > max_dimen) ∨
            (abs(shrink(f)) > max_dimen) then glue_error(f);
      end
```

This code is used in section 1518*.

1526* Applying the factor f to the partial term t (with the operator s) is delayed until the next operator o has been scanned. Here we handle the first factor of a partial term. A glue spec has to be copied unless the next operator is a right parenthesis; this allows us later on to simply modify the glue components.

```
define normalize_glue(#) ≡
          if stretch(#) = 0 then stretch_order(#) ← normal;
        if shrink(#) = 0 then shrink_order(#) ← normal
```

```
⟨ Cases for evaluation of the current term 1526* ⟩ ≡
expr_none: if (l ≥ glue_val) ∧ (o ≠ expr_none) then
    begin t ← new_spec(f); delete_glue_ref(f); normalize_glue(t);
    end
  else t ← f;
```

See also sections 1530*, 1531*, and 1533*.

This code is used in section 1518*.

1527* When a term t has been completed it is copied to, added to, or subtracted from the expression e.

```
define expr_add_sub(#) ≡ add_or_sub(#, r = expr_sub)
define expr_a(#) ≡ expr_add_sub(#, max_dimen)
```

```
⟨ Evaluate the current expression 1527* ⟩ ≡
  begin s ← expr_none;
  if r = expr_none then e ← t
  else if l = int_val then e ← expr_add_sub(e, t, infinity)
    else if l = dimen_val then e ← expr_a(e, t)
      else ⟨ Compute the sum or difference of two glue specs 1529* ⟩;
  r ← o;
  end
```

This code is used in section 1518*.

1528* The function *add_or_sub*(x, y, *max_answer*, *negative*) computes the sum (for *negative* = *false*) or difference (for *negative* = *true*) of x and y, provided the absolute value of the result does not exceed *max_answer*.

```
⟨ Declare subprocedures for scan_expr 1528* ⟩ ≡
function add_or_sub(x, y, max_answer : integer; negative : boolean): integer;
  var a: integer;  { the answer }
  begin if negative then negate(y);
  if $x \ge 0$ then
    if $y \le max\_answer - x$ then $a \leftarrow x + y$ else num_error(a)
  else if $y \ge -max\_answer - x$ then $a \leftarrow x + y$ else num_error(a);
  add_or_sub ← a;
  end;
```

See also sections 1532* and 1534*.

This code is used in section 1517*.

1529* We know that *stretch_order*(e) > *normal* implies *stretch*(e) ≠ 0 and *shrink_order*(e) > *normal* implies *shrink*(e) ≠ 0.

```
⟨ Compute the sum or difference of two glue specs 1529* ⟩ ≡
  begin width(e) ← expr_a(width(e), width(t));
  if stretch_order(e) = stretch_order(t) then stretch(e) ← expr_a(stretch(e), stretch(t))
  else if (stretch_order(e) < stretch_order(t)) ∧ (stretch(t) ≠ 0) then
      begin stretch(e) ← stretch(t); stretch_order(e) ← stretch_order(t);
      end;
  if shrink_order(e) = shrink_order(t) then shrink(e) ← expr_a(shrink(e), shrink(t))
  else if (shrink_order(e) < shrink_order(t)) ∧ (shrink(t) ≠ 0) then
      begin shrink(e) ← shrink(t); shrink_order(e) ← shrink_order(t);
      end;
  delete_glue_ref(t); normalize_glue(e);
  end
```

This code is used in section 1527*.

1530* If a multiplication is followed by a division, the two operations are combined into a 'scaling' operation. Otherwise the term t is multiplied by the factor f.

```
  define expr_m(#) ≡ # ← nx_plus_y(#, f, 0)
⟨ Cases for evaluation of the current term 1526* ⟩ +≡
expr_mult: if o = expr_div then
    begin n ← f; o ← expr_scale;
    end
  else if l = int_val then t ← mult_integers(t, f)
    else if l = dimen_val then expr_m(t)
      else begin expr_m(width(t)); expr_m(stretch(t)); expr_m(shrink(t));
        end;
```

1531* Here we divide the term t by the factor f.

```
  define expr_d(#) ≡ # ← quotient(#, f)
⟨ Cases for evaluation of the current term 1526* ⟩ +≡
expr_div: if l < glue_val then expr_d(t)
  else begin expr_d(width(t)); expr_d(stretch(t)); expr_d(shrink(t));
    end;
```

1532* The function *quotient*(n,d) computes the rounded quotient $q=\lfloor n/d+\frac{1}{2}\rfloor$, when n and d are positive.

```
⟨ Declare subprocedures for scan_expr 1528* ⟩ +≡
function quotient(n, d : integer): integer;
  var negative: boolean;  { should the answer be negated? }
    a: integer;  { the answer }
  begin if $d = 0$ then num_error(a)
  else begin if $d > 0$ then negative ← false
    else begin negate(d); negative ← true;
      end;
    if $n < 0$ then
      begin negate(n); negative ← ¬negative;
      end;
    $a ← n$ div $d$; $n ← n − a ∗ d$; $d ← n − d$;  { avoid certain compiler optimizations! }
    if $d + n ≥ 0$ then incr(a);
    if negative then negate(a);
    end;
  quotient ← a;
  end;
```

1533* Here the term t is multiplied by the quotient n/f.

```
  define expr_s(#) ≡ # ← fract(#, n, f, max_dimen)
⟨ Cases for evaluation of the current term 1526* ⟩ +≡
expr_scale: if $l =$ int_val then $t ←$ fract(t, n, f, infinity)
  else if $l =$ dimen_val then expr_s(t)
    else begin expr_s(width(t)); expr_s(stretch(t)); expr_s(shrink(t));
      end;
```

1534* Finally, the function *fract*(x, n, d, max_answer) computes the integer $q = \lfloor xn/d + \frac{1}{2} \rfloor$, when x, n, and d are positive and the result does not exceed *max_answer*. We can't use floating point arithmetic since the routine must produce identical results in all cases; and it would be too dangerous to multiply by n and then divide by d, in separate operations, since overflow might well occur. Hence this subroutine simulates double precision arithmetic, somewhat analogous to METAFONT's *make_fraction* and *take_fraction* routines.

```
  define too_big = 88  { go here when the result is too big }
⟨ Declare subprocedures for scan_expr 1528* ⟩ +≡
function fract(x, n, d, max_answer : integer): integer;
  label found, found1, too_big, done;
  var negative: boolean;  { should the answer be negated? }
    a: integer;  { the answer }
    f: integer;  { a proper fraction }
    h: integer;  { smallest integer such that $2 * h \ge d$ }
    r: integer;  { intermediate remainder }
    t: integer;  { temp variable }
  begin if $d = 0$ then goto too_big;
  $a \leftarrow 0$;
  if $d > 0$ then negative ← false
  else begin negate(d); negative ← true;
    end;
  if $x < 0$ then
    begin negate(x); negative ← ¬negative;
    end
  else if $x = 0$ then goto done;
  if $n < 0$ then
    begin negate(n); negative ← ¬negative;
    end;
  $t \leftarrow n$ div $d$;
  if $t >$ max_answer div $x$ then goto too_big;
  $a \leftarrow t * x$; $n \leftarrow n - t * d$;
  if $n = 0$ then goto found;
  $t \leftarrow x$ div $d$;
  if $t > (max\_answer - a)$ div $n$ then goto too_big;
  $a \leftarrow a + t * n$; $x \leftarrow x - t * d$;
  if $x = 0$ then goto found;
  if $x < n$ then
    begin $t \leftarrow x$; $x \leftarrow n$; $n \leftarrow t$;
    end;  { now $0 < n \le x < d$ }
  ⟨ Compute $f = \lfloor xn/d + \frac{1}{2} \rfloor$ 1535* ⟩
  if $f > (max\_answer - a)$ then goto too_big;
  $a \leftarrow a + f$;
found: if negative then negate(a);
  goto done;
too_big: num_error(a);
done: fract ← a;
  end;
```

1535* The loop here preserves the following invariant relations between f, x, n, and r: (i) $f+\lfloor (xn+(r+d))/d\rfloor = \lfloor x_0n_0/d+\frac{1}{2}\rfloor$; (ii) $-d\le r<0<n\le x<d$, where x_0, n_0 are the original values of x and n.

Notice that the computation specifies $(x-d)+x$ instead of $(x+x)-d$, because the latter could overflow.

