

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 proto \TeX 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 $\backslash halign$ 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 $\text{\TeX}82$ 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 $\text{\TeX}82$ 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 $\text{\TeX}82$ “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 $\text{\TeX}82$ 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 3.141592653', eTeX_version_string
    { printed when ε-TeX starts }

define pdftex_version ≡ 140 { \pdftexversion }
define pdftex_revision ≡ "26" { \pdftexrevision }
define pdftex_version_string ≡ '-1.40.26' { current pdfTeX version }
define pdfTeX_banner ≡ 'This is pdfTeX, Version 3.141592653', eTeX_version_string,
    pdftex_version_string { printed when pdfTeX starts }

define TeX_banner ≡ 'This is TeX, Version 3.141592653' { printed when TeX starts }
define banner ≡ pdfTeX_banner

define TEX ≡ PDFTEX { change program name into PDFTEX }
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 T_EX 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 and Experience* 6 (1976), 29–42. The T_EX 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-T_EX, 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.

4. The program begins with a normal Pascal program heading, whose components will be filled in later, using the conventions of WEB. For example, the portion of the program called ‘⟨ Global variables 13 ⟩’ below will be replaced by a sequence of variable declarations that starts in §13 of this documentation. In this way, we are able to define each individual global variable when we are prepared to understand what it means; we do not have to define all of the globals at once. Cross references in §13, where it says “See also sections 20, 26, . . . ,” also make it possible to look at the set of all global variables, if desired. Similar remarks apply to the other portions of the program heading.

Actually the heading shown here is not quite normal: The **program** line does not mention any *output* file, because Pascal-H would ask the T_EX user to specify a file name if *output* were specified here.

```
define mtype ≡ t@&y@&p@&e { this is a WEB coding trick: }
format mtype ≡ type { 'mtype' will be equivalent to 'type' }
format type ≡ true { but 'type' will not be treated as a reserved word }

⟨ Compiler directives 9 ⟩
program TEX; { all file names are defined dynamically }
  label ⟨ Labels in the outer block 6 ⟩
  const ⟨ Constants in the outer block 11 ⟩
  mtype ⟨ Types in the outer block 18 ⟩
  var ⟨ Global variables 13 ⟩

  procedure initialize; { this procedure gets things started properly }
    var ⟨ Local variables for initialization 19 ⟩
    begin ⟨ Initialize whatever TEX might access 8 ⟩
      end;

  ⟨ Basic printing procedures 57 ⟩
  ⟨ Error handling procedures 78 ⟩
```

5. The overall T_EX program begins with the heading just shown, after which comes a bunch of procedure declarations and function declarations. Finally we will get to the main program, which begins with the comment '*start_here*'. If you want to skip down to the main program now, you can look up '*start_here*' in the index. But the author suggests that the best way to understand this program is to follow pretty much the order of T_EX's components as they appear in the WEB description you are now reading, since the present ordering is intended to combine the advantages of the "bottom up" and "top down" approaches to the problem of understanding a somewhat complicated system.

6. Three labels must be declared in the main program, so we give them symbolic names.

```
define start_of_TEX = 1 { go here when TEX's variables are initialized }
define end_of_TEX = 9998 { go here to close files and terminate gracefully }
define final_end = 9999 { this label marks the ending of the program }
```

⟨ Labels in the outer block 6 ⟩ ≡

```
start_of_TEX, end_of_TEX, final_end; { key control points }
```

This code is used in section 4.

7. Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when T_EX is being installed or when system wizards are fooling around with T_EX without quite knowing what they are doing. Such code will not normally be compiled; it is delimited by the codewords 'debug ... gubed', with apologies to people who wish to preserve the purity of English.

Similarly, there is some conditional code delimited by 'stat ... tats' that is intended for use when statistics are to be kept about T_EX's memory usage. The stat ... tats code also implements diagnostic information for \tracingparagraphs, \tracingpages, and \tracingrestores.

```
define debug ≡ @{ { change this to 'debug ≡ ' when debugging }
define gubed ≡ @} { change this to 'gubed ≡ ' when debugging }
format debug ≡ begin
format gubed ≡ end
define stat ≡ @{ { change this to 'stat ≡ ' when gathering usage statistics }
define tats ≡ @} { change this to 'tats ≡ ' when gathering usage statistics }
format stat ≡ begin
format tats ≡ end
```

8. This program has two important variations: (1) There is a long and slow version called INITEX, which does the extra calculations needed to initialize T_EX's internal tables; and (2) there is a shorter and faster production version, which cuts the initialization to a bare minimum. Parts of the program that are needed in (1) but not in (2) are delimited by the codewords 'init ... tini'.

```
define init ≡ { change this to 'init ≡ @{' in the production version }
define tini ≡ { change this to 'tini ≡ @}' in the production version }
format init ≡ begin
format tini ≡ end
```

⟨ Initialize whatever T_EX might access 8 ⟩ ≡

⟨ Set initial values of key variables 21 ⟩

init ⟨ Initialize table entries (done by INITEX only) 182 ⟩ tini

This code is used in section 4.

9. If the first character of a Pascal comment is a dollar sign, Pascal-H treats the comment as a list of “compiler directives” that will affect the translation of this program into machine language. The directives shown below specify full checking and inclusion of the Pascal debugger when T_EX is being debugged, but they cause range checking and other redundant code to be eliminated when the production system is being generated. Arithmetic overflow will be detected in all cases.

```
< Compiler directives 9 > ≡
@{ @&$C-, A+, D-@} { no range check, catch arithmetic overflow, no debug overhead }
debug @{ @&$C+, D+@} gubed { but turn everything on when debugging }
```

This code is used in section 4.

10. This T_EX implementation conforms to the rules of the *Pascal User Manual* published by Jensen and Wirth in 1975, except where system-dependent code is necessary to make a useful system program, and except in another respect where such conformity would unnecessarily obscure the meaning and clutter up the code: We assume that **case** statements may include a default case that applies if no matching label is found. Thus, we shall use constructions like

```
case x of
  1: < code for x = 1 >;
  3: < code for x = 3 >;
  others { code for x ≠ 1 and x ≠ 3 }
endcases
```

since most Pascal compilers have plugged this hole in the language by incorporating some sort of default mechanism. For example, the Pascal-H compiler allows ‘others:’ as a default label, and other Pascals allow syntaxes like ‘else’ or ‘otherwise’ or ‘otherwise:’, etc. The definitions of **othercases** and **endcases** should be changed to agree with local conventions. Note that no semicolon appears before **endcases** in this program, so the definition of **endcases** should include a semicolon if the compiler wants one. (Of course, if no default mechanism is available, the **case** statements of T_EX will have to be laboriously extended by listing all remaining cases. People who are stuck with such Pascals have, in fact, done this, successfully but not happily!)

```
define othercases ≡ others: { default for cases not listed explicitly }
define endcases ≡ end { follows the default case in an extended case statement }
format othercases ≡ else
format endcases ≡ end
```

11. The following parameters can be changed at compile time to extend or reduce TeX's capacity. They may have different values in INITEX and in production versions of TeX.

`< Constants in the outer block 11 > =`

```

mem_max = 30000;
    { greatest index in TeX's internal mem array; must be strictly less than max_halfword; must be
      equal to mem_top in INITEX, otherwise  $\geq$  mem_top }

mem_min = 0; { smallest index in TeX's internal mem array; must be min_halfword or more; must be
      equal to mem_bot in INITEX, otherwise  $\leq$  mem_bot }

buf_size = 500; { maximum number of characters simultaneously present in current lines of open files
      and in control sequences between \csname and \endcsname; must not exceed max_halfword }

error_line = 72; { width of context lines on terminal error messages }

half_error_line = 42; { width of first lines of contexts in terminal error messages; should be between 30
      and error_line - 15 }

max_print_line = 79; { width of longest text lines output; should be at least 60 }

stack_size = 200; { maximum number of simultaneous input sources }

max_in_open = 6;
    { maximum number of input files and error insertions that can be going on simultaneously }

font_max = 75; { maximum internal font number; must not exceed max_quarterword and must be at
      most font_base + 256 }

font_mem_size = 20000; { number of words of font.info for all fonts }

param_size = 60; { maximum number of simultaneous macro parameters }

nest_size = 40; { maximum number of semantic levels simultaneously active }

max_strings = 3000; { maximum number of strings; must not exceed max_halfword }

string_vacancies = 8000; { the minimum number of characters that should be available for the user's
      control sequences and font names, after TeX's own error messages are stored }

pool_size = 32000; { maximum number of characters in strings, including all error messages and help
      texts, and the names of all fonts and control sequences; must exceed string_vacancies by the total
      length of TeX's own strings, which is currently about 23000 }

save_size = 600; { space for saving values outside of current group; must be at most max_halfword }

trie_size = 8000; { space for hyphenation patterns; should be larger for INITEX than it is in production
      versions of TeX }

trie_op_size = 500; { space for "opcodes" in the hyphenation patterns }

dvi_buf_size = 800; { size of the output buffer; must be a multiple of 8 }

file_name_size = 40; { file names shouldn't be longer than this }

pool_name = 'TeXformats:TEX.POOL'.
    { string of length file_name_size; tells where the string pool appears }
```

See also sections 675, 679, 695, 721, and 1631.

This code is used in section 4.

12. Like the preceding parameters, the following quantities can be changed at compile time to extend or reduce T_EX's capacity. But if they are changed, it is necessary to rerun the initialization program INITEX to generate new tables for the production T_EX program. One can't simply make helter-skelter changes to the following constants, since certain rather complex initialization numbers are computed from them. They are defined here using WEB macros, instead of being put into Pascal's **const** list, in order to emphasize this distinction.

```
define mem_bot = 0
  { smallest index in the mem array dumped by INITEX; must not be less than mem_min }
define mem_top ≡ 30000 { largest index in the mem array dumped by INITEX; must be substantially
  larger than mem_bot and not greater than mem_max }
define font_base = 0 { smallest internal font number; must not be less than min_quarterword }
define hash_size = 2100 { maximum number of control sequences; it should be at most about
  (mem_max − mem_min)/10 }
define hash_prime = 1777 { a prime number equal to about 85% of hash_size }
define hyph_size = 307 { another prime; the number of \hyphenation exceptions }
```

13. In case somebody has inadvertently made bad settings of the "constants," T_EX checks them using a global variable called *bad*.

This is the first of many sections of T_EX where global variables are defined.

<Global variables 13> ≡
bad: integer; { is some "constant" wrong? }

See also sections 20, 26, 30, 32, 39, 50, 54, 73, 76, 79, 96, 104, 110, 117, 133, 134, 135, 136, 142, 183, 191, 199, 231, 264, 271, 274, 275, 293, 308, 319, 323, 326, 327, 330, 331, 332, 355, 383, 389, 408, 413, 414, 436, 464, 473, 506, 515, 519, 538, 539, 546, 553, 558, 565, 575, 576, 581, 619, 622, 623, 643, 676, 680, 687, 691, 696, 701, 704, 708, 710, 723, 774, 811, 818, 819, 821, 829, 837, 860, 895, 900, 940, 946, 990, 997, 999, 1001, 1004, 1009, 1015, 1023, 1048, 1069, 1077, 1082, 1084, 1098, 1103, 1120, 1124, 1127, 1148, 1157, 1159, 1166, 1209, 1252, 1444, 1459, 1477, 1483, 1511, 1522, 1525, 1543, 1547, 1550, 1557, 1559, 1570, 1583, 1628, 1633, 1640, 1652, 1660, 1705, 1750, 1773, 1814, 1816, 1835, 1842, 1858, and 1859.

This code is used in section 4.

14. Later on we will say '**if** *mem_max* ≥ *max_halfword* **then** *bad* ← 14', or something similar. (We can't do that until *max_halfword* has been defined.)

<Check the "constant" values for consistency 14> ≡
bad ← 0;
if (*half_error_line* < 30) ∨ (*half_error_line* > *error_line* − 15) **then** *bad* ← 1;
if *max_print_line* < 60 **then** *bad* ← 2;
if *dvi_buf_size* mod 8 ≠ 0 **then** *bad* ← 3;
if *mem_bot* + 1100 > *mem_top* **then** *bad* ← 4;
if *hash_prime* > *hash_size* **then** *bad* ← 5;
if *max_in_open* ≥ 128 **then** *bad* ← 6;
if *mem_top* < 256 + 11 **then** *bad* ← 7; { we will want *null_list* > 255 }

See also sections 129, 312, 548, and 1427.

This code is used in section 1512.

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 }
```

16. Here are some macros for common programming idioms.

```
define incr(#) ≡ # ← # + 1 { increase a variable by unity }
define decr(#) ≡ # ← # - 1 { decrease a variable by unity }
define negate(#) ≡ # ← -# { change the sign of a variable }
define loop ≡ while true do { repeat over and over until a goto happens }
format loop ≡ xclause { WEB's xclause acts like 'while true do' }
define do_nothing ≡ { empty statement }
define return ≡ goto exit { terminate a procedure call }
format return ≡ nil
define empty = 0 { symbolic name for a null constant }
```

17. The character set. In order to make \TeX readily portable to a wide variety of computers, all of its input text is converted to an internal eight-bit code that includes standard ASCII, the “American Standard Code for Information Interchange.” This conversion is done immediately when each character is read in. Conversely, characters are converted from ASCII to the user’s external representation just before they are output to a text file.

Such an internal code is relevant to users of \TeX primarily because it governs the positions of characters in the fonts. For example, the character ‘A’ has ASCII code $65 = '101$, and when \TeX typesets this letter it specifies character number 65 in the current font. If that font actually has ‘A’ in a different position, \TeX doesn’t know what the real position is; the program that does the actual printing from \TeX ’s device-independent files is responsible for converting from ASCII to a particular font encoding.

\TeX ’s internal code also defines the value of constants that begin with a reverse apostrophe; and it provides an index to the `\catcode`, `\mathcode`, `\uccode`, `\lccode`, and `\delcode` tables.

18. Characters of text that have been converted to \TeX ’s internal form are said to be of type *ASCII_code*, which is a subrange of the integers.

$\langle \text{Types in the outer block } 18 \rangle \equiv$

$\text{ASCII_code} = 0 \dots 255; \{ \text{eight-bit numbers} \}$

See also sections 25, 38, 101, 109, 131, 168, 230, 291, 322, 574, 621, 694, 707, 722, 1097, 1102, 1627, 1632, and 1678.

This code is used in section 4.

19. The original Pascal compiler was designed in the late 60s, when six-bit character sets were common, so it did not make provision for lowercase letters. Nowadays, of course, we need to deal with both capital and small letters in a convenient way, especially in a program for typesetting; so the present specification of \TeX has been written under the assumption that the Pascal compiler and run-time system permit the use of text files with more than 64 distinguishable characters. More precisely, we assume that the character set contains at least the letters and symbols associated with ASCII codes ‘40 through ‘176; all of these characters are now available on most computer terminals.

Since we are dealing with more characters than were present in the first Pascal compilers, we have to decide what to call the associated data type. Some Pascals use the original name *char* for the characters in text files, even though there now are more than 64 such characters, while other Pascals consider *char* to be a 64-element subrange of a larger data type that has some other name.

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

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

$\langle \text{Local variables for initialization } 19 \rangle \equiv$

i: integer;

See also sections 181 and 1104.

This code is used in section 4.

20. The \TeX processor converts between ASCII code and the user’s external character set by means of arrays *xord* and *xchr* that are analogous to Pascal’s *ord* and *chr* functions.

$\langle \text{Global variables } 13 \rangle +\equiv$

```
xord: array [text_char] of ASCII_code; { specifies conversion of input characters }
xchr: array [ASCII_code] of text_char; { specifies conversion of output characters }
```

21. Since we are assuming that our Pascal system is able to read and write the visible characters of standard ASCII (although not necessarily using the ASCII codes to represent them), the following assignment statements initialize the standard part of the *xchr* array properly, without needing any system-dependent changes. On the other hand, it is possible to implement TeX with less complete character sets, and in such cases it will be necessary to change something here.

{ Set initial values of key variables 21 } ≡