```
⟨ Compute $f = \lfloor xn/d + \frac{1}{2}\rfloor$ 1535* ⟩ ≡
  $f \gets 0$; $r \gets (d$ **div** $2) - d$; $h \gets -r$;
  **loop begin if** *odd*$(n)$ **then**
      **begin** $r \gets r + x$;
      **if** $r \ge 0$ **then**
        **begin** $r \gets r - d$; *incr*$(f)$;
        **end**;
      **end**;
    $n \gets n$ **div** 2;
    **if** $n = 0$ **then goto** *found1*;
    **if** $x < h$ **then** $x \gets x + x$
    **else begin** $t \gets x - d$; $x \gets t + x$; $f \gets f + n$;
      **if** $x < n$ **then**
        **begin if** $x = 0$ **then goto** *found1*;
        $t \gets x$; $x \gets n$; $n \gets t$;
        **end**;
      **end**;
    **end**;
*found1*:
```

This code is used in section 1534*.

1536* The `\gluestretch`, `\glueshrink`, `\gluestretchorder`, and `\glueshrinkorder` commands return the stretch and shrink components and their orders of "infinity" of a glue specification.

```
  **define** *glue_stretch_order_code* = *eTeX_int* + 6  { code for \gluestretchorder }
  **define** *glue_shrink_order_code* = *eTeX_int* + 7  { code for \glueshrinkorder }
  **define** *glue_stretch_code* = *eTeX_dim* + 7  { code for \gluestretch }
  **define** *glue_shrink_code* = *eTeX_dim* + 8  { code for \glueshrink }
```

```
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("gluestretchorder", last_item, glue_stretch_order_code);
  primitive("glueshrinkorder", last_item, glue_shrink_order_code);
  primitive("gluestretch", last_item, glue_stretch_code);
  primitive("glueshrink", last_item, glue_shrink_code);
```

1537* ⟨ Cases of *last_item* for *print_cmd_chr* 1381* ⟩ +≡

```
glue_stretch_order_code: print_esc("gluestretchorder");
glue_shrink_order_code: print_esc("glueshrinkorder");
glue_stretch_code: print_esc("gluestretch");
glue_shrink_code: print_esc("glueshrink");
```

1538* ⟨ Cases for fetching an integer value 1382* ⟩ +≡

```
glue_stretch_order_code, glue_shrink_order_code: **begin** scan_normal_glue; $q \gets$ cur_val;
  **if** $m$ = glue_stretch_order_code **then** cur_val $\gets$ stretch_order$(q)$
  **else** cur_val $\gets$ shrink_order$(q)$;
  delete_glue_ref$(q)$;
  **end**;
```

1539* ⟨Cases for fetching a dimension value 1402*⟩ +≡

```
glue_stretch_code, glue_shrink_code: begin scan_normal_glue; q ← cur_val;
  if m = glue_stretch_code then cur_val ← stretch(q)
  else cur_val ← shrink(q);
  delete_glue_ref(q);
  end;
```

1540* The `\mutoglue` and `\gluetomu` commands convert "math" glue into normal glue and vice versa; they allow to manipulate math glue with `\gluestretch` etc.

```
  define mu_to_glue_code = eTeX_glue   { code for \mutoglue }
  define glue_to_mu_code = eTeX_mu   { code for \gluetomu }
```

⟨Generate all ε-TEX primitives 1380*⟩ +≡

```
  primitive("mutoglue", last_item, mu_to_glue_code); primitive("gluetomu", last_item, glue_to_mu_code);
```

1541* ⟨Cases of *last_item* for *print_cmd_chr* 1381*⟩ +≡

```
mu_to_glue_code: print_esc("mutoglue");
glue_to_mu_code: print_esc("gluetomu");
```

1542* ⟨Cases for fetching a glue value 1542*⟩ ≡

```
mu_to_glue_code: scan_mu_glue;
```

This code is used in section 1515*.

1543* ⟨Cases for fetching a mu value 1543*⟩ ≡

```
glue_to_mu_code: scan_normal_glue;
```

This code is used in section 1515*.

1544* ε-TEX (in extended mode) supports 32768 (i.e., 2^{15}) count, dimen, skip, muskip, box, and token registers. As in TEX the first 256 registers of each kind are realized as arrays in the table of equivalents; the additional registers are realized as tree structures built from variable-size nodes with individual registers existing only when needed. Default values are used for nonexistent registers: zero for count and dimen values, *zero_glue* for glue (skip and muskip) values, void for boxes, and *null* for token lists (and current marks discussed below).

Similarly there are 32768 mark classes; the command `\marks`n creates a mark node for a given mark class $0 \le n \le 32767$ (where `\marks0` is synonymous to `\mark`). The page builder (actually the *fire_up* routine) and the *vsplit* routine maintain the current values of *top_mark*, *first_mark*, *bot_mark*, *split_first_mark*, and *split_bot_mark* for each mark class. They are accessed as `\topmarks`n etc., and `\topmarks0` is again synonymous to `\topmark`. As in TEX the five current marks for mark class zero are realized as *cur_mark* array. The additional current marks are again realized as tree structure with individual mark classes existing only when needed.

⟨Generate all ε-TEX primitives 1380*⟩ +≡

```
  primitive("marks", mark, marks_code);
  primitive("topmarks", top_bot_mark, top_mark_code + marks_code);
  primitive("firstmarks", top_bot_mark, first_mark_code + marks_code);
  primitive("botmarks", top_bot_mark, bot_mark_code + marks_code);
  primitive("splitfirstmarks", top_bot_mark, split_first_mark_code + marks_code);
  primitive("splitbotmarks", top_bot_mark, split_bot_mark_code + marks_code);
```

1545* The *scan_register_num* procedure scans a register number that must not exceed 255 in compatibility mode resp. 32767 in extended mode.

⟨Declare ε-TEX procedures for expanding 1487*⟩ +≡

```
procedure scan_register_num; forward;
```

1546* ⟨ Declare procedures that scan restricted classes of integers 433 ⟩ +≡

```
procedure scan_register_num;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > max_reg_num) then
    begin print_err("Bad␣register␣code");
    help2(max_reg_help_line)("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

1547* ⟨ Initialize variables for ε-TEX compatibility mode 1547* ⟩ ≡

```
  max_reg_num ← 255; max_reg_help_line ← "A␣register␣number␣must␣be␣between␣0␣and␣255.";
```

This code is used in sections 1384* and 1386*.

1548* ⟨ Initialize variables for ε-TEX extended mode 1548* ⟩ ≡

```
  max_reg_num ← 32767; max_reg_help_line ← "A␣register␣number␣must␣be␣between␣0␣and␣32767.";
```

This code is used in sections 1379* and 1386*.

1549* ⟨ Global variables 13 ⟩ +≡

```
max_reg_num: halfword;  { largest allowed register number }
max_reg_help_line: str_number;  { first line of help message }
```

1550* There are seven almost identical doubly linked trees, one for the sparse array of the up to 32512 additional registers of each kind and one for the sparse array of the up to 32767 additional mark classes. The root of each such tree, if it exists, is an index node containing 16 pointers to subtrees for 4096 consecutive array elements. Similar index nodes are the starting points for all nonempty subtrees for 4096, 256, and 16 consecutive array elements. These four levels of index nodes are followed by a fifth level with nodes for the individual array elements.