```

xchr[40] ← ' `; xchr[41] ← '!'; xchr[42] ← '"'; xchr[43] ← '#'; xchr[44] ← '$';
xchr[45] ← '%'; xchr[46] ← '&'; xchr[47] ← '---';
xchr[50] ← '(`; xchr[51] ← ')'; xchr[52] ← '*''; xchr[53] ← '+`'; xchr[54] ← ',`';
xchr[55] ← '-`'; xchr[56] ← '.`'; xchr[57] ← '/`';
xchr[60] ← '0`'; xchr[61] ← '1`'; xchr[62] ← '2`'; xchr[63] ← '3`'; xchr[64] ← '4`';
xchr[65] ← '5`'; xchr[66] ← '6`'; xchr[67] ← '7`';
xchr[70] ← '8`'; xchr[71] ← '9`'; xchr[72] ← ':`'; xchr[73] ← ';`'; xchr[74] ← '<`;
xchr[75] ← '='`'; xchr[76] ← '>`'; xchr[77] ← '?`';
xchr[100] ← '@`'; xchr[101] ← 'A`'; xchr[102] ← 'B`'; xchr[103] ← 'C`'; xchr[104] ← 'D`';
xchr[105] ← 'E`'; xchr[106] ← 'F`'; xchr[107] ← 'G`';
xchr[110] ← 'H`'; xchr[111] ← 'I`'; xchr[112] ← 'J`'; xchr[113] ← 'K`'; xchr[114] ← 'L`';
xchr[115] ← 'M`'; xchr[116] ← 'N`'; xchr[117] ← 'O`';
xchr[120] ← 'P`'; xchr[121] ← 'Q`'; xchr[122] ← 'R`'; xchr[123] ← 'S`'; xchr[124] ← 'T`';
xchr[125] ← 'U`'; xchr[126] ← 'V`'; xchr[127] ← 'W`';
xchr[130] ← 'X`'; xchr[131] ← 'Y`'; xchr[132] ← 'Z`'; xchr[133] ← '['`'; xchr[134] ← '\`';
xchr[135] ← ']'`'; xchr[136] ← '^`'; xchr[137] ← '_`';
xchr[140] ← '`'; xchr[141] ← 'a`'; xchr[142] ← 'b`'; xchr[143] ← 'c`'; xchr[144] ← 'd`';
xchr[145] ← 'e`'; xchr[146] ← 'f`'; xchr[147] ← 'g`';
xchr[150] ← 'h`'; xchr[151] ← 'i`'; xchr[152] ← 'j`'; xchr[153] ← 'k`'; xchr[154] ← 'l`';
xchr[155] ← 'm`'; xchr[156] ← 'n`'; xchr[157] ← 'o`';
xchr[160] ← 'p`'; xchr[161] ← 'q`'; xchr[162] ← 'r`'; xchr[163] ← 's`'; xchr[164] ← 't`';
xchr[165] ← 'u`'; xchr[166] ← 'v`'; xchr[167] ← 'w`';
xchr[170] ← 'x`'; xchr[171] ← 'y`'; xchr[172] ← 'z`'; xchr[173] ← '{`'; xchr[174] ← '|`';
xchr[175] ← '}`'; xchr[176] ← '`';

```

See also sections 23, 24, 74, 77, 80, 97, 118, 184, 233, 272, 276, 294, 309, 390, 409, 465, 507, 516, 547, 577, 582, 620, 623, 633, 677, 681, 688, 697, 709, 711, 724, 820, 830, 838, 861, 947, 1105, 1167, 1210, 1445, 1460, 1478, 1523, 1551, 1571, 1584, 1629, 1634, 1706, 1751, 1817, 1836, and 1860.

This code is used in section 8.

22. Some of the ASCII codes without visible characters have been given symbolic names in this program because they are used with a special meaning.

```

define null_code = '0' { ASCII code that might disappear }
define carriage_return = '15' { ASCII code used at end of line }
define invalid_code = '177' { ASCII code that many systems prohibit in text files }

```

23. The ASCII code is “standard” only to a certain extent, since many computer installations have found it advantageous to have ready access to more than 94 printing characters. Appendix C of *The TeXbook* gives a complete specification of the intended correspondence between characters and TeX’s internal representation.

If TeX is being used on a garden-variety Pascal for which only standard ASCII codes will appear in the input and output files, it doesn’t really matter what codes are specified in $xchr[0 \dots '37]$, but the safest policy is to blank everything out by using the code shown below.

However, other settings of $xchr$ will make TeX more friendly on computers that have an extended character set, so that users can type things like ‘#’ instead of ‘\ne’. People with extended character sets can assign codes arbitrarily, giving an $xchr$ equivalent to whatever characters the users of TeX are allowed to have in their input files. It is best to make the codes correspond to the intended interpretations as shown in Appendix C whenever possible; but this is not necessary. For example, in countries with an alphabet of more than 26 letters, it is usually best to map the additional letters into codes less than ‘40’. To get the most “permissive” character set, change ‘_’ on the right of these assignment statements to $chr(i)$.

```
< Set initial values of key variables 21 > +≡
  for i ← 0 to '37' do xchr[i] ← '_';
  for i ← '177' to '377' do xchr[i] ← '_';
```

24. The following system-independent code makes the $xord$ array contain a suitable inverse to the information in $xchr$. Note that if $xchr[i] = xchr[j]$ where $i < j < '177$, the value of $xord[xchr[i]]$ will turn out to be j or more; hence, standard ASCII code numbers will be used instead of codes below ‘40’ in case there is a coincidence.

```
< Set initial values of key variables 21 > +≡
  for i ← first_text_char to last_text_char do xord[chr(i)] ← invalid_code;
  for i ← '200' to '377' do xord[xchr[i]] ← i;
  for i ← 0 to '176' do xord[xchr[i]] ← i;
```

25. Input and output. The bane of portability is the fact that different operating systems treat input and output quite differently, perhaps because computer scientists have not given sufficient attention to this problem. People have felt somehow that input and output are not part of “real” programming. Well, it is true that some kinds of programming are more fun than others. With existing input/output conventions being so diverse and so messy, the only sources of joy in such parts of the code are the rare occasions when one can find a way to make the program a little less bad than it might have been. We have two choices, either to attack I/O now and get it over with, or to postpone I/O until near the end. Neither prospect is very attractive, so let’s get it over with.

The basic operations we need to do are (1) inputting and outputting of text, to or from a file or the user’s terminal; (2) inputting and outputting of eight-bit bytes, to or from a file; (3) instructing the operating system to initiate (“open”) or to terminate (“close”) input or output from a specified file; (4) testing whether the end of an input file has been reached.

T_EX needs to deal with two kinds of files. We shall use the term *alpha_file* for a file that contains textual data, and the term *byte_file* for a file that contains eight-bit binary information. These two types turn out to be the same on many computers, but sometimes there is a significant distinction, so we shall be careful to distinguish between them. Standard protocols for transferring such files from computer to computer, via high-speed networks, are now becoming available to more and more communities of users.

The program actually makes use also of a third kind of file, called a *word_file*, when dumping and reloading base information for its own initialization. We shall define a word file later; but it will be possible for us to specify simple operations on word files before they are defined.

⟨ Types in the outer block 18 ⟩ +≡

```
eight_bits = 0 .. 255; { unsigned one-byte quantity }
alpha_file = packed file of text_char; { files that contain textual data }
byte_file = packed file of eight_bits; { files that contain binary data }
```

26. Most of what we need to do with respect to input and output can be handled by the I/O facilities that are standard in Pascal, i.e., the routines called *get*, *put*, *eof*, and so on. But standard Pascal does not allow file variables to be associated with file names that are determined at run time, so it cannot be used to implement T_EX; some sort of extension to Pascal’s ordinary *reset* and *rewrite* is crucial for our purposes. We shall assume that *name_of_file* is a variable of an appropriate type such that the Pascal run-time system being used to implement T_EX can open a file whose external name is specified by *name_of_file*.

⟨ Global variables 13 ⟩ +≡

```
name_of_file: packed array [1 .. file_name_size] of char;
{ on some systems this may be a record variable }
name_length: 0 .. file_name_size;
{ this many characters are actually relevant in name_of_file (the rest are blank) }
```

27. The Pascal-H compiler with which the present version of T_EX was prepared has extended the rules of Pascal in a very convenient way. To open file *f*, we can write

<i>reset(f, name, '/0')</i>	for input;
<i>rewrite(f, name, '/0')</i>	for output.

The '*name*' parameter, which is of type 'packed array [*any*] of char', stands for the name of the external file that is being opened for input or output. Blank spaces that might appear in *name* are ignored.

The '/0' parameter tells the operating system not to issue its own error messages if something goes wrong. If a file of the specified name cannot be found, or if such a file cannot be opened for some other reason (e.g., someone may already be trying to write the same file), we will have *erstat(f) ≠ 0* after an unsuccessful *reset* or *rewrite*. This allows T_EX to undertake appropriate corrective action.

T_EX's file-opening procedures return *false* if no file identified by *name_of_file* could be opened.

```
define reset_OK(#) ≡ erstat(#) = 0
define rewrite_OK(#) ≡ erstat(#) = 0

function a_open_in(var f : alpha_file): boolean; { open a text file for input }
begin reset(f, name_of_file, '/0'); a_open_in ← reset_OK(f);
end;

function a_open_out(var f : alpha_file): boolean; { open a text file for output }
begin rewrite(f, name_of_file, '/0'); a_open_out ← rewrite_OK(f);
end;

function b_open_in(var f : byte_file): boolean; { open a binary file for input }
begin reset(f, name_of_file, '/0'); b_open_in ← reset_OK(f);
end;

function b_open_out(var f : byte_file): boolean; { open a binary file for output }
begin rewrite(f, name_of_file, '/0'); b_open_out ← rewrite_OK(f);
end;

function w_open_in(var f : word_file): boolean; { open a word file for input }
begin reset(f, name_of_file, '/0'); w_open_in ← reset_OK(f);
end;

function w_open_out(var f : word_file): boolean; { open a word file for output }
begin rewrite(f, name_of_file, '/0'); w_open_out ← rewrite_OK(f);
end;
```

28. Files can be closed with the Pascal-H routine '*close(f)*', which should be used when all input or output with respect to *f* has been completed. This makes *f* available to be opened again, if desired; and if *f* was used for output, the *close* operation makes the corresponding external file appear on the user's area, ready to be read.

These procedures should not generate error messages if a file is being closed before it has been successfully opened.

```
procedure a_close(var f : alpha_file); { close a text file }
begin close(f);
end;

procedure b_close(var f : byte_file); { close a binary file }
begin close(f);
end;

procedure w_close(var f : word_file); { close a word file }
begin close(f);
end;
```

29. Binary input and output are done with Pascal's ordinary *get* and *put* procedures, so we don't have to make any other special arrangements for binary I/O. Text output is also easy to do with standard Pascal routines. The treatment of text input is more difficult, however, because of the necessary translation to *ASCII_code* values. T_EX's conventions should be efficient, and they should blend nicely with the user's operating environment.

30. Input from text files is read one line at a time, using a routine called *input_ln*. This function is defined in terms of global variables called *buffer*, *first*, and *last* that will be described in detail later; for now, it suffices for us to know that *buffer* is an array of *ASCII_code* values, and that *first* and *last* are indices into this array representing the beginning and ending of a line of text.

```
<Global variables 13> +≡  
buffer: array [0 .. buf_size] of ASCII_code; { lines of characters being read }  
first: 0 .. buf_size; { the first unused position in buffer }  
last: 0 .. buf_size; { end of the line just input to buffer }  
max_buf_stack: 0 .. buf_size; { largest index used in buffer }
```

31. The *input_ln* function brings the next line of input from the specified file into available positions of the buffer array and returns the value *true*, unless the file has already been entirely read, in which case it returns *false* and sets *last* \leftarrow *first*. In general, the *ASCII_code* numbers that represent the next line of the file are input into *buffer*[*first*], *buffer*[*first* + 1], ..., *buffer*[*last* - 1]; and the global variable *last* is set equal to *first* plus the length of the line. Trailing blanks are removed from the line; thus, either *last* = *first* (in which case the line was entirely blank) or *buffer*[*last* - 1] \neq " ".

An overflow error is given, however, if the normal actions of *input_ln* would make *last* \geq *buf_size*; this is done so that other parts of TeX can safely look at the contents of *buffer*[*last* + 1] without overstepping the bounds of the *buffer* array. Upon entry to *input_ln*, the condition *first* < *buf_size* will always hold, so that there is always room for an "empty" line.

The variable *max_buf_stack*, which is used to keep track of how large the *buf_size* parameter must be to accommodate the present job, is also kept up to date by *input_ln*.

If the *bypass_eoln* parameter is *true*, *input_ln* will do a *get* before looking at the first character of the line; this skips over an *eoln* that was in *f*↑. The procedure does not do a *get* when it reaches the end of the line; therefore it can be used to acquire input from the user's terminal as well as from ordinary text files.

Standard Pascal says that a file should have *eoln* immediately before *eof*, but TeX needs only a weaker restriction: If *eof* occurs in the middle of a line, the system function *eoln* should return a *true* result (even though *f*↑ will be undefined).

Since the inner loop of *input_ln* is part of TeX's "inner loop"—each character of input comes in at this place—it is wise to reduce system overhead by making use of special routines that read in an entire array of characters at once, if such routines are available. The following code uses standard Pascal to illustrate what needs to be done, but finer tuning is often possible at well-developed Pascal sites.

```
function input_ln(var f : alpha_file; bypass_eoln : boolean): boolean;
  { inputs the next line or returns false }
  var last_nonblank: 0 .. buf_size;  { last with trailing blanks removed }
  begin if bypass_eoln then
    if  $\neg$ eof(f) then get(f);  { input the first character of the line into f↑ }
    last  $\leftarrow$  first;  { cf. Matthew 19:30 }
    if eof(f) then input_ln  $\leftarrow$  false
    else begin last_nonblank  $\leftarrow$  first;
      while  $\neg$ eoln(f) do
        begin if last  $\geq$  max_buf_stack then
          begin max_buf_stack  $\leftarrow$  last + 1;
            if max_buf_stack = buf_size then <Report overflow of the input buffer, and abort 35>;
            end;
            buffer[last]  $\leftarrow$  xord[f↑]; get(f); incr(last);
            if buffer[last - 1]  $\neq$  " " then last_nonblank  $\leftarrow$  last;
          end;
          last  $\leftarrow$  last_nonblank; input_ln  $\leftarrow$  true;
        end;
      end;
    end;
  end;
```

32. The user's terminal acts essentially like other files of text, except that it is used both for input and for output. When the terminal is considered an input file, the file variable is called *term_in*, and when it is considered an output file the file variable is *term_out*.

```
<Global variables 13> +≡
term_in: alpha_file;  { the terminal as an input file }
term_out: alpha_file;  { the terminal as an output file }
```

33. Here is how to open the terminal files in Pascal-H. The '/I' switch suppresses the first *get*.

```
define t_open_in ≡ reset(term_in, 'TTY:', '/0/I') { open the terminal for text input }
define t_open_out ≡ rewrite(term_out, 'TTY:', '/0') { open the terminal for text output }
```

34. Sometimes it is necessary to synchronize the input/output mixture that happens on the user's terminal, and three system-dependent procedures are used for this purpose. The first of these, *update_terminal*, is called when we want to make sure that everything we have output to the terminal so far has actually left the computer's internal buffers and been sent. The second, *clear_terminal*, is called when we wish to cancel any input that the user may have typed ahead (since we are about to issue an unexpected error message). The third, *wake_up_terminal*, is supposed to revive the terminal if the user has disabled it by some instruction to the operating system. The following macros show how these operations can be specified in Pascal-H:

```
define update_terminal ≡ break(term_out) { empty the terminal output buffer }
define clear_terminal ≡ break_in(term_in, true) { clear the terminal input buffer }
define wake_up_terminal ≡ do_nothing { cancel the user's cancellation of output }
```

35. We need a special routine to read the first line of TeX input from the user's terminal. This line is different because it is read before we have opened the transcript file; there is sort of a "chicken and egg" problem here. If the user types '\input paper' on the first line, or if some macro invoked by that line does such an \input, the transcript file will be named 'paper.log'; but if no \input commands are performed during the first line of terminal input, the transcript file will acquire its default name 'texput.log'. (The transcript file will not contain error messages generated by the first line before the first \input command.)

The first line is even more special if we are lucky enough to have an operating system that treats TeX differently from a run-of-the-mill Pascal object program. It's nice to let the user start running a TeX job by typing a command line like 'tex paper'; in such a case, TeX will operate as if the first line of input were 'paper', i.e., the first line will consist of the remainder of the command line, after the part that invoked TeX.

The first line is special also because it may be read before TeX has input a format file. In such cases, normal error messages cannot yet be given. The following code uses concepts that will be explained later. (If the Pascal compiler does not support non-local *goto*, the statement '*goto final_end*' should be replaced by something that quietly terminates the program.)

```
< Report overflow of the input buffer, and abort 35 > ≡
if format_ident = 0 then
  begin write_ln(term_out, 'Buffer_size_exceeded!'); goto final_end;
  end
else begin cur_input.loc_field ← first; cur_input.limit_field ← last - 1;
  overflow("buffer_size", buf_size);
  end
```

This code is used in sections 31 and 1756.

36. Different systems have different ways to get started. But regardless of what conventions are adopted, the routine that initializes the terminal should satisfy the following specifications:

- 1) It should open file *term_in* for input from the terminal. (The file *term_out* will already be open for output to the terminal.)
- 2) If the user has given a command line, this line should be considered the first line of terminal input. Otherwise the user should be prompted with '**', and the first line of input should be whatever is typed in response.
- 3) The first line of input, which might or might not be a command line, should appear in locations *first* to *last - 1* of the *buffer* array.
- 4) The global variable *loc* should be set so that the character to be read next by TEX is in *buffer[loc]*. This character should not be blank, and we should have *loc < last*.

(It may be necessary to prompt the user several times before a non-blank line comes in. The prompt is '**' instead of the later '*' because the meaning is slightly different: '\input' need not be typed immediately after '**'.)

```
define loc ≡ cur_input.loc_field { location of first unread character in buffer }
```

37. The following program does the required initialization without retrieving a possible command line. It should be clear how to modify this routine to deal with command lines, if the system permits them.

```
function init_terminal: boolean; { gets the terminal input started }
label exit;
begin t_open_in;
loop begin wake_up_terminal; write(term_out, '**'); update_terminal;
if ~input_ln(term_in, true) then { this shouldn't happen }
begin write_ln(term_out); write(term_out, '!EndOfFileOnTheTerminal...why?');
init_terminal ← false; return;
end;
loc ← first;
while (loc < last) ∧ (buffer[loc] = " ") do incr(loc);
if loc < last then
begin init_terminal ← true; return; { return unless the line was all blank }
end;
write_ln(term_out, 'Please type the name of your input file.');
end;
exit: end;
```

38. String handling. Control sequence names and diagnostic messages are variable-length strings of eight-bit characters. Since Pascal does not have a well-developed string mechanism, TeX does all of its string processing by homegrown methods.

Elaborate facilities for dynamic strings are not needed, so all of the necessary operations can be handled with a simple data structure. The array *str_pool* contains all of the (eight-bit) ASCII codes in all of the strings, and the array *str_start* contains indices of the starting points of each string. Strings are referred to by integer numbers, so that string number *s* comprises the characters *str_pool*[*j*] for *str_start*[*s*] ≤ *j* < *str_start*[*s* + 1]. Additional integer variables *pool_ptr* and *str_ptr* indicate the number of entries used so far in *str_pool* and *str_start*, respectively; locations *str_pool*[*pool_ptr*] and *str_start*[*str_ptr*] are ready for the next string to be allocated.

String numbers 0 to 255 are reserved for strings that correspond to single ASCII characters. This is in accordance with the conventions of WEB, which converts single-character strings into the ASCII code number of the single character involved, while it converts other strings into integers and builds a string pool file. Thus, when the string constant ". ." appears in the program below, WEB converts it into the integer 46, which is the ASCII code for a period, while WEB will convert a string like "hello" into some integer greater than 255. String number 46 will presumably be the single character '.'; but some ASCII codes have no standard visible representation, and TeX sometimes needs to be able to print an arbitrary ASCII character, so the first 256 strings are used to specify exactly what should be printed for each of the 256 possibilities.

Elements of the *str_pool* array must be ASCII codes that can actually be printed; i.e., they must have an *xchr* equivalent in the local character set. (This restriction applies only to preloaded strings, not to those generated dynamically by the user.)

Some Pascal compilers won't pack integers into a single byte unless the integers lie in the range –128 .. 127. To accommodate such systems we access the string pool only via macros that can easily be redefined.

```
define si(#) ≡ # { convert from ASCII_code to packed_ASCII_code }
define so(#) ≡ # { convert from packed_ASCII_code to ASCII_code }

⟨ Types in the outer block 18 ⟩ +≡
pool_pointer = 0 .. pool_size; { for variables that point into str_pool }
str_number = 0 .. max_strings; { for variables that point into str_start }
packed_ASCII_code = 0 .. 255; { elements of str_pool array }
```

39. ⟨ Global variables 13 ⟩ +≡

```
str_pool: packed array [pool_pointer] of packed_ASCII_code; { the characters }
str_start: array [str_number] of pool_pointer; { the starting pointers }
pool_ptr: pool_pointer; { first unused position in str_pool }
str_ptr: str_number; { number of the current string being created }
init_pool_ptr: pool_pointer; { the starting value of pool_ptr }
init_str_ptr: str_number; { the starting value of str_ptr }
```

40. Several of the elementary string operations are performed using WEB macros instead of Pascal procedures, because many of the operations are done quite frequently and we want to avoid the overhead of procedure calls. For example, here is a simple macro that computes the length of a string.

```
define length(#) ≡ (str_start[# + 1] – str_start[#]) { the number of characters in string number # }
```

41. The length of the current string is called *cur_length*:

```
define cur_length ≡ (pool_ptr – str_start[str_ptr])
```

42. Strings are created by appending character codes to *str_pool*. The *append_char* macro, defined here, does not check to see if the value of *pool_ptr* has gotten too high; this test is supposed to be made before *append_char* is used. There is also a *flush_char* macro, which erases the last character appended.

To test if there is room to append *l* more characters to *str_pool*, we shall write *str_room(l)*, which aborts TeX and gives an apologetic error message if there isn't enough room.

```
define append_char(#) ≡ { put ASCII_code # at the end of str_pool }
begin str_pool[pool_ptr] ← si(#); incr(pool_ptr);
end
define flush_char ≡ decr(pool_ptr) { forget the last character in the pool }
define str_room(#) ≡ { make sure that the pool hasn't overflowed }
begin if pool_ptr + # > pool_size then overflow("pool_size", pool_size - init_pool_ptr);
end
```

43. Once a sequence of characters has been appended to *str_pool*, it officially becomes a string when the function *make_string* is called. This function returns the identification number of the new string as its value.

```
function make_string: str_number; { current string enters the pool }
begin if str_ptr = max_strings then overflow("number_of_strings", max_strings - init_str_ptr);
incr(str_ptr); str_start[str_ptr] ← pool_ptr; make_string ← str_ptr - 1;
end;
```

44. To destroy the most recently made string, we say *flush_string*.

```
define flush_string ≡
begin decr(str_ptr); pool_ptr ← str_start[str_ptr];
end
```

45. The following subroutine compares string *s* with another string of the same length that appears in *buffer* starting at position *k*; the result is *true* if and only if the strings are equal. Empirical tests indicate that *str_eq_buf* is used in such a way that it tends to return *true* about 80 percent of the time.

```
function str_eq_buf(s : str_number; k : integer): boolean; { test equality of strings }
label not_found; { loop exit }
var j: pool_pointer; { running index }
result: boolean; { result of comparison }
begin j ← str_start[s];
while j < str_start[s + 1] do
begin if so(str_pool[j]) ≠ buffer[k] then
begin result ← false; goto not_found;
end;
incr(j); incr(k);
end;
result ← true;
not_found: str_eq_buf ← result;
end;
```

46. Here is a similar routine, but it compares two strings in the string pool, and it does not assume that they have the same length.

```
function str_eq_str(s, t : str_number): boolean; { test equality of strings }
label not_found; { loop exit }
var j, k: pool_pointer; { running indices }
result: boolean; { result of comparison }
begin result ← false;
if length(s) ≠ length(t) then goto not_found;
j ← str_start[s]; k ← str_start[t];
while j < str_start[s + 1] do
begin if str_pool[j] ≠ str_pool[k] then goto not_found;
incr(j); incr(k);
end;
result ← true;
not_found: str_eq_str ← result;
end;
```

47. The initial values of *str_pool*, *str_start*, *pool_ptr*, and *str_ptr* are computed by the INITEX program, based in part on the information that WEB has output while processing TeX.

```
init function get_strings_started: boolean;
{ initializes the string pool, but returns false if something goes wrong }
label done, exit;
var k, l: 0 .. 255; { small indices or counters }
m, n: text_char; { characters input from pool_file }
g: str_number; { garbage }
a: integer; { accumulator for check sum }
c: boolean; { check sum has been checked }
begin pool_ptr ← 0; str_ptr ← 0; str_start[0] ← 0; { Make the first 256 strings 48 };
{ Read the other strings from the TEX.POOL file and return true, or give an error message and return
false 51 };
exit: end;
tini
```

48. define app_lc_hex(#) ≡ l ← #;

if $l < 10$ then append_char($l + "0"$) else append_char($l - 10 + "a"$)

{ Make the first 256 strings 48 } ≡

```
for k ← 0 to 255 do
begin if ((Character k cannot be printed 49)) then
begin append_char("^"); append_char("^");
if  $k < '100$  then append_char( $k + '100$ )
else if  $k < '200$  then append_char( $k - '100$ )
else begin app_lc_hex( $k \text{ div } 16$ ); app_lc_hex( $k \text{ mod } 16$ );
end;
end
else append_char(k);
g ← make_string;
end
```

This code is used in section 47.

49. The first 128 strings will contain 95 standard ASCII characters, and the other 33 characters will be printed in three-symbol form like ‘`^^A`’ unless a system-dependent change is made here. Installations that have an extended character set, where for example `xchr[32] = '#'`, would like string ‘`32`’ to be the single character ‘`32`’ instead of the three characters ‘`136, 136, 132`’ (‘`Z`’). On the other hand, even people with an extended character set will want to represent string ‘`15`’ by ‘`^M`’, since ‘`15`’ is *carriage_return*; the idea is to produce visible strings instead of tabs or line-feeds or carriage-returns or bell-rings or characters that are treated anomalously in text files.

Unprintable characters of codes 128–255 are, similarly, rendered ‘`^80-^ff`’.

The boolean expression defined here should be *true* unless TeX internal code number k corresponds to a non-troublesome visible symbol in the local character set. An appropriate formula for the extended character set recommended in *The TeXbook* would, for example, be ‘ $k \in [0, 10 \dots 12, 14, 15, 33, 177 \dots 377]$ ’. If character k cannot be printed, and $k < 200$, then character $k + 100$ or $k - 100$ must be printable; moreover, ASCII codes $[41 \dots 46, 60 \dots 71, 136, 141 \dots 146, 160 \dots 171]$ must be printable. Thus, at least 80 printable characters are needed.

`(Character k cannot be printed 49) ≡
(k < "□") ∨ (k > "~")`

This code is used in section 48.

50. When the WEB system program called TANGLE processes the `TEX.WEB` description that you are now reading, it outputs the Pascal program `TEX.PAS` and also a string pool file called `TEX.POOL`. The INITEX program reads the latter file, where each string appears as a two-digit decimal length followed by the string itself, and the information is recorded in TeX’s string memory.

`(Global variables 13) +≡
init pool_file: alpha_file; { the string-pool file output by TANGLE }
tini`

51. `define bad_pool(#) ≡
begin wake_up_terminal; write_ln(term_out, #); a_close(pool_file); get_strings_started ← false;
return;
end`

`(Read the other strings from the TEX.POOL file and return true, or give an error message and return
false 51) ≡
name_of_file ← pool_name; { we needn’t set name_length }
if a_open_in(pool_file) then
begin c ← false;
repeat { Read one string, but return false if the string memory space is getting too tight for
comfort 52};
until c;
a_close(pool_file); get_strings_started ← true;
end
else bad_pool(`!IcantreadTEX.POOL.')`

This code is used in section 47.

52. ⟨ Read one string, but return *false* if the string memory space is getting too tight for comfort 52 ⟩ ≡

```

begin if eof(pool_file) then bad_pool(`!TEX.POOL.has.no.check.sum. `);
read(pool_file, m, n); { read two digits of string length }
if m = '*' then ⟨ Check the pool check sum 53 ⟩
else begin if (xord[m] < "0") ∨ (xord[m] > "9") ∨ (xord[n] < "0") ∨ (xord[n] > "9") then
    bad_pool(`!TEX.POOL.line.doesn't.begin.with.two.digits.`);
    l ← xord[m]*10 + xord[n] - "0" * 11; { compute the length }
    if pool_ptr + l + string_vacancies > pool_size then bad_pool(`!You.have.to.increase.POOLSIZE.`);
    for k ← 1 to l do
        begin if eoln(pool_file) then m ← ' ' else read(pool_file, m);
        append_char(xord[m]);
        end;
    read_ln(pool_file); g ← make_string;
    end;
end

```

This code is used in section 51.

53. The WEB operation @@ denotes the value that should be at the end of this TEX.POOL file; any other value means that the wrong pool file has been loaded.

⟨ Check the pool check sum 53 ⟩ ≡

```

begin a ← 0; k ← 1;
loop begin if (xord[n] < "0") ∨ (xord[n] > "9") then
    bad_pool(`!TEX.POOL.check.sum.doesn't.have.nine.digits.`);
    a ← 10 * a + xord[n] - "0";
    if k = 9 then goto done;
    incr(k); read(pool_file, n);
    end;
done: if a ≠ @@ then bad_pool(`!TEX.POOL.doesn't.match; TANGLE.me.again.`);
    c ← true;
end

```

This code is used in section 52.

54. On-line and off-line printing. Messages that are sent to a user's terminal and to the transcript-log file are produced by several '*print*' procedures. These procedures will direct their output to a variety of places, based on the setting of the global variable *selector*, which has the following possible values:

term_and_log, the normal setting, prints on the terminal and on the transcript file.

log_only, prints only on the transcript file.

term_only, prints only on the terminal.

no_print, doesn't print at all. This is used only in rare cases before the transcript file is open.

pseudo, puts output into a cyclic buffer that is used by the *show_context* routine; when we get to that routine we shall discuss the reasoning behind this curious mode.

new_string, appends the output to the current string in the string pool.

0 to 15, prints on one of the sixteen files for \write output.

The symbolic names '*term_and_log*', etc., have been assigned numeric codes that satisfy the convenient relations *no_print* + 1 = *term_only*, *no_print* + 2 = *log_only*, *term_only* + 2 = *log_only* + 1 = *term_and_log*.

Three additional global variables, *tally* and *term_offset* and *file_offset*, record the number of characters that have been printed since they were most recently cleared to zero. We use *tally* to record the length of (possibly very long) stretches of printing; *term_offset* and *file_offset*, on the other hand, keep track of how many characters have appeared so far on the current line that has been output to the terminal or to the transcript file, respectively.

```
define no_print = 16 { selector setting that makes data disappear }
define term_only = 17 { printing is destined for the terminal only }
define log_only = 18 { printing is destined for the transcript file only }
define term_and_log = 19 { normal selector setting }
define pseudo = 20 { special selector setting for show_context }
define new_string = 21 { printing is deflected to the string pool }
define max_selector = 21 { highest selector setting }

⟨Global variables 13⟩ +==
log_file: alpha_file; { transcript of TeX session }
selector: 0 .. max_selector; { where to print a message }
dig: array [0 .. 22] of 0 .. 15; { digits in a number being output }
tally: integer; { the number of characters recently printed }
term_offset: 0 .. max_print_line; { the number of characters on the current terminal line }
file_offset: 0 .. max_print_line; { the number of characters on the current file line }
trick_buf: array [0 .. error_line] of ASCII_code; { circular buffer for pseudoprinting }
trick_count: integer; { threshold for pseudoprinting, explained later }
first_count: integer; { another variable for pseudoprinting }
```

55. ⟨Initialize the output routines 55⟩ ≡

selector ← *term_only*; *tally* ← 0; *term_offset* ← 0; *file_offset* ← 0;

See also sections 61, 554, and 559.

This code is used in section 1512.

56. Macro abbreviations for output to the terminal and to the log file are defined here for convenience. Some systems need special conventions for terminal output, and it is possible to adhere to those conventions by changing *wterm*, *wterm_ln*, and *wterm_cr* in this section.

```
define wterm(#) ≡ write(term_out, #)
define wterm_ln(#) ≡ write_ln(term_out, #)
define wterm_cr ≡ write_ln(term_out)
define wlog(#) ≡ write(log_file, #)
define wlog_ln(#) ≡ write_ln(log_file, #)
define wlog_cr ≡ write_ln(log_file)
```

57. To end a line of text output, we call *print_ln*.

```
< Basic printing procedures 57 > ≡
procedure print_ln; { prints an end-of-line }
begin case selector of
  term_and_log: begin wterm_cr; wlog_cr; term_offset ← 0; file_offset ← 0;
  end;
  log_only: begin wlog_cr; file_offset ← 0;
  end;
  term_only: begin wterm_cr; term_offset ← 0;
  end;
  no_print, pseudo, new_string: do_nothing;
  othercases write_ln(write_file[selector])
  endcases;
end; { tally is not affected }
```

See also sections 58, 59, 60, 62, 63, 64, 65, 284, 285, 544, 875, 1602, and 1822.

This code is used in section 4.

58. The *print_char* procedure sends one character to the desired destination, using the *xchr* array to map it into an external character compatible with *input_ln*. All printing comes through *print_ln* or *print_char*.

```
< Basic printing procedures 57 > +≡
procedure print_char(s : ASCII_code); { prints a single character }
label exit;
begin if < Character s is the current new-line character 262 > then
  if selector < pseudo then
    begin print_ln; return;
    end;
  case selector of
    term_and_log: begin wterm(xchr[s]); wlog(xchr[s]); incr(term_offset); incr(file_offset);
    if term_offset = max_print_line then
      begin wterm_cr; term_offset ← 0;
      end;
    if file_offset = max_print_line then
      begin wlog_cr; file_offset ← 0;
      end;
    end;
    log_only: begin wlog(xchr[s]); incr(file_offset);
    if file_offset = max_print_line then print_ln;
    end;
    term_only: begin wterm(xchr[s]); incr(term_offset);
    if term_offset = max_print_line then print_ln;
    end;
    no_print: do_nothing;
    pseudo: if tally < trick_count then trick_buf[tally mod error_line] ← s;
    new_string: begin if pool_ptr < pool_size then append_char(s);
    end; { we drop characters if the string space is full }
    othercases write(write_file[selector], xchr[s])
    endcases;
    incr(tally);
exit: end;
```

59. An entire string is output by calling *print*. Note that if we are outputting the single standard ASCII character *c*, we could call *print("c")*, since "c" = 99 is the number of a single-character string, as explained above. But *print_char("c")* is quicker, so TeX goes directly to the *print_char* routine when it knows that this is safe. (The present implementation assumes that it is always safe to print a visible ASCII character.)