Each index node is nine words long. The pointers to the 16 possible subtrees or are kept in the *info* and *link* fields of the last eight words. (It would be both elegant and efficient to declare them as array, unfortunately Pascal doesn't allow this.)

The fields in the first word of each index node and in the nodes for the array elements are closely related. The *link* field points to the next lower index node and the *sa_index* field contains four bits (one hexadecimal digit) of the register number or mark class. For the lowest index node the *link* field is *null* and the *sa_index* field indicates the type of quantity (*int_val*, *dimen_val*, *glue_val*, *mu_val*, *box_val*, *tok_val*, or *mark_val*). The *sa_used* field in the index nodes counts how many of the 16 pointers are non-null.

The *sa_index* field in the nodes for array elements contains the four bits plus 16 times the type. Therefore such a node represents a count or dimen register if and only if $\mathit{sa\_index} < \mathit{dimen\_val\_limit}$; it represents a skip or muskip register if and only if $\mathit{dimen\_val\_limit} \le \mathit{sa\_index} < \mathit{mu\_val\_limit}$; it represents a box register if and only if $\mathit{mu\_val\_limit} \le \mathit{sa\_index} < \mathit{box\_val\_limit}$; it represents a token list register if and only if $\mathit{box\_val\_limit} \le \mathit{sa\_index} < \mathit{tok\_val\_limit}$; finally it represents a mark class if and only if $\mathit{tok\_val\_limit} \le \mathit{sa\_index}$.

The *new_index* procedure creates an index node (returned in *cur_ptr*) having given contents of the *sa_index* and *link* fields.

```
define box_val ≡ 4 { the additional box registers }
define mark_val = 6 { the additional mark classes }
define dimen_val_limit = ″20 { $2^4 \cdot (dimen\_val + 1)$ }
define mu_val_limit = ″40 { $2^4 \cdot (mu\_val + 1)$ }
define box_val_limit = ″50 { $2^4 \cdot (box\_val + 1)$ }
define tok_val_limit = ″60 { $2^4 \cdot (tok\_val + 1)$ }
define index_node_size = 9 { size of an index node }
define sa_index ≡ type { a four-bit address or a type or both }
define sa_used ≡ subtype { count of non-null pointers }
```

```
⟨ Declare ε-TEX procedures for expanding 1487* ⟩ +≡
procedure new_index(i : quarterword; q : pointer);
  var k: small_number; { loop index }
  begin cur_ptr ← get_node(index_node_size); sa_index(cur_ptr) ← i; sa_used(cur_ptr) ← 0;
  link(cur_ptr) ← q;
  for k ← 1 to index_node_size − 1 do { clear all 16 pointers }
    mem[cur_ptr + k] ← sa_null;
  end;
```

1551* The roots of the seven trees for the additional registers and mark classes are kept in the *sa_root* array. The first six locations must be dumped and undumped; the last one is also known as *sa_mark*.

```
define sa_mark ≡ sa_root[mark_val] { root for mark classes }
```

```
⟨ Global variables 13 ⟩ +≡
sa_root: array [int_val .. mark_val] of pointer; { roots of sparse arrays }
cur_ptr: pointer; { value returned by new_index and find_sa_element }
sa_null: memory_word; { two null pointers }
```

1552* ⟨ Set initial values of key variables 21 ⟩ +≡

```
sa_mark ← null; sa_null.hh.lh ← null; sa_null.hh.rh ← null;
```

1553* ⟨Initialize table entries (done by `INITEX` only) 164⟩ +≡

```
for i ← int_val to tok_val do sa_root[i] ← null;
```

1554* Given a type t and a sixteen-bit number n, the *find_sa_element* procedure returns (in *cur_ptr*) a pointer to the node for the corresponding array element, or *null* when no such element exists. The third parameter w is set *true* if the element must exist, e.g., because it is about to be modified. The procedure has two main branches: one follows the existing tree structure, the other (only used when w is *true*) creates the missing nodes.

We use macros to extract the four-bit pieces from a sixteen-bit register number or mark class and to fetch or store one of the 16 pointers from an index node.

```
define if_cur_ptr_is_null_then_return_or_goto(#) ≡  { some tree element is missing }
        begin if cur_ptr = null then
          if w then goto # else return;
        end

define hex_dig1(#) ≡ # div 4096  { the fourth lowest hexadecimal digit }
define hex_dig2(#) ≡ (# div 256) mod 16  { the third lowest hexadecimal digit }
define hex_dig3(#) ≡ (# div 16) mod 16  { the second lowest hexadecimal digit }
define hex_dig4(#) ≡ # mod 16  { the lowest hexadecimal digit }

define get_sa_ptr ≡
          if odd(i) then cur_ptr ← link(q + (i div 2) + 1)
          else cur_ptr ← info(q + (i div 2) + 1)
                { set cur_ptr to the pointer indexed by i from index node q }
define put_sa_ptr(#) ≡
          if odd(i) then link(q + (i div 2) + 1) ← #
          else info(q + (i div 2) + 1) ← #  { store the pointer indexed by i in index node q }
define add_sa_ptr ≡
          begin put_sa_ptr(cur_ptr); incr(sa_used(q));
          end  { add cur_ptr as the pointer indexed by i in index node q }
define delete_sa_ptr ≡
          begin put_sa_ptr(null); decr(sa_used(q));
          end  { delete the pointer indexed by i in index node q }
```

⟨ Declare ε-TEX procedures for expanding 1487* ⟩ +≡

```
procedure find_sa_element(t : small_number; n : halfword; w : boolean);
        { sets cur_val to sparse array element location or null }
  label not_found, not_found1, not_found2, not_found3, not_found4, exit;
  var q: pointer;  { for list manipulations }
    i: small_number;  { a four bit index }
  begin cur_ptr ← sa_root[t]; if_cur_ptr_is_null_then_return_or_goto(not_found);
  q ← cur_ptr; i ← hex_dig1(n); get_sa_ptr; if_cur_ptr_is_null_then_return_or_goto(not_found1);
  q ← cur_ptr; i ← hex_dig2(n); get_sa_ptr; if_cur_ptr_is_null_then_return_or_goto(not_found2);
  q ← cur_ptr; i ← hex_dig3(n); get_sa_ptr; if_cur_ptr_is_null_then_return_or_goto(not_found3);
  q ← cur_ptr; i ← hex_dig4(n); get_sa_ptr;
  if (cur_ptr = null) ∧ w then goto not_found4;
  return;
not_found: new_index(t, null);  { create first level index node }
  sa_root[t] ← cur_ptr; q ← cur_ptr; i ← hex_dig1(n);
not_found1: new_index(i, q);  { create second level index node }
  add_sa_ptr; q ← cur_ptr; i ← hex_dig2(n);
not_found2: new_index(i, q);  { create third level index node }
  add_sa_ptr; q ← cur_ptr; i ← hex_dig3(n);
not_found3: new_index(i, q);  { create fourth level index node }
  add_sa_ptr; q ← cur_ptr; i ← hex_dig4(n);
not_found4: ⟨ Create a new array element of type t with index i 1555* ⟩;
  link(cur_ptr) ← q; add_sa_ptr;
```

```
exit: end;
```

1555* The array elements for registers are subject to grouping and have an *sa_lev* field (quite analogous to *eq_level*) instead of *sa_used*. Since saved values as well as shorthand definitions (created by e.g., `\countdef`) refer to the location of the respective array element, we need a reference count that is kept in the *sa_ref* field. An array element can be deleted (together with all references to it) when its *sa_ref* value is *null* and its value is the default value.