```
< Basic printing procedures 57 > +≡
procedure print(s : integer); { prints string s }
label exit;
var j: pool_pointer; { current character code position }
nl: integer; { new-line character to restore }
begin if s ≥ str_ptr then s ← "???" { this can't happen }
else if s < 256 then
  if s < 0 then s ← "???" { can't happen }
  else begin if selector > pseudo then
    begin print_char(s); return; { internal strings are not expanded }
    end;
  if ((Character s is the current new-line character 262)) then
    if selector < pseudo then
      begin print_ln; return;
    end;
  nl ← new_line_char; new_line_char ← -1; { temporarily disable new-line character }
  j ← str_start[s];
  while j < str_start[s + 1] do
    begin print_char(so(str_pool[j])); incr(j);
    end;
  new_line_char ← nl; return;
  end;
j ← str_start[s];
while j < str_start[s + 1] do
  begin print_char(so(str_pool[j])); incr(j);
  end;
exit: end;
```

60. Control sequence names, file names, and strings constructed with \string might contain *ASCII_code* values that can't be printed using *print_char*. Therefore we use *slow_print* for them:

```
< Basic printing procedures 57 > +≡
procedure slow_print(s : integer); { prints string s }
var j: pool_pointer; { current character code position }
begin if (s ≥ str_ptr) ∨ (s < 256) then print(s)
else begin j ← str_start[s];
  while j < str_start[s + 1] do
    begin print(so(str_pool[j])); incr(j);
    end;
  end;
end;
```

61. Here is the very first thing that TeX prints: a headline that identifies the version number and format package. The *term_offset* variable is temporarily incorrect, but the discrepancy is not serious since we assume that this part of the program is system dependent.

```
< Initialize the output routines 55 > +≡
  wterm(banner);
  if format_ident = 0 then wterm_ln(`no_format_preloaded')
  else begin slow_print(format_ident); print_ln;
    end;
  update_terminal;
```

62. The procedure *print_nl* is like *print*, but it makes sure that the string appears at the beginning of a new line.

```
< Basic printing procedures 57 > +≡
procedure print_nl(s : str_number); { prints string s at beginning of line }
  begin if ((term_offset > 0) ∧ (odd(selector))) ∨ ((file_offset > 0) ∧ (selector ≥ log_only)) then print_ln;
    print(s);
  end;
```

63. The procedure *print_esc* prints a string that is preceded by the user's escape character (which is usually a backslash).

```
< Basic printing procedures 57 > +≡
procedure print_esc(s : str_number); { prints escape character, then s }
  var c: integer; { the escape character code }
  begin { Set variable c to the current escape character 261 };
    if c ≥ 0 then
      if c < 256 then print(c);
      slow_print(s);
    end;
```

64. An array of digits in the range 0 .. 15 is printed by *print_the_digs*.

```
< Basic printing procedures 57 > +≡
procedure print_the_digs(k : eight_bits); { prints dig[k - 1]...dig[0]}
  begin while k > 0 do
    begin decr(k);
      if dig[k] < 10 then print_char("0" + dig[k])
      else print_char("A" - 10 + dig[k]);
    end;
  end;
```

65. The following procedure, which prints out the decimal representation of a given integer n , has been written carefully so that it works properly if $n = 0$ or if $(-n)$ would cause overflow. It does not apply **mod** or **div** to negative arguments, since such operations are not implemented consistently by all Pascal compilers.

```
< Basic printing procedures 57 > +≡
procedure print_int( $n$  : longinteger); { prints an integer in decimal form }
  var  $k$ : 0 .. 23; { index to current digit; we assume that  $|n| < 10^{23}$  }
     $m$ : longinteger; { used to negate  $n$  in possibly dangerous cases }
  begin  $k \leftarrow 0$ ;
  if  $n < 0$  then
    begin print_char("-");
    if  $n > -100000000$  then negate( $n$ )
    else begin  $m \leftarrow -1 - n$ ;  $n \leftarrow m \text{ div } 10$ ;  $m \leftarrow (m \text{ mod } 10) + 1$ ;  $k \leftarrow 1$ ;
      if  $m < 10$  then  $\text{dig}[0] \leftarrow m$ 
      else begin  $\text{dig}[0] \leftarrow 0$ ; incr( $n$ );
        end;
      end;
    end;
  repeat  $\text{dig}[k] \leftarrow n \text{ mod } 10$ ;  $n \leftarrow n \text{ div } 10$ ; incr( $k$ );
  until  $n = 0$ ;
  print_the_digs( $k$ );
end;
```

66. Here is a trivial procedure to print two digits; it is usually called with a parameter in the range $0 \leq n \leq 99$.

```
procedure print_two( $n$  : integer); { prints two least significant digits }
begin  $n \leftarrow \text{abs}(n) \text{ mod } 100$ ; print_char("0" + ( $n \text{ div } 10$ )); print_char("0" + ( $n \text{ mod } 10$ ));
end;
```

67. Hexadecimal printing of nonnegative integers is accomplished by *print_hex*.

```
procedure print_hex( $n$  : integer); { prints a positive integer in hexadecimal form }
  var  $k$ : 0 .. 22; { index to current digit; we assume that  $0 \leq n < 16^{22}$  }
  begin  $k \leftarrow 0$ ; print_char("0");
  repeat  $\text{dig}[k] \leftarrow n \text{ mod } 16$ ;  $n \leftarrow n \text{ div } 16$ ; incr( $k$ );
  until  $n = 0$ ;
  print_the_digs( $k$ );
end;
```

68. Old versions of T_EX needed a procedure called *print_ASCII* whose function is now subsumed by *print*. We retain the old name here as a possible aid to future software archaeologists.

```
define print_ASCII ≡ print
```

69. Roman numerals are produced by the *print_roman_int* routine. Readers who like puzzles might enjoy trying to figure out how this tricky code works; therefore no explanation will be given. Notice that 1990 yields `mcmxc`, not `mxxm`.

```
procedure print_roman_int(n : integer);
label exit;
var j, k: pool_pointer; { mysterious indices into str_pool }
    u, v: nonnegative_integer; { mysterious numbers }
begin j ← str_start["m2d5c215x2v5i"]; v ← 1000;
loop begin while n ≥ v do
    begin print_char(so(str_pool[j])); n ← n - v;
    end;
    if n ≤ 0 then return; { nonpositive input produces no output }
    k ← j + 2; u ← v div (so(str_pool[k - 1]) - "0");
    if str_pool[k - 1] = si("2") then
        begin k ← k + 2; u ← u div (so(str_pool[k - 1]) - "0");
        end;
    if n + u ≥ v then
        begin print_char(so(str_pool[k])); n ← n + u;
        end
    else begin j ← j + 2; v ← v div (so(str_pool[j - 1]) - "0");
        end;
    end;
exit: end;
```

70. The *print* subroutine will not print a string that is still being created. The following procedure will.

```
procedure print_current_string; { prints a yet-unmade string }
var j: pool_pointer; { points to current character code }
begin j ← str_start[str_ptr];
while j < pool_ptr do
    begin print_char(so(str_pool[j])); incr(j);
    end;
end;
```

71. Here is a procedure that asks the user to type a line of input, assuming that the *selector* setting is either *term_only* or *term_and_log*. The input is placed into locations *first* through *last - 1* of the *buffer* array, and echoed on the transcript file if appropriate.

This procedure is never called when *interaction < scroll_mode*.

```
define prompt_input(#) ≡
    begin wake_up_terminal; print(#); term_input;
    end { prints a string and gets a line of input }

procedure term_input; { gets a line from the terminal }
var k: 0 .. buf_size; { index into buffer }
begin update_terminal; { now the user sees the prompt for sure }
if ¬input_ln(term_in, true) then fatal_error("End_of_file_on_the_terminal!");
term_offset ← 0; { the user's line ended with <return> }
decr(selector); { prepare to echo the input }
if last ≠ first then
    for k ← first to last - 1 do print(buffer[k]);
print_ln; incr(selector); { restore previous status }
end;
```

72. Reporting errors. When something anomalous is detected, TeX typically does something like this:

```
print_err("Something_anomalous_has_been_detected");
help3("This_is_the_first_line_of_my_offer_to_help.");
("This_is_the_second_line.\u2022I'm_trying_to")
("explain_the_best_way_for_you_to_proceed.");
error;
```

A two-line help message would be given using `help2`, etc.; these informal helps should use simple vocabulary that complements the words used in the official error message that was printed. (Outside the U.S.A., the help messages should preferably be translated into the local vernacular. Each line of help is at most 60 characters long, in the present implementation, so that `max_print_line` will not be exceeded.)

The `print_err` procedure supplies a ‘!’ before the official message, and makes sure that the terminal is awake if a stop is going to occur. The `error` procedure supplies a ‘.’ after the official message, then it shows the location of the error; and if `interaction = error_stop_mode`, it also enters into a dialog with the user, during which time the help message may be printed.

73. The global variable `interaction` has four settings, representing increasing amounts of user interaction:

```
define batch_mode = 0 { omits all stops and omits terminal output }
define nonstop_mode = 1 { omits all stops }
define scroll_mode = 2 { omits error stops }
define error_stop_mode = 3 { stops at every opportunity to interact }
define print_err(#) ==
  begin if interaction = error_stop_mode then wake_up_terminal;
  print_nl("!\u2022"); print(#);
  end
⟨ Global variables 13 ⟩ +≡
interaction: batch_mode .. error_stop_mode; { current level of interaction }
```

74. ⟨ Set initial values of key variables 21 ⟩ +≡
`interaction ← error_stop_mode;`

75. TeX is careful not to call `error` when the print `selector` setting might be unusual. The only possible values of `selector` at the time of error messages are

```
no_print (when interaction = batch_mode and log_file not yet open);
term_only (when interaction > batch_mode and log_file not yet open);
log_only (when interaction = batch_mode and log_file is open);
term_and_log (when interaction > batch_mode and log_file is open).
```

⟨ Initialize the print `selector` based on `interaction` 75 ⟩ ≡
`if interaction = batch_mode then selector ← no_print else selector ← term_only`

This code is used in sections 1443 and 1517.

76. A global variable *deletions_allowed* is set *false* if the *get_next* routine is active when *error* is called; this ensures that *get_next* and related routines like *get_token* will never be called recursively. A similar interlock is provided by *set_box_allowed*.

The global variable *history* records the worst level of error that has been detected. It has four possible values: *spotless*, *warning_issued*, *error_message_issued*, and *fatal_error_stop*.

Another global variable, *error_count*, is increased by one when an *error* occurs without an interactive dialog, and it is reset to zero at the end of every paragraph. If *error_count* reaches 100, TeX decides that there is no point in continuing further.

```
define spotless = 0 { history value when nothing has been amiss yet }
define warning_issued = 1 { history value when begin_diagnostic has been called }
define error_message_issued = 2 { history value when error has been called }
define fatal_error_stop = 3 { history value when termination was premature }

⟨Global variables 13⟩ +≡
deletions_allowed: boolean; { is it safe for error to call get_token? }
set_box_allowed: boolean; { is it safe to do a \setbox assignment? }
history: spotless .. fatal_error_stop; { has the source input been clean so far? }
error_count: -1 .. 100; { the number of scrolled errors since the last paragraph ended }
```

77. The value of *history* is initially *fatal_error_stop*, but it will be changed to *spotless* if TeX survives the initialization process.

```
⟨ Set initial values of key variables 21 ⟩ +≡
deletions_allowed ← true; set_box_allowed ← true; error_count ← 0; { history is initialized elsewhere }
```

78. Since errors can be detected almost anywhere in TeX, we want to declare the error procedures near the beginning of the program. But the error procedures in turn use some other procedures, which need to be declared *forward* before we get to *error* itself.

It is possible for *error* to be called recursively if some error arises when *get_token* is being used to delete a token, and/or if some fatal error occurs while TeX is trying to fix a non-fatal one. But such recursion is never more than two levels deep.

```
⟨ Error handling procedures 78 ⟩ ≡
procedure normalize_selector; forward;
procedure get_token; forward;
procedure term_input; forward;
procedure show_context; forward;
procedure begin_file_reading; forward;
procedure open_log_file; forward;
procedure close_files_and_terminate; forward;
procedure clear_for_error_prompt; forward;
procedure give_err_help; forward;
debug procedure debug_help; forward; gubed
```

See also sections 81, 82, 93, 94, and 95.

This code is used in section 4.

79. Individual lines of help are recorded in the array *help_line*, which contains entries in positions 0 .. (*help_ptr* - 1). They should be printed in reverse order, i.e., with *help_line*[0] appearing last.

```
define hlp1 (#) ≡ help_line[0] ← #; end
define hlp2 (#) ≡ help_line[1] ← #; hlp1
define hlp3 (#) ≡ help_line[2] ← #; hlp2
define hlp4 (#) ≡ help_line[3] ← #; hlp3
define hlp5 (#) ≡ help_line[4] ← #; hlp4
define hlp6 (#) ≡ help_line[5] ← #; hlp5
define help0 ≡ help_ptr ← 0 { sometimes there might be no help }
define help1 ≡ begin help_ptr ← 1; hlp1 { use this with one help line }
define help2 ≡ begin help_ptr ← 2; hlp2 { use this with two help lines }
define help3 ≡ begin help_ptr ← 3; hlp3 { use this with three help lines }
define help4 ≡ begin help_ptr ← 4; hlp4 { use this with four help lines }
define help5 ≡ begin help_ptr ← 5; hlp5 { use this with five help lines }
define help6 ≡ begin help_ptr ← 6; hlp6 { use this with six help lines }

⟨ Global variables 13 ⟩ +≡
help_line: array [0 .. 5] of str_number; { helps for the next error }
help_ptr: 0 .. 6; { the number of help lines present }
use_err_help: boolean; { should the err_help list be shown? }
```

80. ⟨ Set initial values of key variables 21 ⟩ +≡
help_ptr ← 0; *use_err_help* ← false;

81. The *jump_out* procedure just cuts across all active procedure levels and goes to *end_of_TEX*. This is the only nontrivial **goto** statement in the whole program. It is used when there is no recovery from a particular error.

Some Pascal compilers do not implement non-local **goto** statements. In such cases the body of *jump_out* should simply be ‘*close_files_and_terminate*;’ followed by a call on some system procedure that quietly terminates the program.

```
⟨ Error handling procedures 78 ⟩ +≡
procedure jump_out;
begin goto end_of_TEX;
end;
```

82. Here now is the general *error* routine.

```
⟨ Error handling procedures 78 ⟩ +≡
procedure error; { completes the job of error reporting }
label continue, exit;
var c: ASCII_code; { what the user types }
s1, s2, s3, s4: integer; { used to save global variables when deleting tokens }
begin if history < error_message_issued then history ← error_message_issued;
print_char(".");
show_context;
if interaction = error_stop_mode then ⟨ Get user's advice and return 83 ⟩;
incr(error_count);
if error_count = 100 then
begin print_nl("That makes 100 errors; please try again."); history ← fatal_error_stop;
jump_out;
end;
⟨ Put help message on the transcript file 90 ⟩;
exit: end;
```

83. ⟨Get user's advice and return 83⟩ ≡

```

loop begin continue: if interaction ≠ error_stop_mode then return;
  clear_for_error_prompt; prompt_input("?\u2022");
  if last = first then return;
  c ← buffer[first];
  if c ≥ "a" then c ← c + "A" – "a"; { convert to uppercase }
  ⟨Interpret code c and return if done 84⟩;
end

```

This code is used in section 82.

84. It is desirable to provide an 'E' option here that gives the user an easy way to return from TeX to the system editor, with the offending line ready to be edited. But such an extension requires some system wizardry, so the present implementation simply types out the name of the file that should be edited and the relevant line number.

There is a secret 'D' option available when the debugging routines haven't been commented out.

⟨ Interpret code c and return if done 84 ⟩ ≡

```

case c of
  "0", "1", "2", "3", "4", "5", "6", "7", "8", "9": if deletions_allowed then
    ⟨ Delete c – "0" tokens and goto continue 88 ⟩;
  debug "D": begin debug_help; goto continue; end; gubed
  "E": if base_ptr > 0 then
    if input_stack[base_ptr].name_field ≥ 256 then
      begin print_nl("You\u2022want\u2022to\u2022edit\u2022file\u2022"); slow_print(input_stack[base_ptr].name_field);
        print("\u2022at\u2022line\u2022"); print_int(line); interaction ← scroll_mode; jump_out;
      end;
    "H": ⟨ Print the help information and goto continue 89 ⟩;
    "I": ⟨ Introduce new material from the terminal and return 87 ⟩;
    "Q", "R", "S": ⟨ Change the interaction level and return 86 ⟩;
    "X": begin interaction ← scroll_mode; jump_out;
    end;
  othercases do_nothing
endcases;
⟨ Print the menu of available options 85 ⟩

```

This code is used in section 83.

85. ⟨Print the menu of available options 85⟩ ≡

```

begin print("Type\u2022<return>\u2022to\u2022proceed,\u2022S\u2022to\u2022scroll\u2022future\u2022error\u2022messages,");
print_nl("R\u2022to\u2022run\u2022without\u2022stopping,\u2022Q\u2022to\u2022run\u2022quietly,\u2022");
print_nl("I\u2022to\u2022insert\u2022something,\u2022");
if base_ptr > 0 then
  if input_stack[base_ptr].name_field ≥ 256 then print("E\u2022to\u2022edit\u2022your\u2022file,\u2022");
if deletions_allowed then
  print_nl("1\u2022or\u2022...\u2022or\u20229\u2022to\u2022ignore\u2022the\u2022next\u20221\u2022to\u20229\u2022tokens\u2022of\u2022input,\u2022");
  print_nl("H\u2022for\u2022help,\u2022X\u2022to\u2022quit.\u2022");
end

```

This code is used in section 84.

86. Here the author of TeX apologizes for making use of the numerical relation between "Q", "R", "S", and the desired interaction settings *batch_mode*, *nonstop_mode*, *scroll_mode*.

(Change the interaction level and return 86) ≡

```
begin error_count ← 0; interaction ← batch_mode + c - "Q"; print("OK, ↴entering↵");
case c of
  "Q": begin print_esc("batchmode"); decr(selector);
  end;
  "R": print_esc("nonstopmode");
  "S": print_esc("scrollmode");
end; { there are no other cases }
print("..."); print_ln; update_terminal; return;
end
```

This code is used in section 84.

87. When the following code is executed, *buffer*[(*first* + 1) .. (*last* - 1)] may contain the material inserted by the user; otherwise another prompt will be given. In order to understand this part of the program fully, you need to be familiar with TeX's input stacks.

(Introduce new material from the terminal and return 87) ≡

```
begin begin_file_reading; { enter a new syntactic level for terminal input }
  { now state = mid_line, so an initial blank space will count as a blank }
if last > first + 1 then
  begin loc ← first + 1; buffer[first] ← " ";
  end
else begin prompt_input("insert>"); loc ← first;
  end;
first ← last; cur_input.limit_field ← last - 1; { no end_line_char ends this line }
return;
end
```

This code is used in section 84.

88. We allow deletion of up to 99 tokens at a time.

(Delete c - "0" tokens and goto continue 88) ≡

```
begin s1 ← cur_tok; s2 ← cur_cmd; s3 ← cur_chr; s4 ← align_state; align_state ← 1000000;
OK_to_interrupt ← false;
if (last > first + 1) ∧ (buffer[first + 1] ≥ "0") ∧ (buffer[first + 1] ≤ "9") then
  c ← c * 10 + buffer[first + 1] - "0" * 11
else c ← c - "0";
while c > 0 do
  begin get_token; { one-level recursive call of error is possible }
  decr(c);
  end;
cur_tok ← s1; cur_cmd ← s2; cur_chr ← s3; align_state ← s4; OK_to_interrupt ← true;
help2("I ↴have ↴just ↴deleted ↴some ↴text, ↴as ↴you ↴asked. ")
("You ↴can ↴now ↴delete ↴more, ↴or ↴insert, ↴or ↴whatever. ");
show_context; goto continue;
end
```

This code is used in section 84.

```

89. < Print the help information and goto continue 89 > ≡
begin if use_err_help then
  begin give_err_help; use_err_help ← false;
  end
else begin if help_ptr = 0 then help2("Sorry, I don't know how to help in this situation.")
  ("Maybe you should try asking a human?");
  repeat decr(help_ptr); print(help_line[help_ptr]); print_ln;
  until help_ptr = 0;
end;
help4("Sorry, I already gave what help I could...")
("Maybe you should try asking a human?")
("An error might have occurred before I noticed any problems.")
(``If all else fails, read the instructions.'');
goto continue;
end

```

This code is used in section 84.

```

90. < Put help message on the transcript file 90 > ≡
if interaction > batch_mode then decr(selector); { avoid terminal output }
if use_err_help then
  begin print_ln; give_err_help;
  end
else while help_ptr > 0 do
  begin decr(help_ptr); print_nl(help_line[help_ptr]);
  end;
print_ln;
if interaction > batch_mode then incr(selector); { re-enable terminal output }
print_ln

```

This code is used in section 82.

91. A dozen or so error messages end with a parenthesized integer, so we save a teeny bit of program space by declaring the following procedure:

```

procedure int_error(n : integer);
begin print("("); print_int(n); print_char(")"); error;
end;

```

92. In anomalous cases, the print selector might be in an unknown state; the following subroutine is called to fix things just enough to keep running a bit longer.

```

procedure normalize_selector;
begin if log_opened then selector ← term_and_log
  else selector ← term_only;
  if job_name = 0 then open_log_file;
  if interaction = batch_mode then decr(selector);
end;

```

93. The following procedure prints T_EX's last words before dying.

```
define succumb ==
  begin if interaction = error_stop_mode then interaction ← scroll_mode;
    { no more interaction }
  if log_opened then error;
  debug if interaction > batch_mode then debug_help;
  gubed
  history ← fatal_error_stop; jump_out; { irrecoverable error }
  end

⟨ Error handling procedures 78 ⟩ +≡
procedure fatal_error(s : str_number); { prints s, and that's it }
  begin normalize_selector;
  print_err("Emergency_stop"); help1(s); succumb;
  end;
```

94. Here is the most dreaded error message.

```
⟨ Error handling procedures 78 ⟩ +≡
procedure overflow(s : str_number; n : integer); { stop due to finiteness }
  begin normalize_selector; print_err("TeX_capacity_exceeded, sorry["); print(s); print_char("=");
  print_int(n); print_char("]"); help2("If you really absolutely need more capacity, "
  ("you can ask a wizard to enlarge me.")); succumb;
  end;
```

95. The program might sometime run completely amok, at which point there is no choice but to stop. If no previous error has been detected, that's bad news; a message is printed that is really intended for the T_EX maintenance person instead of the user (unless the user has been particularly diabolical). The index entries for 'this can't happen' may help to pinpoint the problem.

```
⟨ Error handling procedures 78 ⟩ +≡
procedure confusion(s : str_number); { consistency check violated; s tells where }
  begin normalize_selector;
  if history < error_message_issued then
    begin print_err("This can't happen!"); print(s); print_char(")");
    help1("I'm broken. Please show this to someone who can fix it");
    end
  else begin print_err("I can't go on meeting you like this");
    help2("One of your faux pas seems to have wounded me deeply...")
    ("in fact, I'm barely conscious. Please fix it and try again.");
    end;
  succumb;
  end;
```

96. Users occasionally want to interrupt T_EX while it's running. If the Pascal runtime system allows this, one can implement a routine that sets the global variable *interrupt* to some nonzero value when such an interrupt is signalled. Otherwise there is probably at least a way to make *interrupt* nonzero using the Pascal debugger.

```
define check_interrupt ==
  begin if interrupt ≠ 0 then pause_for_instructions;
  end

⟨ Global variables 13 ⟩ +≡
interrupt: integer; { should TEX pause for instructions? }
OK_to_interrupt: boolean; { should interrupts be observed? }
```

97. ⟨ Set initial values of key variables 21 ⟩ +≡
interrupt ← 0; *OK_to_interrupt* ← true;

98. When an interrupt has been detected, the program goes into its highest interaction level and lets the user have nearly the full flexibility of the *error* routine. TeX checks for interrupts only at times when it is safe to do this.

```
procedure pause_for_instructions;
begin if OK_to_interrupt then
  begin interaction ← error_stop_mode;
  if (selector = log_only) ∨ (selector = no_print) then incr(selector);
  print_err("Interruption"); help3("You rang?");
  ("Try to insert an instruction for me (e.g., `I\\showlists') ,")
  ("unless you just want to quit by typing `X'."); deletions_allowed ← false; error;
  deletions_allowed ← true; interrupt ← 0;
  end;
end;
```

99. Arithmetic with scaled dimensions. The principal computations performed by T_EX are done entirely in terms of integers less than 2^{31} in magnitude; and divisions are done only when both dividend and divisor are nonnegative. Thus, the arithmetic specified in this program can be carried out in exactly the same way on a wide variety of computers, including some small ones. Why? Because the arithmetic calculations need to be spelled out precisely in order to guarantee that T_EX will produce identical output on different machines. If some quantities were rounded differently in different implementations, we would find that line breaks and even page breaks might occur in different places. Hence the arithmetic of T_EX has been designed with care, and systems that claim to be implementations of T_EX82 should follow precisely the calculations as they appear in the present program.

(Actually there are three places where T_EX uses `div` with a possibly negative numerator. These are harmless; see `div` in the index. Also if the user sets the `\time` or the `\year` to a negative value, some diagnostic information will involve negative-numerator division. The same remarks apply for `mod` as well as for `div`.)

100. Here is a routine that calculates half of an integer, using an unambiguous convention with respect to signed odd numbers.

```
function half(x : integer): integer;
begin if odd(x) then half ← (x + 1) div 2
else half ← x div 2;
end;
```

101. Fixed-point arithmetic is done on *scaled integers* that are multiples of 2^{-16} . In other words, a binary point is assumed to be sixteen bit positions from the right end of a binary computer word.

```
define unity ≡ '200000 {  $2^{16}$ , represents 1.00000 }
define two ≡ '400000 {  $2^{17}$ , represents 2.00000 }

⟨ Types in the outer block 18 ⟩ +≡
scaled = integer; { this type is used for scaled integers }
nonnegative_integer = 0 .. '177777777777; {  $0 \leq x < 2^{31}$  }
small_number = 0 .. 63; { this type is self-explanatory }
```

102. The following function is used to create a scaled integer from a given decimal fraction $(.d_0 d_1 \dots d_{k-1})$, where $0 \leq k \leq 17$. The digit d_i is given in `dig[i]`, and the calculation produces a correctly rounded result.

```
function round_decimals(k : small_number): scaled; { converts a decimal fraction }
var a: integer; { the accumulator }
begin a ← 0;
while k > 0 do
begin decr(k); a ← (a + dig[k] * two) div 10;
end;
round_decimals ← (a + 1) div 2;
end;
```

103. Conversely, here is a procedure analogous to *print_int*. If the output of this procedure is subsequently read by TeX and converted by the *round_decimals* routine above, it turns out that the original value will be reproduced exactly; the “simplest” such decimal number is output, but there is always at least one digit following the decimal point.

The invariant relation in the **repeat** loop is that a sequence of decimal digits yet to be printed will yield the original number if and only if they form a fraction f in the range $s - \delta \leq 10 \cdot 2^{16}f < s$. We can stop if and only if $f = 0$ satisfies this condition; the loop will terminate before s can possibly become zero.

```
procedure print_scaled(s : scaled); { prints scaled real, rounded to five digits }
  var delta: scaled; { amount of allowable inaccuracy }
  begin if s < 0 then
    begin print_char("-"); negate(s); { print the sign, if negative }
    end;
    print_int(s div unity); { print the integer part }
    print_char("."); s ← 10 * (s mod unity) + 5; delta ← 10;
    repeat if delta > unity then s ← s + '100000 - 50000; { round the last digit }
      print_char("0" + (s div unity)); s ← 10 * (s mod unity); delta ← delta * 10;
    until s ≤ delta;
  end;
```

104. Physical sizes that a TeX user specifies for portions of documents are represented internally as scaled points. Thus, if we define an ‘sp’ (scaled point) as a unit equal to 2^{-16} printer’s points, every dimension inside of TeX is an integer number of sp. There are exactly 4,736,286.72 sp per inch. Users are not allowed to specify dimensions larger than $2^{30} - 1$ sp, which is a distance of about 18.892 feet (5.7583 meters); two such quantities can be added without overflow on a 32-bit computer.

The present implementation of TeX does not check for overflow when dimensions are added or subtracted. This could be done by inserting a few dozen tests of the form ‘**if** $x \geq 10000000000$ **then** *report_overflow*’, but the chance of overflow is so remote that such tests do not seem worthwhile.

TeX needs to do only a few arithmetic operations on scaled quantities, other than addition and subtraction, and the following subroutines do most of the work. A single computation might use several subroutine calls, and it is desirable to avoid producing multiple error messages in case of arithmetic overflow; so the routines set the global variable *arith_error* to *true* instead of reporting errors directly to the user. Another global variable, *remainder*, holds the remainder after a division.

```
{ Global variables 13 } +≡
arith_error: boolean; { has arithmetic overflow occurred recently? }
remainder: scaled; { amount subtracted to get an exact division }
```

105. The first arithmetical subroutine we need computes $nx + y$, where x and y are *scaled* and n is an integer. We will also use it to multiply integers.

```
define nx_plus_y(#) ≡ mult_and_add(#, '777777777777)
define mult_integers(#) ≡ mult_and_add(#, 0, '177777777777)

function mult_and_add(n : integer; x, y, max_answer : scaled): scaled;
  begin if n < 0 then
    begin negate(x); negate(n);
    end;
  if n = 0 then mult_and_add ← y
  else if ((x ≤ (max_answer - y) div n) ∧ (-x ≤ (max_answer + y) div n)) then mult_and_add ← n * x + y
    else begin arith_error ← true; mult_and_add ← 0;
    end;
  end;
```

106. We also need to divide scaled dimensions by integers.

```
function x_over_n(x : scaled; n : integer): scaled;
  var negative: boolean; { should remainder be negated? }
  begin negative ← false;
  if n = 0 then
    begin arith_error ← true; x_over_n ← 0; remainder ← x;
    end
  else begin if n < 0 then
    begin negate(x); negate(n); negative ← true;
    end;
    if x ≥ 0 then
      begin x_over_n ← x div n; remainder ← x mod n;
      end
    else begin x_over_n ← -((-x) div n); remainder ← -((-x) mod n);
    end;
  end;
  if negative then negate(remainder);
end;
```

107. Then comes the multiplication of a scaled number by a fraction n/d , where n and d are nonnegative integers $\leq 2^{16}$ and d is positive. It would be too dangerous to multiply by n and then divide by d , in separate operations, since overflow might well occur; and it would be too inaccurate to divide by d and then multiply by n . Hence this subroutine simulates 1.5-precision arithmetic.

```
function xn_over_d(x : scaled; n, d : integer): scaled;
  var positive: boolean; { was x ≥ 0? }
  t, u, v: nonnegative_integer; { intermediate quantities }
  begin if x ≥ 0 then positive ← true
  else begin negate(x); positive ← false;
  end;
  t ← (x mod '100000) * n; u ← (x div '100000) * n + (t div '100000);
  v ← (u mod d) * '100000 + (t mod '100000);
  if u div d ≥ '100000 then arith_error ← true
  else u ← '100000 * (u div d) + (v div d);
  if positive then
    begin xn_over_d ← u; remainder ← v mod d;
    end
  else begin xn_over_d ← -u; remainder ← -(v mod d);
  end;
end;
```

108. The next subroutine is used to compute the “badness” of glue, when a total t is supposed to be made from amounts that sum to s . According to *The TeXbook*, the badness of this situation is $100(t/s)^3$; however, badness is simply a heuristic, so we need not squeeze out the last drop of accuracy when computing it. All we really want is an approximation that has similar properties.

The actual method used to compute the badness is easier to read from the program than to describe in words. It produces an integer value that is a reasonably close approximation to $100(t/s)^3$, and all implementations of TeX should use precisely this method. Any badness of 2^{13} or more is treated as infinitely bad, and represented by 10000.

It is not difficult to prove that

$$\text{badness}(t + 1, s) \geq \text{badness}(t, s) \geq \text{badness}(t, s + 1).$$

The badness function defined here is capable of computing at most 1095 distinct values, but that is plenty.

```
define inf_bad = 10000 {infinitely bad value}
function badness(t, s : scaled): halfword; {compute badness, given t ≥ 0}
  var r: integer; {approximation to at/s, where α³ ≈ 100 · 2¹⁸}
  begin if t = 0 then badness ← 0
  else if s ≤ 0 then badness ← inf_bad
  else begin if t ≤ 7230584 then r ← (t * 297) div s {297³ = 99.94 × 2¹⁸}
    else if s ≥ 1663497 then r ← t div (s div 297)
      else r ← t;
    if r > 1290 then badness ← inf_bad {1290³ < 2³¹ < 1291³}
    else badness ← (r * r * r + '400000) div '1000000;
  end; {that was r³/2¹⁸, rounded to the nearest integer}
end;
```

109. When TeX “packages” a list into a box, it needs to calculate the proportionality ratio by which the glue inside the box should stretch or shrink. This calculation does not affect TeX’s decision making, so the precise details of rounding, etc., in the glue calculation are not of critical importance for the consistency of results on different computers.

We shall use the type *glue_ratio* for such proportionality ratios. A glue ratio should take the same amount of memory as an *integer* (usually 32 bits) if it is to blend smoothly with TeX’s other data structures. Thus *glue_ratio* should be equivalent to *short_real* in some implementations of Pascal. Alternatively, it is possible to deal with glue ratios using nothing but fixed-point arithmetic; see *TUGboat* 3,1 (March 1982), 10–27. (But the routines cited there must be modified to allow negative glue ratios.)