Skip, muskip, box, and token registers use two word nodes, their values are stored in the *sa_ptr* field. Count and dimen registers use three word nodes, their values are stored in the *sa_int* resp. *sa_dim* field in the third word; the *sa_ptr* field is used under the name *sa_num* to store the register number. Mark classes use four word nodes. The last three words contain the five types of current marks

```
define sa_lev ≡ sa_used   { grouping level for the current value }
define pointer_node_size = 2   { size of an element with a pointer value }
define sa_type(#) ≡ (sa_index(#) div 16)   { type part of combined type/index }
define sa_ref(#) ≡ info(# + 1)   { reference count of a sparse array element }
define sa_ptr(#) ≡ link(# + 1)   { a pointer value }

define word_node_size = 3   { size of an element with a word value }
define sa_num ≡ sa_ptr   { the register number }
define sa_int(#) ≡ mem[# + 2].int   { an integer }
define sa_dim(#) ≡ mem[# + 2].sc   { a dimension (a somewhat esotheric distinction) }

define mark_class_node_size = 4   { size of an element for a mark class }

define fetch_box(#) ≡   { fetch box(cur_val) }
      if cur_val < 256 then # ← box(cur_val)
      else begin find_sa_element(box_val, cur_val, false);
        if cur_ptr = null then # ← null else # ← sa_ptr(cur_ptr);
        end
```

```
⟨ Create a new array element of type t with index i 1555* ⟩ ≡
  if t = mark_val then   { a mark class }
    begin cur_ptr ← get_node(mark_class_node_size); mem[cur_ptr + 1] ← sa_null;
    mem[cur_ptr + 2] ← sa_null; mem[cur_ptr + 3] ← sa_null;
    end
  else begin if t ≤ dimen_val then   { a count or dimen register }
      begin cur_ptr ← get_node(word_node_size); sa_int(cur_ptr) ← 0; sa_num(cur_ptr) ← n;
      end
    else begin cur_ptr ← get_node(pointer_node_size);
      if t ≤ mu_val then   { a skip or muskip register }
        begin sa_ptr(cur_ptr) ← zero_glue; add_glue_ref(zero_glue);
        end
      else sa_ptr(cur_ptr) ← null;   { a box or token list register }
      end;
    sa_ref(cur_ptr) ← null;   { all registers have a reference count }
    end;
  sa_index(cur_ptr) ← 16 ∗ t + i; sa_lev(cur_ptr) ← level_one
```

This code is used in section 1554*.

1556* The *delete_sa_ref* procedure is called when a pointer to an array element representing a register is being removed; this means that the reference count should be decreased by one. If the reduced reference count is *null* and the register has been (globally) assigned its default value the array element should disappear, possibly together with some index nodes. This procedure will never be used for mark class nodes.

```
  define add_sa_ref(#) ≡ incr(sa_ref(#))   { increase reference count }
  define change_box(#) ≡   { change box(cur_val), the eq_level stays the same }
        if cur_val < 256 then box(cur_val) ← # else set_sa_box(#)
  define set_sa_box(#) ≡
          begin find_sa_element(box_val, cur_val, false);
          if cur_ptr ≠ null then
            begin sa_ptr(cur_ptr) ← #; add_sa_ref(cur_ptr); delete_sa_ref(cur_ptr);
            end;
          end
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
procedure delete_sa_ref(q : pointer);   { reduce reference count }
  label exit;
  var p: pointer;   { for list manipulations }
    i: small_number;   { a four bit index }
    s: small_number;   { size of a node }
  begin decr(sa_ref(q));
  if sa_ref(q) ≠ null then return;
  if sa_index(q) < dimen_val_limit then
    if sa_int(q) = 0 then s ← word_node_size
    else return
  else begin if sa_index(q) < mu_val_limit then
      if sa_ptr(q) = zero_glue then delete_glue_ref(zero_glue)
      else return
    else if sa_ptr(q) ≠ null then return;
    s ← pointer_node_size;
    end;
  repeat i ← hex_dig4(sa_index(q)); p ← q; q ← link(p); free_node(p, s);
    if q = null then   { the whole tree has been freed }
      begin sa_root[i] ← null; return;
      end;
    delete_sa_ptr; s ← index_node_size;   { node q is an index node }
  until sa_used(q) > 0;
exit: end;
```

1557* The *print_sa_num* procedure prints the register number corresponding to an array element.

```
⟨ Basic printing procedures 57 ⟩ +≡
procedure print_sa_num(q : pointer);   { print register number }
  var n: halfword;   { the register number }
  begin if sa_index(q) < dimen_val_limit then n ← sa_num(q)   { the easy case }
  else begin n ← hex_dig4(sa_index(q)); q ← link(q); n ← n + 16 ∗ sa_index(q); q ← link(q);
    n ← n + 256 ∗ (sa_index(q) + 16 ∗ sa_index(link(q)));
    end;
  print_int(n);
  end;
```

1558* Here is a procedure that displays the contents of an array element symbolically. It is used under similar circumstances as is *restore_trace* (together with *show_eqtb*) for the quantities kept in the *eqtb* array.

```
⟨Declare ε-TeX procedures for tracing and input 284*⟩ +≡
  stat procedure show_sa(p : pointer; s : str_number);
  var t: small_number; {the type of element}
  begin begin_diagnostic; print_char("{"); print(s); print_char("␣");
  if p = null then print_char("?")  {this can't happen}
  else begin t ← sa_type(p);
    if t < box_val then print_cmd_chr(register, p)
    else if t = box_val then
          begin print_esc("box"); print_sa_num(p);
          end
      else if t = tok_val then print_cmd_chr(toks_register, p)
        else print_char("?"); {this can't happen either}
    print_char("=");
    if t = int_val then print_int(sa_int(p))
    else if t = dimen_val then
          begin print_scaled(sa_dim(p)); print("pt");
          end
      else begin p ← sa_ptr(p);
        if t = glue_val then print_spec(p, "pt")
        else if t = mu_val then print_spec(p, "mu")
          else if t = box_val then
              if p = null then print("void")
              else begin depth_threshold ← 0; breadth_max ← 1; show_node_list(p);
                end
            else if t = tok_val then
                begin if p ≠ null then show_token_list(link(p), null, 32);
                end
              else print_char("?"); {this can't happen either}
        end;
    end;
  print_char("}"); end_diagnostic(false);
  end;
  tats
```

1559* Here we compute the pointer to the current mark of type *t* and mark class *cur_val*.

```
⟨Compute the mark pointer for mark type t and class cur_val 1559*⟩ ≡
  begin find_sa_element(mark_val, cur_val, false);
  if cur_ptr ≠ null then
    if odd(t) then cur_ptr ← link(cur_ptr + (t div 2) + 1)
    else cur_ptr ← info(cur_ptr + (t div 2) + 1);
  end
```

This code is used in section 386*.

1560* The current marks for all mark classes are maintained by the *vsplit* and *fire_up* routines and are finally destroyed (for INITEX only) by the *final_cleanup* routine. Apart from updating the current marks when mark nodes are encountered, these routines perform certain actions on all existing mark classes. The recursive *do_marks* procedure walks through the whole tree or a subtree of existing mark class nodes and preforms certain actions indicted by its first parameter a, the action code. The second parameter l indicates the level of recursion (at most four); the third parameter points to a nonempty tree or subtree. The result is *true* if the complete tree or subtree has been deleted.

```
  define vsplit_init ≡ 0   { action code for vsplit initialization }
  define fire_up_init ≡ 1   { action code for fire_up initialization }
  define fire_up_done ≡ 2   { action code for fire_up completion }
  define destroy_marks ≡ 3   { action code for final_cleanup }
  define sa_top_mark(#) ≡ info(# + 1)   { \topmarksn }
  define sa_first_mark(#) ≡ link(# + 1)   { \firstmarksn }
  define sa_bot_mark(#) ≡ info(# + 2)   { \botmarksn }
  define sa_split_first_mark(#) ≡ link(# + 2)   { \splitfirstmarksn }
  define sa_split_bot_mark(#) ≡ info(# + 3)   { \splitbotmarksn }
```

⟨ Declare the function called *do_marks* 1560* ⟩ ≡

```
function do_marks(a, l : small_number; q : pointer): boolean;
  var i: small_number;   { a four bit index }
  begin if $l < 4$ then   { q is an index node }
    begin for $i \gets 0$ to 15 do
      begin get_sa_ptr;
      if $cur\_ptr \neq null$ then
        if do_marks(a, l + 1, cur_ptr) then delete_sa_ptr;
      end;
    if $sa\_used(q) = 0$ then
      begin free_node(q, index_node_size); $q \gets null$;
      end;
    end
  else   { q is the node for a mark class }
  begin case a of
    ⟨ Cases for do_marks 1561* ⟩
  end;   { there are no other cases }
  if $sa\_bot\_mark(q) = null$ then
    if $sa\_split\_bot\_mark(q) = null$ then
      begin free_node(q, mark_class_node_size); $q \gets null$;
      end;
  end; $do\_marks \gets (q = null)$;
  end;
```

This code is used in section 977*.