```
define set_glue_ratio_zero(#) ≡ # ← 0.0 {store the representation of zero ratio}
define set_glue_ratio_one(#) ≡ # ← 1.0 {store the representation of unit ratio}
define float(#) ≡ # {convert from glue_ratio to type real}
define unfloat(#) ≡ # {convert from real to type glue_ratio}
define float_constant(#) ≡ #.0 {convert integer constant to real}

⟨ Types in the outer block 18 ⟩ +≡
  glue_ratio = real; {one-word representation of a glue expansion factor}
```

110. Random numbers. This section is (almost) straight from METAPOST. I had to change the types (use *integer* instead of *fraction*), but that should not have any influence on the actual calculations (the original comments refer to quantities like *fraction_four* (2^{30}), and that is the same as the numeric representation of *maxdimen*).

I've copied the low-level variables and routines that are needed, but only those (e.g. *m_log*), not the accompanying ones like *m_exp*. Most of the following low-level numeric routines are only needed within the calculation of *norm_rand*. I've been forced to rename *make_fraction* to *make_frac* because TeX already has a routine by that name with a wholly different function (it creates a *fraction_noad* for math typesetting) – Taco.

And now let's complete our collection of numeric utility routines by considering random number generation. METAPOST generates pseudo-random numbers with the additive scheme recommended in Section 3.6 of *The Art of Computer Programming*; however, the results are random fractions between 0 and *fraction_one* – 1, inclusive.

There's an auxiliary array *randoms* that contains 55 pseudo-random fractions. Using the recurrence $x_n = (x_{n-55} - x_{n-31}) \bmod 2^{28}$, we generate batches of 55 new x_n 's at a time by calling *new_randoms*. The global variable *j_random* tells which element has most recently been consumed.

```
(Global variables 13) +≡
randoms: array [0 .. 54] of integer; { the last 55 random values generated }
j_random: 0 .. 54; { the number of unused randoms }
random_seed: scaled; { the default random seed }
```

111. A small bit of METAFONT is needed.

```
define fraction_half ≡ '1000000000 {  $2^{27}$ , represents 0.50000000 }
define fraction_one ≡ '2000000000 {  $2^{28}$ , represents 1.00000000 }
define fraction_four ≡ '1000000000 {  $2^{30}$ , represents 4.00000000 }
define el_gordo ≡ '1777777777 {  $2^{31}$  – 1, the largest value that METAPOST likes }
define halfp(#) ≡ (#) div 2
define double(#) ≡ # ← # + # { multiply a variable by two }
```

112. The *make_fra*c routine produces the *fraction* equivalent of p/q , given integers p and q ; it computes the integer $f = \lfloor 2^{28}p/q + \frac{1}{2} \rfloor$, when p and q are positive. If p and q are both of the same scaled type t , the “type relation” $\text{make_frac}(t, t) = \text{fraction}$ is valid; and it’s also possible to use the subroutine “backwards,” using the relation $\text{make_frac}(t, \text{fraction}) = t$ between scaled types.

If the result would have magnitude 2^{31} or more, *make_fra*c sets *arith_error* \leftarrow *true*. Most of METAPOST’s internal computations have been designed to avoid this sort of error.

If this subroutine were programmed in assembly language on a typical machine, we could simply compute $(2^{28} * p) \text{ div } q$, since a double-precision product can often be input to a fixed-point division instruction. But when we are restricted to Pascal arithmetic it is necessary either to resort to multiple-precision maneuvering or to use a simple but slow iteration. The multiple-precision technique would be about three times faster than the code adopted here, but it would be comparatively long and tricky, involving about sixteen additional multiplications and divisions.

This operation is part of METAPOST’s “inner loop”; indeed, it will consume nearly 10% of the running time (exclusive of input and output) if the code below is left unchanged. A machine-dependent recoding will therefore make METAPOST run faster. The present implementation is highly portable, but slow; it avoids multiplication and division except in the initial stage. System wizards should be careful to replace it with a routine that is guaranteed to produce identical results in all cases.

As noted below, a few more routines should also be replaced by machine-dependent code, for efficiency. But when a procedure is not part of the “inner loop,” such changes aren’t advisable; simplicity and robustness are preferable to trickery, unless the cost is too high.

```

function make_frac(p, q : integer): integer;
  var f: integer; { the fraction bits, with a leading 1 bit }
  n: integer; { the integer part of  $|p/q|$  }
  negative: boolean; { should the result be negated? }
  be_careful: integer; { disables certain compiler optimizations }
begin if p  $\geq$  0 then negative  $\leftarrow$  false
  else begin negate(p); negative  $\leftarrow$  true;
  end;
  if q  $\leq$  0 then
    begin debug if q = 0 then confusion("/");
    negate(q); negative  $\leftarrow$   $\neg$ negative;
    end;
  n  $\leftarrow$  p div q; p  $\leftarrow$  p mod q;
  if n  $\geq$  8 then
    begin arith_error  $\leftarrow$  true;
    if negative then make_frac  $\leftarrow$  -el_gordo else make_frac  $\leftarrow$  el_gordo;
    end
  else begin n  $\leftarrow$  (n - 1) * fraction_one; { Compute  $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$  113 };
    if negative then make_frac  $\leftarrow$   $-(f + n)$  else make_frac  $\leftarrow$  f + n;
    end;
  end;
end;

```

113. The **repeat** loop here preserves the following invariant relations between f , p , and q : (i) $0 \leq p < q$; (ii) $fq + p = 2^k(q + p_0)$, where k is an integer and p_0 is the original value of p .

Notice that the computation specifies $(p - q) + p$ instead of $(p + p) - q$, because the latter could overflow. Let us hope that optimizing compilers do not miss this point; a special variable *be_careful* is used to emphasize the necessary order of computation. Optimizing compilers should keep *be_careful* in a register, not store it in memory.

```
<Compute  $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$  113> ≡
  f ← 1;
  repeat be_careful ← p - q; p ← be_careful + p;
    if  $p \geq 0$  then  $f \leftarrow f + f + 1$ 
    else begin double(f); p ← p + q;
      end;
    until  $f \geq fraction\_one$ ;
    be_careful ← p - q;
    if be_careful + p ≥ 0 then incr(f)
```

This code is used in section 112.

114.

```
function take_frac(q : integer; f : integer): integer;
  var p: integer; { the fraction so far }
  negative: boolean; { should the result be negated? }
  n: integer; { additional multiple of q }
  be_careful: integer; { disables certain compiler optimizations }
begin <Reduce to the case that  $f \geq 0$  and  $q > 0$  115>;
  if  $f < fraction\_one$  then  $n \leftarrow 0$ 
  else begin  $n \leftarrow f \text{ div } fraction\_one$ ;  $f \leftarrow f \text{ mod } fraction\_one$ ;
    if  $q \leq el\_gordo \text{ div } n$  then  $n \leftarrow n * q$ 
    else begin arith_error ← true;  $n \leftarrow el\_gordo$ ;
      end;
    end;
   $f \leftarrow f + fraction\_one$ ; <Compute  $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$  116>;
  be_careful ← n - el_gordo;
  if be_careful + p > 0 then
    begin arith_error ← true;  $n \leftarrow el\_gordo - p$ ;
    end;
  if negative then take_frac ← -(n + p)
  else take_frac ← n + p;
  end;
```

115. <Reduce to the case that $f \geq 0$ and $q > 0$ 115> ≡

```
if  $f \geq 0$  then negative ← false
else begin negate(f); negative ← true;
  end;
if  $q < 0$  then
  begin negate(q); negative ←  $\neg$ negative;
  end;
```

This code is used in section 114.

116. The invariant relations in this case are (i) $\lfloor (qf + p)/2^k \rfloor = \lfloor qf_0/2^{28} + \frac{1}{2} \rfloor$, where k is an integer and f_0 is the original value of f ; (ii) $2^k \leq f < 2^{k+1}$.

```
< Compute  $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$  116 > ==
   $p \leftarrow \text{fraction\_half};$  { that's  $2^{27}$ ; the invariants hold now with  $k = 28$  }
  if  $q < \text{fraction\_four}$  then
    repeat if odd( $f$ ) then  $p \leftarrow \text{halfp}(p + q)$  else  $p \leftarrow \text{halfp}(p);$ 
     $f \leftarrow \text{halfp}(f);$ 
    until  $f = 1$ 
  else repeat if odd( $f$ ) then  $p \leftarrow p + \text{halfp}(q - p)$  else  $p \leftarrow \text{halfp}(p);$ 
     $f \leftarrow \text{halfp}(f);$ 
    until  $f = 1$ 
```

This code is used in section 114.

117. The subroutines for logarithm and exponential involve two tables. The first is simple: $\text{two_to_the}[k]$ equals 2^k . The second involves a bit more calculation, which the author claims to have done correctly: $\text{spec_log}[k]$ is 2^{27} times $\ln(1/(1 - 2^{-k})) = 2^{-k} + \frac{1}{2}2^{-2k} + \frac{1}{3}2^{-3k} + \dots$, rounded to the nearest integer.

```
< Global variables 13 > +≡
two_to_the: array [0 .. 30] of integer; { powers of two }
spec_log: array [1 .. 28] of integer; { special logarithms }
```

118. < Set initial values of key variables 21 > +≡

```
two_to_the[0] ← 1;
for  $k \leftarrow 1$  to 30 do  $\text{two\_to\_the}[k] \leftarrow 2 * \text{two\_to\_the}[k - 1];$ 
 $\text{spec\_log}[1] \leftarrow 93032640;$   $\text{spec\_log}[2] \leftarrow 38612034;$   $\text{spec\_log}[3] \leftarrow 17922280;$   $\text{spec\_log}[4] \leftarrow 8662214;$ 
 $\text{spec\_log}[5] \leftarrow 4261238;$   $\text{spec\_log}[6] \leftarrow 2113709;$   $\text{spec\_log}[7] \leftarrow 1052693;$   $\text{spec\_log}[8] \leftarrow 525315;$ 
 $\text{spec\_log}[9] \leftarrow 262400;$   $\text{spec\_log}[10] \leftarrow 131136;$   $\text{spec\_log}[11] \leftarrow 65552;$   $\text{spec\_log}[12] \leftarrow 32772;$ 
 $\text{spec\_log}[13] \leftarrow 16385;$ 
for  $k \leftarrow 14$  to 27 do  $\text{spec\_log}[k] \leftarrow \text{two\_to\_the}[27 - k];$ 
 $\text{spec\_log}[28] \leftarrow 1;$ 
```

119.

```
function  $m\_log(x : \text{integer})$ :  $\text{integer};$ 
  var  $y, z : \text{integer};$  { auxiliary registers }
   $k : \text{integer};$  { iteration counter }
begin if  $x \leq 0$  then < Handle non-positive logarithm 121 >
else begin  $y \leftarrow 1302456956 + 4 - 100;$  {  $14 \times 2^{27} \ln 2 \approx 1302456956.421063$  }
 $z \leftarrow 27595 + 6553600;$  { and  $2^{16} \times .421063 \approx 27595$  }
while  $x < \text{fraction\_four}$  do
  begin double( $x$ );  $y \leftarrow y - 93032639;$   $z \leftarrow z - 48782;$ 
  end; {  $2^{27} \ln 2 \approx 93032639.74436163$  and  $2^{16} \times .74436163 \approx 48782$  }
 $y \leftarrow y + (z \text{ div } \text{unity});$   $k \leftarrow 2;$ 
while  $x > \text{fraction\_four} + 4$  do
  { Increase  $k$  until  $x$  can be multiplied by a factor of  $2^{-k}$ , and adjust  $y$  accordingly 120 };
 $m\_log \leftarrow y \text{ div } 8;$ 
end;
end;
```

120. *(Increase k until x can be multiplied by a factor of 2^{-k} , and adjust y accordingly 120)* \equiv

```

begin  $z \leftarrow ((x - 1) \text{ div } \text{two\_to\_the}[k]) + 1;$  {  $z = \lceil x/2^k \rceil$  }
while  $x < \text{fraction\_four} + z$  do
  begin  $z \leftarrow \text{halfp}(z + 1); k \leftarrow k + 1;$ 
  end;
   $y \leftarrow y + \text{spec\_log}[k]$ ;  $x \leftarrow x - z;$ 
end

```

This code is used in section 119.

121. *(Handle non-positive logarithm 121)* \equiv

```

begin  $\text{print\_err}(\text{"Logarithm\_of\_u"}); \text{print\_scaled}(x); \text{print}(\text{"\_has\_been\_replaced\_by\_0"});$ 
 $\text{help2}(\text{"Since I don't take logs of non-positive numbers,"})$ 
 $(\text{"I'm zeroing this one. Proceed, with fingers crossed."}); \text{error}; m\_log \leftarrow 0;$ 
end

```

This code is used in section 119.

122. The following somewhat different subroutine tests rigorously if ab is greater than, equal to, or less than cd , given integers (a, b, c, d) . In most cases a quick decision is reached. The result is $+1$, 0 , or -1 in the three respective cases.

```

define  $\text{return\_sign}(\#) \equiv$ 
  begin  $ab\_vs\_cd \leftarrow \#;$  return;
  end

function  $ab\_vs\_cd(a, b, c, d : \text{integer}): \text{integer};$ 
label exit;
var  $q, r: \text{integer};$  { temporary registers }
begin { Reduce to the case that  $a, c \geq 0, b, d > 0$  123 };
loop begin  $q \leftarrow a \text{ div } d;$   $r \leftarrow c \text{ div } b;$ 
  if  $q \neq r$  then
    if  $q > r$  then  $\text{return\_sign}(1)$  else  $\text{return\_sign}(-1);$ 
     $q \leftarrow a \text{ mod } d;$   $r \leftarrow c \text{ mod } b;$ 
  if  $r = 0$  then
    if  $q = 0$  then  $\text{return\_sign}(0)$  else  $\text{return\_sign}(1);$ 
    if  $q = 0$  then  $\text{return\_sign}(-1);$ 
     $a \leftarrow b;$   $b \leftarrow q;$   $c \leftarrow d;$   $d \leftarrow r;$ 
  end; { now  $a > d > 0$  and  $c > b > 0$  }
exit: end;

```

123. \langle Reduce to the case that $a, c \geq 0, b, d > 0$ [123](#) $\rangle \equiv$

```

if  $a < 0$  then
  begin negate( $a$ ); negate( $b$ );
  end;
if  $c < 0$  then
  begin negate( $c$ ); negate( $d$ );
  end;
if  $d \leq 0$  then
  begin if  $b \geq 0$  then
    if  $((a = 0) \vee (b = 0)) \wedge ((c = 0) \vee (d = 0))$  then return-sign(0)
    else return-sign(1);
  if  $d = 0$  then
    if  $a = 0$  then return-sign(0) else return-sign(-1);
     $q \leftarrow a$ ;  $a \leftarrow c$ ;  $c \leftarrow q$ ;  $q \leftarrow -b$ ;  $b \leftarrow -d$ ;  $d \leftarrow q$ ;
  end
else if  $b \leq 0$  then
  begin if  $b < 0$  then
    if  $a > 0$  then return-sign(-1);
    if  $c = 0$  then return-sign(0)
    else return-sign(-1);
  end

```

This code is used in section [122](#).

124. To consume a random integer, the program below will say '*next_random*' and then it will fetch *randoms*[*j_random*].

```

define next_random  $\equiv$ 
  if j_random = 0 then new_randoms
  else decr(j_random)

procedure new_randoms;
  var k: 0 .. 54; { index into randoms }
  x: integer; { accumulator }
  begin for k  $\leftarrow$  0 to 23 do
    begin x  $\leftarrow$  randoms[k]  $-$  randoms[k + 31];
    if x < 0 then x  $\leftarrow$  x + fraction-one;
    randoms[k]  $\leftarrow$  x;
    end;
  for k  $\leftarrow$  24 to 54 do
    begin x  $\leftarrow$  randoms[k]  $-$  randoms[k - 24];
    if x < 0 then x  $\leftarrow$  x + fraction-one;
    randoms[k]  $\leftarrow$  x;
    end;
  j_random  $\leftarrow$  54;
end;

```

125. To initialize the *randoms* table, we call the following routine.