1561* At the start of the *vsplit* routine the existing *split_fist_mark* and *split_bot_mark* are discarded.

⟨ Cases for *do_marks* 1561* ⟩ ≡

```
vsplit_init: if $sa\_split\_first\_mark(q) \neq null$ then
    begin delete_token_ref(sa_split_first_mark(q)); $sa\_split\_first\_mark(q) \gets null$;
    delete_token_ref(sa_split_bot_mark(q)); $sa\_split\_bot\_mark(q) \gets null$;
    end;
```

See also sections 1563*, 1564*, and 1566*.

This code is used in section 1560*.

1562* We use again the fact that *split_first_mark* = *null* if and only if *split_bot_mark* = *null*.

```
⟨ Update the current marks for vsplit 1562* ⟩ ≡
  begin find_sa_element(mark_val, mark_class(p), true);
  if sa_split_first_mark(cur_ptr) = null then
    begin sa_split_first_mark(cur_ptr) ← mark_ptr(p); add_token_ref(mark_ptr(p));
    end
  else delete_token_ref(sa_split_bot_mark(cur_ptr));
  sa_split_bot_mark(cur_ptr) ← mark_ptr(p); add_token_ref(mark_ptr(p));
  end
```

This code is used in section 979*.

1563* At the start of the *fire_up* routine the old *top_mark* and *first_mark* are discarded, whereas the old *bot_mark* becomes the new *top_mark*. An empty new *top_mark* token list is, however, discarded as well in order that mark class nodes can eventually be released. We use again the fact that *bot_mark* ≠ *null* implies *first_mark* ≠ *null*; it also knows that *bot_mark* = *null* implies *top_mark* = *first_mark* = *null*.

```
⟨ Cases for do_marks 1561* ⟩ +≡
fire_up_init: if sa_bot_mark(q) ≠ null then
    begin if sa_top_mark(q) ≠ null then delete_token_ref(sa_top_mark(q));
    delete_token_ref(sa_first_mark(q)); sa_first_mark(q) ← null;
    if link(sa_bot_mark(q)) = null then  { an empty token list }
      begin delete_token_ref(sa_bot_mark(q)); sa_bot_mark(q) ← null;
      end
    else add_token_ref(sa_bot_mark(q));
    sa_top_mark(q) ← sa_bot_mark(q);
    end;
```

1564* ⟨ Cases for *do_marks* 1561* ⟩ +≡

```
fire_up_done: if (sa_top_mark(q) ≠ null) ∧ (sa_first_mark(q) = null) then
    begin sa_first_mark(q) ← sa_top_mark(q); add_token_ref(sa_top_mark(q));
    end;
```

1565* ⟨ Update the current marks for *fire_up* 1565* ⟩ ≡

```
  begin find_sa_element(mark_val, mark_class(p), true);
  if sa_first_mark(cur_ptr) = null then
    begin sa_first_mark(cur_ptr) ← mark_ptr(p); add_token_ref(mark_ptr(p));
    end;
  if sa_bot_mark(cur_ptr) ≠ null then delete_token_ref(sa_bot_mark(cur_ptr));
  sa_bot_mark(cur_ptr) ← mark_ptr(p); add_token_ref(mark_ptr(p));
  end
```

This code is used in section 1014*.

1566*. Here we use the fact that the five current mark pointers in a mark class node occupy the same locations as the the first five pointers of an index node. For systems using a run-time switch to distinguish between `VIRTEX` and `INITEX`, the codewords '**init** . . . **tini**' surrounding the following piece of code should be removed.

```
⟨ Cases for do_marks 1561* ⟩ +≡
  init destroy_marks: for i ← top_mark_code to split_bot_mark_code do
    begin get_sa_ptr;
    if cur_ptr ≠ null then
      begin delete_token_ref(cur_ptr); put_sa_ptr(null);
      end;
    end;
  tini
```

1567*. The command code *register* is used for '`\count`', '`\dimen`', etc., as well as for references to sparse array elements defined by '`\countdef`', etc.

```
⟨ Cases of register for print_cmd_chr 1567* ⟩ ≡
  begin if (chr_code < mem_bot) ∨ (chr_code > lo_mem_stat_max) then cmd ← sa_type(chr_code)
  else begin cmd ← chr_code − mem_bot; chr_code ← null;
    end;
  if cmd = int_val then print_esc("count")
  else if cmd = dimen_val then print_esc("dimen")
    else if cmd = glue_val then print_esc("skip")
      else print_esc("muskip");
  if chr_code ≠ null then print_sa_num(chr_code);
  end
```

This code is used in section 412*.

1568*. Similarly the command code *toks_register* is used for '`\toks`' as well as for references to sparse array elements defined by '`\toksdef`'.

```
⟨ Cases of toks_register for print_cmd_chr 1568* ⟩ ≡
  begin print_esc("toks");
  if chr_code ≠ mem_bot then print_sa_num(chr_code);
  end
```

This code is used in section 266*.

1569*. When a shorthand definition for an element of one of the sparse arrays is destroyed, we must reduce the reference count.

```
⟨ Cases for eq_destroy 1569* ⟩ ≡
toks_register, register: if (equiv_field(w) < mem_bot) ∨ (equiv_field(w) > lo_mem_stat_max) then
    delete_sa_ref(equiv_field(w));
```

This code is used in section 275*.

1570*. The task to maintain (change, save, and restore) register values is essentially the same when the register is realized as sparse array element or entry in *eqtb*. The global variable *sa_chain* is the head of a linked list of entries saved at the topmost level *sa_level*; the lists for lowel levels are kept in special save stack entries.

```
⟨ Global variables 13 ⟩ +≡
sa_chain: pointer;  { chain of saved sparse array entries }
sa_level: quarterword;  { group level for sa_chain }
```

1571* ⟨ Set initial values of key variables 21 ⟩ +≡

```
  sa_chain ← null; sa_level ← level_zero;
```

1572* The individual saved items are kept in pointer or word nodes similar to those used for the array elements: a word node with value zero is, however, saved as pointer node with the otherwise impossible *sa_index* value *tok_val_limit*.

```
  define sa_loc ≡ sa_ref    { location of saved item }
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
procedure sa_save(p : pointer);   { saves value of p }
  var q: pointer;   { the new save node }
    i: quarterword;   { index field of node }
  begin if cur_level ≠ sa_level then
    begin check_full_save_stack; save_type(save_ptr) ← restore_sa; save_level(save_ptr) ← sa_level;
    save_index(save_ptr) ← sa_chain; incr(save_ptr); sa_chain ← null; sa_level ← cur_level;
    end;
  i ← sa_index(p);
  if i < dimen_val_limit then
    begin if sa_int(p) = 0 then
      begin q ← get_node(pointer_node_size); i ← tok_val_limit;
      end
    else begin q ← get_node(word_node_size); sa_int(q) ← sa_int(p);
      end;
    sa_ptr(q) ← null;
    end
  else begin q ← get_node(pointer_node_size); sa_ptr(q) ← sa_ptr(p);
    end;
  sa_loc(q) ← p; sa_index(q) ← i; sa_lev(q) ← sa_lev(p); link(q) ← sa_chain; sa_chain ← q; add_sa_ref(p);
  end;
```

1573* ⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡

```
procedure sa_destroy(p : pointer);   { destroy value of p }
  begin if sa_index(p) < mu_val_limit then delete_glue_ref(sa_ptr(p))
  else if sa_ptr(p) ≠ null then
      if sa_index(p) < box_val_limit then flush_node_list(sa_ptr(p))
      else delete_token_ref(sa_ptr(p));
  end;
```

1574* The procedure *sa_def* assigns a new value to sparse array elements, and saves the former value if appropriate. This procedure is used only for skip, muskip, box, and token list registers. The counterpart of *sa_def* for count and dimen registers is called *sa_w_def*.

```
  define sa_define(#) ≡
          if e then
            if global then gsa_def(#) else sa_def(#)
          else define
  define sa_def_box ≡ { assign cur_box to box(cur_val) }
        begin find_sa_element(box_val, cur_val, true);
        if global then gsa_def(cur_ptr, cur_box) else sa_def(cur_ptr, cur_box);
        end
  define sa_word_define(#) ≡
          if e then
            if global then gsa_w_def(#) else sa_w_def(#)
          else word_define(#)
```

⟨Declare ε-TEX procedures for tracing and input 284*⟩ +≡

```
procedure sa_def(p : pointer; e : halfword);   { new data for sparse array elements }
  begin add_sa_ref(p);
  if sa_ptr(p) = e then
    begin stat if tracing_assigns > 0 then show_sa(p, "reassigning");
    tats
    sa_destroy(p);
    end
  else begin stat if tracing_assigns > 0 then show_sa(p, "changing");
    tats
    if sa_lev(p) = cur_level then sa_destroy(p) else sa_save(p);
    sa_lev(p) ← cur_level; sa_ptr(p) ← e;
    stat if tracing_assigns > 0 then show_sa(p, "into");
    tats
    end;
  delete_sa_ref(p);
  end;

procedure sa_w_def(p : pointer; w : integer);
  begin add_sa_ref(p);
  if sa_int(p) = w then
    begin stat if tracing_assigns > 0 then show_sa(p, "reassigning");
    tats
    end
  else begin stat if tracing_assigns > 0 then show_sa(p, "changing");
    tats
    if sa_lev(p) ≠ cur_level then sa_save(p);
    sa_lev(p) ← cur_level; sa_int(p) ← w;
    stat if tracing_assigns > 0 then show_sa(p, "into");
    tats
    end;
  delete_sa_ref(p);
  end;
```

1575* The *sa_def* and *sa_w_def* routines take care of local definitions. Global definitions are done in almost the same way, but there is no need to save old values, and the new value is associated with *level_one*.

```
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
procedure gsa_def(p : pointer; e : halfword);   { global sa_def }
  begin add_sa_ref(p);
  stat if tracing_assigns > 0 then show_sa(p, "globally␣changing");
  tats
  sa_destroy(p); sa_lev(p) ← level_one; sa_ptr(p) ← e;
  stat if tracing_assigns > 0 then show_sa(p, "into");
  tats
  delete_sa_ref(p);
  end;
procedure gsa_w_def(p : pointer; w : integer);   { global sa_w_def }
  begin add_sa_ref(p);
  stat if tracing_assigns > 0 then show_sa(p, "globally␣changing");
  tats
  sa_lev(p) ← level_one; sa_int(p) ← w;
  stat if tracing_assigns > 0 then show_sa(p, "into");
  tats
  delete_sa_ref(p);
  end;
```

1576* The *sa_restore* procedure restores the sparse array entries pointed at by *sa_chain*

```
⟨ Declare ε-TEX procedures for tracing and input 284* ⟩ +≡
procedure sa_restore;
  var p: pointer;   { sparse array element }
  begin repeat p ← sa_loc(sa_chain);
    if sa_lev(p) = level_one then
      begin if sa_index(p) ≥ dimen_val_limit then sa_destroy(sa_chain);
      stat if tracing_restores > 0 then show_sa(p, "retaining");
      tats
      end
    else begin if sa_index(p) < dimen_val_limit then
        if sa_index(sa_chain) < dimen_val_limit then sa_int(p) ← sa_int(sa_chain)
        else sa_int(p) ← 0
      else begin sa_destroy(p); sa_ptr(p) ← sa_ptr(sa_chain);
        end;
      sa_lev(p) ← sa_lev(sa_chain);
      stat if tracing_restores > 0 then show_sa(p, "restoring");
      tats
      end;
    delete_sa_ref(p); p ← sa_chain; sa_chain ← link(p);
    if sa_index(p) < dimen_val_limit then free_node(p, word_node_size)
    else free_node(p, pointer_node_size);
  until sa_chain = null;
  end;
```

1577* When the value of *last_line_fit* is positive, the last line of a (partial) paragraph is treated in a special way and we need additional fields in the active nodes.

```
define active_node_size_extended = 5   { number of words in extended active nodes }
define active_short(#) ≡ mem[# + 3].sc   { shortfall of this line }
define active_glue(#) ≡ mem[# + 4].sc   { corresponding glue stretch or shrink }
```

```
⟨ Global variables 13 ⟩ +≡
last_line_fill: pointer;   { the par_fill_skip glue node of the new paragraph }
do_last_line_fit: boolean;   { special algorithm for last line of paragraph? }
active_node_size: small_number;   { number of words in active nodes }
fill_width: array [0 .. 2] of scaled;   { infinite stretch components of par_fill_skip }
best_pl_short: array [very_loose_fit .. tight_fit] of scaled;   { shortfall corresponding to minimal_demerits }
best_pl_glue: array [very_loose_fit .. tight_fit] of scaled;   { corresponding glue stretch or shrink }
```

1578* The new algorithm for the last line requires that the stretchability of *par_fill_skip* is infinite and the stretchability of *left_skip* plus *right_skip* is finite.

```
⟨ Check for special treatment of last line of paragraph 1578* ⟩ ≡
  do_last_line_fit ← false; active_node_size ← active_node_size_normal;   { just in case }
  if last_line_fit > 0 then
    begin q ← glue_ptr(last_line_fill);
    if (stretch(q) > 0) ∧ (stretch_order(q) > normal) then
      if (background[3] = 0) ∧ (background[4] = 0) ∧ (background[5] = 0) then
        begin do_last_line_fit ← true; active_node_size ← active_node_size_extended; fill_width[0] ← 0;
        fill_width[1] ← 0; fill_width[2] ← 0; fill_width[stretch_order(q) − 1] ← stretch(q);
        end;
    end
```

This code is used in section 827*.

1579* ⟨ Other local variables for *try_break* 830 ⟩ +≡

```
g: scaled;   { glue stretch or shrink of test line, adjustment for last line }
```

1580* Here we initialize the additional fields of the first active node representing the beginning of the paragraph.

```
⟨ Initialize additional fields of the first active node 1580* ⟩ ≡
  begin active_short(q) ← 0; active_glue(q) ← 0;
  end
```

This code is used in section 864*.

1581* Here we compute the adjustment g and badness b for a line from r to the end of the paragraph. When any of the criteria for adjustment is violated we fall through to the normal algorithm.