```

procedure init_randoms(seed : integer);
  var j, jj, k: integer; { more or less random integers }
    i: 0 .. 54; { index into randoms }
  begin j  $\leftarrow$  abs(seed);
  while j  $\geq$  fraction_one do j  $\leftarrow$  halfp(j);
  k  $\leftarrow$  1;
  for i  $\leftarrow$  0 to 54 do
    begin jj  $\leftarrow$  k; k  $\leftarrow$  j - k; j  $\leftarrow$  jj;
    if k < 0 then k  $\leftarrow$  k + fraction_one;
    randoms[(i * 21) mod 55]  $\leftarrow$  j;
    end;
  new_randoms; new_randoms; new_randoms; { "warm up" the array }
  end;

```

126. To produce a uniform random number in the range $0 \leq u < x$ or $0 \geq u > x$ or $0 = u = x$, given a scaled value *x*, we proceed as shown here.

Note that the call of *take_frac* will produce the values 0 and *x* with about half the probability that it will produce any other particular values between 0 and *x*, because it rounds its answers.

```

function unif_rand(x : integer): integer;
  var y: integer; { trial value }
  begin next_random; y  $\leftarrow$  take_frac(abs(x), randoms[j_random]);
  if y = abs(x) then unif_rand  $\leftarrow$  0
  else if x > 0 then unif_rand  $\leftarrow$  y
  else unif_rand  $\leftarrow$  -y;
  end;

```

127. Finally, a normal deviate with mean zero and unit standard deviation can readily be obtained with the ratio method (Algorithm 3.4.1R in *The Art of Computer Programming*).

```

function norm_rand: integer;
  var x, u, l: integer; { what the book would call  $2^{16}X$ ,  $2^{28}U$ , and  $-2^{24}\ln U$  }
  begin repeat repeat next_random; x  $\leftarrow$  take_frac(112429, randoms[j_random] - fraction_half);
    {  $2^{16}\sqrt{8/e} \approx 112428.82793$  }
    next_random; u  $\leftarrow$  randoms[j_random];
  until abs(x) < u;
  x  $\leftarrow$  make_frac(x, u); l  $\leftarrow$  139548960 - m_log(u); {  $2^{24} \cdot 12 \ln 2 \approx 139548959.6165$  }
  until ab_vs_cd(1024, l, x, x)  $\geq$  0;
  norm_rand  $\leftarrow$  x;
  end;

```

128. Packed data. In order to make efficient use of storage space, TeX bases its major data structures on a *memory_word*, which contains either a (signed) integer, possibly scaled, or a (signed) *glue_ratio*, or a small number of fields that are one half or one quarter of the size used for storing integers.

If *x* is a variable of type *memory_word*, it contains up to four fields that can be referred to as follows:

<i>x.int</i>	(an <i>integer</i>)
<i>x.sc</i>	(a <i>scaled integer</i>)
<i>x.gr</i>	(a <i>glue_ratio</i>)
<i>x.hh.lh, x.hh.rh</i>	(two <i>halfword</i> fields)
<i>x.hh.b0, x.hh.b1, x.hh.rh</i>	(two <i>quarterword</i> fields, one <i>halfword</i> field)
<i>x.qqqq.b0, x.qqqq.b1, x.qqqq.b2, x.qqqq.b3</i>	(four <i>quarterword</i> fields)

This is somewhat cumbersome to write, and not very readable either, but macros will be used to make the notation shorter and more transparent. The Pascal code below gives a formal definition of *memory_word* and its subsidiary types, using packed variant records. TeX makes no assumptions about the relative positions of the fields within a word.

Since we are assuming 32-bit integers, a *halfword* must contain at least 16 bits, and a *quarterword* must contain at least 8 bits. But it doesn't hurt to have more bits; for example, with enough 36-bit words you might be able to have *mem_max* as large as 262142, which is eight times as much memory as anybody had during the first four years of TeX's existence.

N.B.: Valuable memory space will be dreadfully wasted unless TeX is compiled by a Pascal that packs all of the *memory_word* variants into the space of a single integer. This means, for example, that *glue_ratio* words should be *short_real* instead of *real* on some computers. Some Pascal compilers will pack an integer whose subrange is '0 .. 255' into an eight-bit field, but others insist on allocating space for an additional sign bit; on such systems you can get 256 values into a *quarterword* only if the subrange is '-128 .. 127'.

The present implementation tries to accommodate as many variations as possible, so it makes few assumptions. If integers having the subrange '*min_quarterword* .. *max_quarterword*' can be packed into a *quarterword*, and if integers having the subrange '*min_halfword* .. *max_halfword*' can be packed into a *halfword*, everything should work satisfactorily.

It is usually most efficient to have *min_quarterword* = *min_halfword* = 0, so one should try to achieve this unless it causes a severe problem. The values defined here are recommended for most 32-bit computers.

```
define min_quarterword = 0 {smallest allowable value in a quarterword}
define max_quarterword = 255 {largest allowable value in a quarterword}
define min_halfword ≡ 0 {smallest allowable value in a halfword}
define max_halfword ≡ 65535 {largest allowable value in a halfword}
```

129. Here are the inequalities that the *quarterword* and *halfword* values must satisfy (or rather, the inequalities that they mustn't satisfy):

```
< Check the "constant" values for consistency 14 > +≡
init if (mem_min ≠ mem_bot) ∨ (mem_max ≠ mem_top) then bad ← 10;
tini
if (mem_min > mem_bot) ∨ (mem_max < mem_top) then bad ← 10;
if (min_quarterword > 0) ∨ (max_quarterword < 127) then bad ← 11;
if (min_halfword > 0) ∨ (max_halfword < 32767) then bad ← 12;
if (min_quarterword < min_halfword) ∨ (max_quarterword > max_halfword) then bad ← 13;
if (mem_min < min_halfword) ∨ (mem_max ≥ max_halfword) ∨
    (mem_bot - mem_min > max_halfword + 1) then bad ← 14;
if (font_base < min_quarterword) ∨ (font_max > max_quarterword) then bad ← 15;
if font_max > font_base + 256 then bad ← 16;
if (save_size > max_halfword) ∨ (max_strings > max_halfword) then bad ← 17;
if buf_size > max_halfword then bad ← 18;
if max_quarterword - min_quarterword < 255 then bad ← 19;
```

130. The operation of adding or subtracting *min_quarterword* occurs quite frequently in TeX, so it is convenient to abbreviate this operation by using the macros *qi* and *qo* for input and output to and from quarterword format.

The inner loop of TeX will run faster with respect to compilers that don't optimize expressions like '*x + 0'* and '*x - 0'*, if these macros are simplified in the obvious way when *min_quarterword* = 0.

```
define qi(#) ≡ # + min_quarterword { to put an eight_bits item into a quarterword }
define qo(#) ≡ # - min_quarterword { to take an eight_bits item out of a quarterword }
define hi(#) ≡ # + min_halfword { to put a sixteen-bit item into a halfword }
define ho(#) ≡ # - min_halfword { to take a sixteen-bit item from a halfword }
```

131. The reader should study the following definitions closely:

```
define sc ≡ int { scaled data is equivalent to integer }
⟨Types in the outer block 18⟩ +≡
quarterword = min_quarterword .. max_quarterword; { 1/4 of a word }
halfword = min_halfword .. max_halfword; { 1/2 of a word }
two_choices = 1 .. 2; { used when there are two variants in a record }
four_choices = 1 .. 4; { used when there are four variants in a record }
two_halves = packed record rh: halfword;
  case two_choices of
    1: (lh : halfword);
    2: (b0 : quarterword; b1 : quarterword);
  end;
four_quarters = packed record b0: quarterword;
  b1: quarterword;
  b2: quarterword;
  b3: quarterword;
end;
memory_word = record
  case four_choices of
    1: (int : integer);
    2: (gr : glue_ratio);
    3: (hh : two_halves);
    4: (qqqq : four_quarters);
  end;
word_file = file of memory_word;
```

132. When debugging, we may want to print a *memory_word* without knowing what type it is; so we print it in all modes.

```
debug procedure print_word(w : memory_word); { prints w in all ways }
begin print_int(w.int); print_char(" ");
print_scaled(w.sc); print_char(" ");
print_scaled(round(unity * float(w.gr))); print_ln;
print_int(w.hh.lh); print_char("="); print_int(w.hh.b0); print_char(":"); print_int(w.hh.b1);
print_char(";"); print_int(w.hh.rh); print_char(" ");
print_int(w.qqqq.b0); print_char(":"); print_int(w.qqqq.b1); print_char(":"); print_int(w.qqqq.b2);
print_char(":"); print_int(w.qqqq.b3);
end;
gubed
```

133. Dynamic memory allocation. The TeX system does nearly all of its own memory allocation, so that it can readily be transported into environments that do not have automatic facilities for strings, garbage collection, etc., and so that it can be in control of what error messages the user receives. The dynamic storage requirements of TeX are handled by providing a large array *mem* in which consecutive blocks of words are used as nodes by the TeX routines.

Pointer variables are indices into this array, or into another array called *eqtb* that will be explained later. A pointer variable might also be a special flag that lies outside the bounds of *mem*, so we allow pointers to assume any *halfword* value. The minimum *halfword* value represents a null pointer. TeX does not assume that *mem*[*null*] exists.

```
define pointer ≡ halfword { a flag or a location in mem or eqtb }
define null ≡ min_halfword { the null pointer }

⟨ Global variables 13 ⟩ +≡
temp_ptr: pointer; { a pointer variable for occasional emergency use }
```

134. The *mem* array is divided into two regions that are allocated separately, but the dividing line between these two regions is not fixed; they grow together until finding their “natural” size in a particular job. Locations less than or equal to *lo_mem_max* are used for storing variable-length records consisting of two or more words each. This region is maintained using an algorithm similar to the one described in exercise 2.5–19 of *The Art of Computer Programming*. However, no size field appears in the allocated nodes; the program is responsible for knowing the relevant size when a node is freed. Locations greater than or equal to *hi_mem_min* are used for storing one-word records; a conventional AVAIL stack is used for allocation in this region.

Locations of *mem* between *mem_bot* and *mem_top* may be dumped as part of preloaded format files, by the INITEX preprocessor. Production versions of TeX may extend the memory at both ends in order to provide more space; locations between *mem_min* and *mem_bot* are always used for variable-size nodes, and locations between *mem_top* and *mem_max* are always used for single-word nodes.

The key pointers that govern *mem* allocation have a prescribed order:

$$\text{null} \leq \text{mem_min} \leq \text{mem_bot} < \text{lo_mem_max} < \text{hi_mem_min} < \text{mem_top} \leq \text{mem_end} \leq \text{mem_max}.$$

Empirical tests show that the present implementation of TeX tends to spend about 9% of its running time allocating nodes, and about 6% deallocating them after their use.

```
⟨ Global variables 13 ⟩ +≡
mem: array [mem_min .. mem_max] of memory_word; { the big dynamic storage area }
lo_mem_max: pointer; { the largest location of variable-size memory in use }
hi_mem_min: pointer; { the smallest location of one-word memory in use }
```

135. In order to study the memory requirements of particular applications, it is possible to prepare a version of TeX that keeps track of current and maximum memory usage. When code between the delimiters *stat ... tats* is not “commented out,” TeX will run a bit slower but it will report these statistics when *tracing_stats* is sufficiently large.

```
⟨ Global variables 13 ⟩ +≡
var_used, dyn_used: integer; { how much memory is in use }
```

136. Let's consider the one-word memory region first, since it's the simplest. The pointer variable *mem_end* holds the highest-numbered location of *mem* that has ever been used. The free locations of *mem* that occur between *hi_mem_min* and *mem_end*, inclusive, are of type *two_halves*, and we write *info(p)* and *link(p)* for the *lh* and *rh* fields of *mem[p]* when it is of this type. The single-word free locations form a linked list

avail, link(avail), link(link(avail)), ...

terminated by *null*.

```
define link(#) ≡ mem[#].hh.rh { the link field of a memory word }
define info(#) ≡ mem[#].hh.lh { the info field of a memory word }

⟨ Global variables 13 ⟩ +≡
avail: pointer; { head of the list of available one-word nodes }
mem.end: pointer; { the last one-word node used in mem }
```

137. If memory is exhausted, it might mean that the user has forgotten a right brace. We will define some procedures later that try to help pinpoint the trouble.

```
⟨ Declare the procedure called show_token_list 314 ⟩
⟨ Declare the procedure called runaway 328 ⟩
```

138. The function *get_avail* returns a pointer to a new one-word node whose *link* field is *null*. However, T_EX will halt if there is no more room left.

If the available-space list is empty, i.e., if *avail = null*, we try first to increase *mem_end*. If that cannot be done, i.e., if *mem_end = mem_max*, we try to decrease *hi_mem_min*. If that cannot be done, i.e., if *hi_mem_min = lo_mem_max + 1*, we have to quit.

```
function get_avail: pointer; { single-word node allocation }
  var p: pointer; { the new node being got }
  begin p ← avail; { get top location in the avail stack }
  if p ≠ null then avail ← link(avail) { and pop it off }
  else if mem_end < mem_max then { or go into virgin territory }
    begin incr(mem_end); p ← mem_end;
    end
  else begin decr(hi_mem_min); p ← hi_mem_min;
    if hi_mem_min ≤ lo_mem_max then
      begin runaway; { if memory is exhausted, display possible runaway text }
      overflow("main_memory_size", mem_max + 1 - mem_min); { quit; all one-word nodes are busy }
      end;
    end;
  link(p) ← null; { provide an oft-desired initialization of the new node }
  stat incr(dyn_used); tats { maintain statistics }
  get_avail ← p;
end;
```

139. Conversely, a one-word node is recycled by calling *free_avail*. This routine is part of T_EX's "inner loop," so we want it to be fast.

```
define free_avail(#) ≡ { single-word node liberation }
  begin link(#) ← avail; avail ← #;
  stat decr(dyn_used); tats
  end
```

140. There's also a *fast_get_avail* routine, which saves the procedure-call overhead at the expense of extra programming. This routine is used in the places that would otherwise account for the most calls of *get_avail*.

```
define fast_get_avail(#) =
    begin # ← avail; { avoid get_avail if possible, to save time }
    if # = null then # ← get_avail
    else begin avail ← link(#); link(#) ← null;
        stat incr(dyn_used); tats
    end;
end
```

141. The procedure *flush_list*(*p*) frees an entire linked list of one-word nodes that starts at position *p*.

```
procedure flush_list(p : pointer); { makes list of single-word nodes available }
var q, r: pointer; { list traversers }
begin if p ≠ null then
    begin r ← p;
    repeat q ← r; r ← link(r);
        stat decr(dyn_used); tats
    until r = null; { now q is the last node on the list }
    link(q) ← avail; avail ← p;
    end;
end;
```

142. The available-space list that keeps track of the variable-size portion of *mem* is a nonempty, doubly-linked circular list of empty nodes, pointed to by the roving pointer *rover*.

Each empty node has size 2 or more; the first word contains the special value *max_halfword* in its *link* field and the size in its *info* field; the second word contains the two pointers for double linking.

Each nonempty node also has size 2 or more. Its first word is of type *two_halves*, and its *link* field is never equal to *max_halfword*. Otherwise there is complete flexibility with respect to the contents of its other fields and its other words.

(We require *mem_max* < *max_halfword* because terrible things can happen when *max_halfword* appears in the *link* field of a nonempty node.)

```
define empty_flag ≡ max_halfword { the link of an empty variable-size node }
define is_empty(#) ≡ (link(#) = empty_flag) { tests for empty node }
define node_size ≡ info { the size field in empty variable-size nodes }
define llink(#) ≡ info(# + 1) { left link in doubly-linked list of empty nodes }
define rlink(#) ≡ link(# + 1) { right link in doubly-linked list of empty nodes }

{ Global variables 13 } +≡
rover: pointer; { points to some node in the list of empties }
```

143. A call to *get_node* with argument *s* returns a pointer to a new node of size *s*, which must be 2 or more. The *link* field of the first word of this new node is set to null. An overflow stop occurs if no suitable space exists.

If *get_node* is called with *s* = 2^{30} , it simply merges adjacent free areas and returns the value *max_halfword*.

```
function get_node(s : integer): pointer; { variable-size node allocation }
label found, exit, restart;
var p: pointer; { the node currently under inspection }
q: pointer; { the node physically after node p }
r: integer; { the newly allocated node, or a candidate for this honor }
t: integer; { temporary register }
begin restart: p ← rover; { start at some free node in the ring }
repeat {Try to allocate within node p and its physical successors, and goto found if allocation was
possible 145};
    p ← rlink(p); { move to the next node in the ring }
until p = rover; { repeat until the whole list has been traversed }
if s = '100000000000 then
    begin get_node ← max_halfword; return;
end;
if lo_mem_max + 2 < hi_mem_min then
    if lo_mem_max + 2 ≤ mem_bot + max_halfword then
        {Grow more variable-size memory and goto restart 144};
        overflow("main_memory_size", mem_max + 1 - mem_min); { sorry, nothing satisfactory is left }
found: link(r) ← null; { this node is now nonempty }
stat var_used ← var_used + s; { maintain usage statistics }
tats
get_node ← r;
exit: end;
```

144. The lower part of *mem* grows by 1000 words at a time, unless we are very close to going under. When it grows, we simply link a new node into the available-space list. This method of controlled growth helps to keep the *mem* usage consecutive when TeX is implemented on “virtual memory” systems.