The last line must be too short, and have infinite stretch entirely due to *par_fill_skip*.

```
⟨Perform computations for last line and goto found 1581*⟩ ≡
  begin if $(active\_short(r) = 0) \lor (active\_glue(r) \le 0)$ then goto not_found;
        { previous line was neither stretched nor shrunk, or was infinitely bad }
  if $(cur\_active\_width[3] \ne fill\_width[0]) \lor (cur\_active\_width[4] \ne fill\_width[1]) \lor$
        $(cur\_active\_width[5] \ne fill\_width[2])$ then goto not_found;
        { infinite stretch of this line not entirely due to par_fill_skip }
  if $active\_short(r) > 0$ then $g \gets cur\_active\_width[2]$
  else $g \gets cur\_active\_width[6]$;
  if $g \le 0$ then goto not_found;   { no finite stretch resp. no shrink }
  $arith\_error \gets false$; $g \gets fract(g, active\_short(r), active\_glue(r), max\_dimen)$;
  if $last\_line\_fit < 1000$ then $g \gets fract(g, last\_line\_fit, 1000, max\_dimen)$;
  if arith_error then
    if $active\_short(r) > 0$ then $g \gets max\_dimen$ else $g \gets -max\_dimen$;
  if $g > 0$ then ⟨Set the value of b to the badness of the last line for stretching, compute the corresponding
        fit_class, and goto found 1582*⟩
  else if $g < 0$ then ⟨Set the value of b to the badness of the last line for shrinking, compute the
          corresponding fit_class, and goto found 1583*⟩;
not_found: end
```

This code is used in section 852*.

1582* These badness computations are rather similar to those of the standard algorithm, with the adjustment amount g replacing the *shortfall*.

```
⟨Set the value of b to the badness of the last line for stretching, compute the corresponding fit_class, and
      goto found 1582*⟩ ≡
  begin if $g > shortfall$ then $g \gets shortfall$;
  if $g > 7230584$ then
    if $cur\_active\_width[2] < 1663497$ then
      begin $b \gets inf\_bad$; $fit\_class \gets very\_loose\_fit$; goto found;
      end;
  $b \gets badness(g, cur\_active\_width[2])$;
  if $b > 12$ then
    if $b > 99$ then $fit\_class \gets very\_loose\_fit$
    else $fit\_class \gets loose\_fit$
  else $fit\_class \gets decent\_fit$;
  goto found;
  end
```

This code is used in section 1581*.

1583* ⟨Set the value of b to the badness of the last line for shrinking, compute the corresponding *fit_class*, and **goto** *found* 1583*⟩ ≡

```
  begin if $-g > cur\_active\_width[6]$ then $g \gets -cur\_active\_width[6]$;
  $b \gets badness(-g, cur\_active\_width[6])$;
  if $b > 12$ then $fit\_class \gets tight\_fit$ else $fit\_class \gets decent\_fit$;
  goto found;
  end
```

This code is used in section 1581*.

1584*. Vanishing values of *shortfall* and *g* indicate that the last line is not adjusted.

```
⟨ Adjust the additional data for last line 1584* ⟩ ≡
  begin if cur_p = null then shortfall ← 0;
  if shortfall > 0 then g ← cur_active_width[2]
  else if shortfall < 0 then g ← cur_active_width[6]
    else g ← 0;
  end
```

This code is used in section 851*.

1585*. For each feasible break we record the shortfall and glue stretch or shrink (or adjustment).

```
⟨ Store additional data for this feasible break 1585* ⟩ ≡
  begin best_pl_short[fit_class] ← shortfall; best_pl_glue[fit_class] ← g;
  end
```

This code is used in section 855*.

1586*. Here we save these data in the active node representing a potential line break.

```
⟨ Store additional data in the new active node 1586* ⟩ ≡
  begin active_short(q) ← best_pl_short[fit_class]; active_glue(q) ← best_pl_glue[fit_class];
  end
```

This code is used in section 845*.

1587*.

```
⟨ Print additional data in the new active node 1587* ⟩ ≡
  begin print("␣s="); print_scaled(active_short(q));
  if cur_p = null then print("␣a=") else print("␣g=");
  print_scaled(active_glue(q));
  end
```

This code is used in section 846*.

1588*. Here we either reset *do_last_line_fit* or adjust the *par_fill_skip* glue.

```
⟨ Adjust the final line of the paragraph 1588* ⟩ ≡
  if active_short(best_bet) = 0 then do_last_line_fit ← false
  else begin q ← new_spec(glue_ptr(last_line_fill)); delete_glue_ref(glue_ptr(last_line_fill));
    width(q) ← width(q) + active_short(best_bet) − active_glue(best_bet); stretch(q) ← 0;
    glue_ptr(last_line_fill) ← q;
    end
```

This code is used in section 863*.

1589*. When reading `\patterns` while `\savinghyphcodes` is positive the current *lc_code* values are stored together with the hyphenation patterns for the current language. They will later be used instead of the *lc_code* values for hyphenation purposes.

The *lc_code* values are stored in the linked trie analogous to patterns p_1 of length 1, with *hyph_root* = *trie_r*[0] replacing *trie_root* and *lc_code*(*p_1*) replacing the *trie_op* code. This allows to compress and pack them together with the patterns with minimal changes to the existing code.

```
  define hyph_root ≡ trie_r[0]   { root of the linked trie for hyph_codes }
⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
  hyph_root ← 0; hyph_start ← 0;
```

1590*. ⟨ Store hyphenation codes for current language 1590* ⟩ ≡

```
begin $c \gets$ cur_lang; first_child $\gets$ false; $p \gets 0$;
repeat $q \gets p$; $p \gets$ trie_r[q];
until $(p = 0) \lor (c \le so(trie\_c[p]))$;
if $(p = 0) \lor (c < so(trie\_c[p]))$ then ⟨ Insert a new trie node between q and p, and make p point to it 964 ⟩;
$q \gets p$; { now node q represents cur_lang }
⟨ Store all current lc_code values 1591* ⟩;
end
```

This code is used in section 960*.

1591*. We store all nonzero *lc_code* values, overwriting any previously stored values (and possibly wasting a few trie nodes that were used previously and are not needed now). We always store at least one *lc_code* value such that *hyph_index* (defined below) will not be zero.

⟨ Store all current *lc_code* values 1591* ⟩ ≡

```
$p \gets$ trie_l[q]; first_child $\gets$ true;
for $c \gets 0$ to 255 do
  if $(lc\_code(c) > 0) \lor ((c = 255) \land first\_child)$ then
    begin if $p = 0$ then ⟨ Insert a new trie node between q and p, and make p point to it 964 ⟩
    else trie_c[p] $\gets$ si(c);
    trie_o[p] $\gets$ qi(lc_code(c)); $q \gets p$; $p \gets$ trie_r[q]; first_child $\gets$ false;
    end;
if first_child then trie_l[q] $\gets 0$ else trie_r[q] $\gets 0$
```

This code is used in section 1590*.

1592*. We must avoid to "take" location 1, in order to distinguish between *lc_code* values and patterns.

⟨ Pack all stored *hyph_codes* 1592* ⟩ ≡

```
begin if trie_root = 0 then
  for $p \gets 0$ to 255 do trie_min[p] $\gets p + 2$;
first_fit(hyph_root); trie_pack(hyph_root); hyph_start $\gets$ trie_ref[hyph_root];
end
```

This code is used in section 966*.

1593*. The global variable *hyph_index* will point to the hyphenation codes for the current language.

```
define set_hyph_index $\equiv$ { set hyph_index for current language }
        if trie_char(hyph_start + cur_lang) $\ne$ qi(cur_lang) then hyph_index $\gets 0$
              { no hyphenation codes for cur_lang }
        else hyph_index $\gets$ trie_link(hyph_start + cur_lang)
define set_lc_code(#) $\equiv$ { set hc[0] to hyphenation or lc code for # }
        if hyph_index = 0 then hc[0] $\gets$ lc_code(#)
        else if trie_char(hyph_index + #) $\ne$ qi(#) then hc[0] $\gets 0$
          else hc[0] $\gets$ qo(trie_op(hyph_index + #))
```

⟨ Global variables 13 ⟩ +≡

```
hyph_start: trie_pointer; { root of the packed trie for hyph_codes }
hyph_index: trie_pointer; { pointer to hyphenation codes for cur_lang }
```

1594* When *saving_vdiscards* is positive then the glue, kern, and penalty nodes removed by the page builder or by `\vsplit` from the top of a vertical list are saved in special lists instead of being discarded.

```
define tail_page_disc ≡ disc_ptr[copy_code]  { last item removed by page builder }
define page_disc ≡ disc_ptr[last_box_code]  { first item removed by page builder }
define split_disc ≡ disc_ptr[vsplit_code]  { first item removed by \vsplit }
```

```
⟨ Global variables 13 ⟩ +≡
disc_ptr: array [copy_code .. vsplit_code] of pointer;  { list pointers }
```

1595* ⟨ Set initial values of key variables 21 ⟩ +≡

```
  page_disc ← null; split_disc ← null;
```

1596* The `\pagediscards` and `\splitdiscards` commands share the command code *un_vbox* with `\unvbox` and `\unvcopy`, they are distinguished by their *chr_code* values *last_box_code* and *vsplit_code*. These *chr_code* values are larger than *box_code* and *copy_code*.

```
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("pagediscards", un_vbox, last_box_code);
  primitive("splitdiscards", un_vbox, vsplit_code);
```

1597* ⟨ Cases of *un_vbox* for *print_cmd_chr* 1597* ⟩ ≡

```
else if chr_code = last_box_code then print_esc("pagediscards")
  else if chr_code = vsplit_code then print_esc("splitdiscards")
```

This code is used in section 1108*.

1598* ⟨ Handle saved items and **goto** *done* 1598* ⟩ ≡

```
  begin link(tail) ← disc_ptr[cur_chr]; disc_ptr[cur_chr] ← null; goto done;
  end
```

This code is used in section 1110*.

1599* The `\interlinepenalties`, `\clubpenalties`, `\widowpenalties`, and `\displaywidowpenalties` commands allow to define arrays of penalty values to be used instead of the corresponding single values.

```
define inter_line_penalties_ptr ≡ equiv(inter_line_penalties_loc)
define club_penalties_ptr ≡ equiv(club_penalties_loc)
define widow_penalties_ptr ≡ equiv(widow_penalties_loc)
define display_widow_penalties_ptr ≡ equiv(display_widow_penalties_loc)
```

```
⟨ Generate all ε-TEX primitives 1380* ⟩ +≡
  primitive("interlinepenalties", set_shape, inter_line_penalties_loc);
  primitive("clubpenalties", set_shape, club_penalties_loc);
  primitive("widowpenalties", set_shape, widow_penalties_loc);
  primitive("displaywidowpenalties", set_shape, display_widow_penalties_loc);
```

1600* ⟨ Cases of *set_shape* for *print_cmd_chr* 1600* ⟩ ≡

```
inter_line_penalties_loc: print_esc("interlinepenalties");
club_penalties_loc: print_esc("clubpenalties");
widow_penalties_loc: print_esc("widowpenalties");
display_widow_penalties_loc: print_esc("displaywidowpenalties");
```

This code is used in section 266*.

1601* ⟨Fetch a penalties array element 1601*⟩ ≡

```
begin scan_int;
if $(equiv(m) = null) \lor (cur\_val < 0)$ then $cur\_val \gets 0$
else begin if $cur\_val > penalty(equiv(m))$ then $cur\_val \gets penalty(equiv(m))$;
  $cur\_val \gets penalty(equiv(m) + cur\_val)$;
  end;
end
```

This code is used in section 423*.

1602* System-dependent changes. This section should be replaced, if necessary, by any special modifications of the program that are necessary to make TEX work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

1603*. Index. Here is where you can find all uses of each identifier in the program, with underlined entries pointing to where the identifier was defined. If the identifier is only one letter long, however, you get to see only the underlined entries. *All references are to section numbers instead of page numbers.*

This index also lists error messages and other aspects of the program that you might want to look up some day. For example, the entry for "system dependencies" lists all sections that should receive special attention from people who are installing TEX in a new operating environment. A list of various things that can't happen appears under "this can't happen". Approximately 40 sections are listed under "inner loop"; these account for about 60% of TEX's running time, exclusive of input and output.

The following sections were changed by the change file: 1, 2, 3, 15, 135, 141, 142, 147, 175, 184, 192, 196, 208, 209, 210, 212, 213, 215, 216, 230, 231, 232, 233, 236, 237, 264, 265, 266, 268, 273, 274, 275, 277, 278, 279, 281, 282, 284, 289, 294, 296, 298, 299, 303, 307, 311, 313, 314, 326, 328, 329, 331, 362, 366, 367, 377, 378, 382, 385, 386, 389, 409, 411, 412, 413, 415, 416, 417, 419, 420, 423, 424, 427, 461, 464, 465, 468, 469, 471, 472, 478, 482, 483, 487, 488, 494, 496, 498, 501, 505, 510, 536, 581, 616, 619, 620, 622, 623, 625, 626, 628, 632, 633, 637, 638, 649, 651, 687, 696, 727, 760, 762, 785, 791, 807, 808, 814, 815, 816, 819, 827, 829, 845, 846, 851, 852, 855, 863, 864, 866, 876, 877, 879, 880, 881, 890, 891, 896, 897, 898, 899, 934, 937, 952, 958, 960, 966, 968, 977, 979, 982, 991, 996, 999, 1012, 1014, 1021, 1023, 1026, 1070, 1071, 1075, 1077, 1079, 1080, 1081, 1082, 1096, 1101, 1105, 1108, 1110, 1130, 1138, 1145, 1146, 1147, 1185, 1189, 1191, 1192, 1194, 1199, 1202, 1203, 1204, 1205, 1206, 1208, 1209, 1211, 1212, 1213, 1218, 1221, 1224, 1225, 1226, 1227, 1236, 1237, 1238, 1239, 1240, 1241, 1246, 1247, 1248, 1257, 1292, 1293, 1295, 1296, 1307, 1308, 1311, 1312, 1324, 1325, 1335, 1336, 1337, 1362, 1379, 1380, 1381, 1382, 1383, 1384, 1385, 1386, 1387, 1388, 1389, 1390, 1391, 1392, 1393, 1394, 1395, 1396, 1397, 1398, 1399, 1400, 1401, 1402, 1403, 1404, 1405, 1406, 1407, 1408, 1409, 1410, 1411, 1412, 1413, 1414, 1415, 1416, 1417, 1418, 1419, 1420, 1421, 1422, 1423, 1424, 1425, 1426, 1427, 1428, 1429, 1430, 1431, 1432, 1433, 1434, 1435, 1436, 1437, 1438, 1439, 1440, 1441, 1442, 1443, 1444, 1445, 1446, 1447, 1448, 1449, 1450, 1451, 1452, 1453, 1454, 1455, 1456, 1457, 1458, 1459, 1460, 1461, 1462, 1463, 1464, 1465, 1466, 1467, 1468, 1469, 1470, 1471, 1472, 1473, 1474, 1475, 1476, 1477, 1478, 1479, 1480, 1481, 1482, 1483, 1484, 1485, 1486, 1487, 1488, 1489, 1490, 1491, 1492, 1493, 1494, 1495, 1496, 1497, 1498, 1499, 1500, 1501, 1502, 1503, 1504, 1505, 1506, 1507, 1508, 1509, 1510, 1511, 1512, 1513, 1514, 1515, 1516, 1517, 1518, 1519, 1520, 1521, 1522, 1523, 1524, 1525, 1526, 1527, 1528, 1529, 1530, 1531, 1532, 1533, 1534, 1535, 1536, 1537, 1538, 1539, 1540, 1541, 1542, 1543, 1544, 1545, 1546, 1547, 1548, 1549, 1550, 1551, 1552, 1553, 1554, 1555, 1556, 1557, 1558, 1559, 1560, 1561, 1562, 1563, 1564, 1565, 1566, 1567, 1568, 1569, 1570, 1571, 1572, 1573, 1574, 1575, 1576, 1577, 1578, 1579, 1580, 1581, 1582, 1583, 1584, 1585, 1586, 1587, 1588, 1589, 1590, 1591, 1592, 1593, 1594, 1595, 1596, 1597, 1598, 1599, 1600, 1601, 1602, 1603.