```
{Grow more variable-size memory and goto restart 144} ≡
begin if hi_mem_min - lo_mem_max ≥ 1998 then t ← lo_mem_max + 1000
else t ← lo_mem_max + 1 + (hi_mem_min - lo_mem_max) div 2; { lo_mem_max + 2 ≤ t < hi_mem_min }
p ← llink(rover); q ← lo_mem_max; rlink(p) ← q; llink(rover) ← q;
if t > mem_bot + max_halfword then t ← mem_bot + max_halfword;
rlink(q) ← rover; llink(q) ← p; link(q) ← empty_flag; node_size(q) ← t - lo_mem_max;
lo_mem_max ← t; link(lo_mem_max) ← null; info(lo_mem_max) ← null; rover ← q; goto restart;
end
```

This code is used in section 143.

145. Empirical tests show that the routine in this section performs a node-merging operation about 0.75 times per allocation, on the average, after which it finds that $r > p + 1$ about 95% of the time.

```
< Try to allocate within node  $p$  and its physical successors, and goto found if allocation was possible 145 > ≡
 $q \leftarrow p + \text{node\_size}(p);$  { find the physical successor }
while is_empty( $q$ ) do { merge node  $p$  with node  $q$  }
  begin  $t \leftarrow \text{rlink}(q);$ 
  if  $q = \text{rover}$  then  $\text{rover} \leftarrow t;$ 
   $\text{llink}(t) \leftarrow \text{llink}(q); \text{rlink}(\text{llink}(q)) \leftarrow t;$ 
   $q \leftarrow q + \text{node\_size}(q);$ 
  end;
 $r \leftarrow q - s;$ 
if  $r > p + 1$  then < Allocate from the top of node  $p$  and goto found 146 >;
if  $r = p$  then
  if  $\text{rlink}(p) \neq p$  then < Allocate entire node  $p$  and goto found 147 >;
   $\text{node\_size}(p) \leftarrow q - p$  { reset the size in case it grew }
```

This code is used in section 143.

146. < Allocate from the top of node p and goto *found* 146 > ≡

```
begin  $\text{node\_size}(p) \leftarrow r - p;$  { store the remaining size }
 $\text{rover} \leftarrow p;$  { start searching here next time }
goto found;
end
```

This code is used in section 145.

147. Here we delete node p from the ring, and let rover rove around.

```
< Allocate entire node  $p$  and goto found 147 > ≡
begin  $\text{rover} \leftarrow \text{rlink}(p); t \leftarrow \text{llink}(p); \text{llink}(\text{rover}) \leftarrow t; \text{rlink}(t) \leftarrow \text{rover}; \text{goto}$  found;
end
```

This code is used in section 145.

148. Conversely, when some variable-size node p of size s is no longer needed, the operation $\text{free_node}(p, s)$ will make its words available, by inserting p as a new empty node just before where rover now points.

```
procedure  $\text{free\_node}(p : \text{pointer}; s : \text{halfword});$  { variable-size node liberation }
  var  $q : \text{pointer};$  {  $\text{llink}(\text{rover})$  }
  begin  $\text{node\_size}(p) \leftarrow s;$   $\text{link}(p) \leftarrow \text{empty\_flag};$   $q \leftarrow \text{llink}(\text{rover});$   $\text{llink}(p) \leftarrow q;$   $\text{rlink}(p) \leftarrow \text{rover};$ 
    { set both links }
     $\text{llink}(\text{rover}) \leftarrow p;$   $\text{rlink}(q) \leftarrow p;$  { insert  $p$  into the ring }
    stat  $\text{var\_used} \leftarrow \text{var\_used} - s;$  tats { maintain statistics }
  end;
```

149. Just before INITEX writes out the memory, it sorts the doubly linked available space list. The list is probably very short at such times, so a simple insertion sort is used. The smallest available location will be pointed to by *rover*, the next-smallest by *rlink(rover)*, etc.

```

init procedure sort_avail; { sorts the available variable-size nodes by location }
var p, q, r: pointer; { indices into mem }
    old_rover: pointer; { initial rover setting }
begin p  $\leftarrow$  get_node('100000000000); { merge adjacent free areas }
    p  $\leftarrow$  rlink(rover); rlink(rover)  $\leftarrow$  max_halfword; old_rover  $\leftarrow$  rover;
    while p  $\neq$  old_rover do {Sort p into the list starting at rover and advance p to rlink(p) 150};
        p  $\leftarrow$  rover;
        while rlink(p)  $\neq$  max_halfword do
            begin llink(rlink(p))  $\leftarrow$  p; p  $\leftarrow$  rlink(p);
            end;
            rlink(p)  $\leftarrow$  rover; llink(rover)  $\leftarrow$  p;
        end;
        tini
    
```

150. The following while loop is guaranteed to terminate, since the list that starts at *rover* ends with *max_halfword* during the sorting procedure.

```

{Sort p into the list starting at rover and advance p to rlink(p) 150}  $\equiv$ 
if p < rover then
    begin q  $\leftarrow$  p; p  $\leftarrow$  rlink(q); rlink(q)  $\leftarrow$  rover; rover  $\leftarrow$  q;
    end
else begin q  $\leftarrow$  rover;
    while rlink(q) < p do q  $\leftarrow$  rlink(q);
    r  $\leftarrow$  rlink(p); rlink(p)  $\leftarrow$  rlink(q); rlink(q)  $\leftarrow$  p; p  $\leftarrow$  r;
end
    
```

This code is used in section 149.

151. Data structures for boxes and their friends. From the computer's standpoint, TeX's chief mission is to create horizontal and vertical lists. We shall now investigate how the elements of these lists are represented internally as nodes in the dynamic memory.

A horizontal or vertical list is linked together by *link* fields in the first word of each node. Individual nodes represent boxes, glue, penalties, or special things like discretionary hyphens; because of this variety, some nodes are longer than others, and we must distinguish different kinds of nodes. We do this by putting a '*type*' field in the first word, together with the link and an optional '*subtype*'.

```
define type(#) ≡ mem[#].hh.b0 { identifies what kind of node this is }
define subtype(#) ≡ mem[#].hh.b1 { secondary identification in some cases }
```

152. A *char_node*, which represents a single character, is the most important kind of node because it accounts for the vast majority of all boxes. Special precautions are therefore taken to ensure that a *char_node* does not take up much memory space. Every such node is one word long, and in fact it is identifiable by this property, since other kinds of nodes have at least two words, and they appear in *mem* locations less than *hi_mem_min*. This makes it possible to omit the *type* field in a *char_node*, leaving us room for two bytes that identify a *font* and a *character* within that font.

Note that the format of a *char_node* allows for up to 256 different fonts and up to 256 characters per font; but most implementations will probably limit the total number of fonts to fewer than 75 per job, and most fonts will stick to characters whose codes are less than 128 (since higher codes are more difficult to access on most keyboards).

Extensions of TeX intended for oriental languages will need even more than 256×256 possible characters, when we consider different sizes and styles of type. It is suggested that Chinese and Japanese fonts be handled by representing such characters in two consecutive *char_node* entries: The first of these has *font* = *font_base*, and its *link* points to the second; the second identifies the font and the character dimensions. The *character* field of the first *char_node* is a "charext" that distinguishes between graphic symbols whose dimensions are identical for typesetting purposes. (See the METAFONT manual.) Such an extension of TeX would not be difficult; further details are left to the reader.

In order to make sure that the *character* code fits in a quarterword, TeX adds the quantity *min_quarterword* to the actual code.

Character nodes appear only in horizontal lists, never in vertical lists.

```
define is_char_node(#) ≡ (# ≥ hi_mem_min) { does the argument point to a char_node? }
define font ≡ type { the font code in a char_node }
define character ≡ subtype { the character code in a char_node }
```

153. 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 }
```

154. The *new_null_box* function returns a pointer to an *hlist_node* in which all subfields have the values corresponding to ‘\hbox{}`’. (The *subtype* field is set to *min_quarterword*, for historic reasons that are no longer relevant.)

```
function new_null_box: pointer; { creates a new box node }
  var p: pointer; { the new node }
begin p ← get_node(box_node_size); type(p) ← hlist_node; subtype(p) ← min_quarterword;
  width(p) ← 0; depth(p) ← 0; height(p) ← 0; shift_amount(p) ← 0; list_ptr(p) ← null;
  glue_sign(p) ← normal; glue_order(p) ← normal; set_glue_ratio_zero(glue_set(p)); new_null_box ← p;
end;
```

155. A *vlist_node* is like an *hlist_node* in all respects except that it contains a vertical list.

```
define vlist_node = 1 { type of vlist nodes }
```

156. A *rule_node* stands for a solid black rectangle; it has *width*, *depth*, and *height* fields just as in an *hlist_node*. However, if any of these dimensions is -2^{30} , the actual value will be determined by running the rule up to the boundary of the innermost enclosing box. This is called a “running dimension.” The *width* is never running in an *hlist*; the *height* and *depth* are never running in a *vlist*.

```
define rule_node = 2 { type of rule nodes }
define rule_node_size = 4 { number of words to allocate for a rule node }
define null_flag ≡ -10000000000 { -230, signifies a missing item }
define is_running(#) ≡ (# = null_flag) { tests for a running dimension }
```

157. A new rule node is delivered by the *new_rule* function. It makes all the dimensions “running” so you have to change the ones that are not allowed to run.

```
function new_rule: pointer;
  var p: pointer; { the new node }
begin p ← get_node(rule_node_size); type(p) ← rule_node; subtype(p) ← 0; { the subtype is not used }
  width(p) ← null_flag; depth(p) ← null_flag; height(p) ← null_flag; new_rule ← p;
end;
```

158. Insertions are represented by *ins_node* records, where the *subtype* indicates the corresponding box number. For example, ‘\insert 250’ leads to an *ins_node* whose *subtype* is $250 + \text{min_quarterword}$. The *height* field of an *ins_node* is slightly misnamed; it actually holds the natural height plus depth of the vertical list being inserted. The *depth* field holds the *split_max_depth* to be used in case this insertion is split, and the *split_top_ptr* points to the corresponding *split_top_skip*. The *float_cost* field holds the *floating_penalty* that will be used if this insertion floats to a subsequent page after a split insertion of the same class. There is one more field, the *ins_ptr*, which points to the beginning of the *vlist* for the insertion.

```
define ins_node = 3 { type of insertion nodes }
define ins_node_size = 5 { number of words to allocate for an insertion }
define float_cost(#) ≡ mem[# + 1].int { the floating_penalty to be used }
define ins_ptr(#) ≡ info(# + 4) { the vertical list to be inserted }
define split_top_ptr(#) ≡ link(# + 4) { the split_top_skip to be used }
```

159. 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 }
```

160. 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_pre ≡ subtype { if 0 = pre-adjustment }
  { append_list is used to append a list to tail }
define append_list(#) ≡
  begin link(tail) ← link(#); append_list_end
define append_list_end(#) ≡ tail ← #;
  end
define adjust_ptr(#) ≡ mem[# + 1].int { vertical list to be moved out of horizontal list }
```

161. A *ligature_node*, which occurs only in horizontal lists, specifies a character that was fabricated from the interaction of two or more actual characters. The second word of the node, which is called the *lig_char* word, contains *font* and *character* fields just as in a *char_node*. The characters that generated the ligature have not been forgotten, since they are needed for diagnostic messages and for hyphenation; the *lig_ptr* field points to a linked list of character nodes for all original characters that have been deleted. (This list might be empty if the characters that generated the ligature were retained in other nodes.)

The *subtype* field is 0, plus 2 and/or 1 if the original source of the ligature included implicit left and/or right boundaries.

```
define ligature_node = 6 { type of a ligature node }
define lig_char(#) ≡ # + 1 { the word where the ligature is to be found }
define lig_ptr(#) ≡ link(lig_char(#)) { the list of characters }
```

162. The *new_ligature* function creates a ligature node having given contents of the *font*, *character*, and *lig_ptr* fields. We also have a *new_lig_item* function, which returns a two-word node having a given *character* field. Such nodes are used for temporary processing as ligatures are being created.

```
function new_ligature(f, c : quarterword; q : pointer): pointer;
  var p: pointer; { the new node }
  begin p ← get_node(small_node_size); type(p) ← ligature_node; font(lig_char(p)) ← f;
    character(lig_char(p)) ← c; lig_ptr(p) ← q; subtype(p) ← 0; new_ligature ← p;
  end;

function new_lig_item(c : quarterword): pointer;
  var p: pointer; { the new node }
  begin p ← get_node(small_node_size); character(p) ← c; lig_ptr(p) ← null; new_lig_item ← p;
  end;
```

163. A *disc_node*, which occurs only in horizontal lists, specifies a “discretionary” line break. If such a break occurs at node *p*, the text that starts at *pre_break(p)* will precede the break, the text that starts at *post_break(p)* will follow the break, and text that appears in the next *replace_count(p)* nodes will be ignored. For example, an ordinary discretionary hyphen, indicated by ‘\-’, yields a *disc_node* with *pre_break* pointing to a *char_node* containing a hyphen, *post_break* = *null*, and *replace_count* = 0. All three of the discretionary texts must be lists that consist entirely of character, kern, box, rule, and ligature nodes.

If *pre_break(p)* = *null*, the *ex_hyphen_penalty* will be charged for this break. Otherwise the *hyphen_penalty* will be charged. The texts will actually be substituted into the list by the line-breaking algorithm if it decides to make the break, and the discretionary node will disappear at that time; thus, the output routine sees only discretionaries that were not chosen.

```
define disc_node = 7 { type of a discretionary node }
define replace_count ≡ subtype { how many subsequent nodes to replace }
define pre_break ≡ llink { text that precedes a discretionary break }
define post_break ≡ rlink { text that follows a discretionary break }

function new_disc: pointer; { creates an empty disc_node }
  var p: pointer; { the new node }
  begin p ← get_node(small_node_size); type(p) ← disc_node; replace_count(p) ← 0; pre_break(p) ← null;
    post_break(p) ← null; new_disc ← p;
  end;
```

164. A *whatsit_node* is a wild card reserved for extensions to T_EX. The *subtype* field in its first word says what ‘*whatsit*’ it is, and implicitly determines the node size (which must be 2 or more) and the format of the remaining words. When a *whatsit_node* is encountered in a list, special actions are invoked; knowledgeable people who are careful not to mess up the rest of T_EX are able to make T_EX do new things by adding code at the end of the program. For example, there might be a ‘T_EXnicolor’ extension to specify different colors of ink, and the *whatsit* node might contain the desired parameters.

The present implementation of T_EX treats the features associated with ‘\write’ and ‘\special’ as if they were extensions, in order to illustrate how such routines might be coded. We shall defer further discussion of extensions until the end of this program.

```
define whatsit_node = 8 { type of special extension nodes }
```

165. 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 (\begin{L}, \end{L}, \begin{R}, and \end{R}).

```
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 \begin{M} node }
define end_M_code = M_code + after { subtype for \end{M} node }
define L_code = 4
define begin_L_code = L_code + begin_M_code { subtype for \begin{L} node }
define end_L_code = L_code + end_M_code { subtype for \end{L} node }
define R_code = L_code + L_code
define begin_R_code = R_code + begin_M_code { subtype for \begin{R} node }
define end_R_code = R_code + end_M_code { subtype for \end{R} 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;
```

166. T_EX makes use of the fact that *hlist_node*, *vlist_node*, *rule_node*, *ins_node*, *mark_node*, *adjust_node*, *ligature_node*, *disc_node*, *whatsit_node*, and *math_node* are at the low end of the type codes, by permitting a break at glue in a list if and only if the *type* of the previous node is less than *math_node*. Furthermore, a node is discarded after a break if its type is *math_node* or more.

```
define precedes_break(#) ≡ (type(#) < math_node)
define non_discardable(#) ≡ (type(#) < math_node)
```

167. A *glue_node* represents glue in a list. However, it is really only a pointer to a separate glue specification, since TeX makes use of the fact that many essentially identical nodes of glue are usually present. If *p* points to a *glue_node*, *glue_ptr(p)* points to another packet of words that specify the stretch and shrink components, etc.

Glue nodes also serve to represent leaders; the *subtype* is used to distinguish between ordinary glue (which is called *normal*) and the three kinds of leaders (which are called *a_leaders*, *c_leaders*, and *x_leaders*). The *leader_ptr* field points to a rule node or to a box node containing the leaders; it is set to *null* in ordinary glue nodes.

Many kinds of glue are computed from TeX's "skip" parameters, and it is helpful to know which parameter has led to a particular glue node. Therefore the *subtype* is set to indicate the source of glue, whenever it originated as a parameter. We will be defining symbolic names for the parameter numbers later (e.g., *line_skip_code* = 0, *baseline_skip_code* = 1, etc.); it suffices for now to say that the *subtype* of parametric glue will be the same as the parameter number, plus one.

In math formulas there are two more possibilities for the *subtype* in a glue node: *mu_glue* denotes an *\mskip* (where the units are scaled *mu* instead of scaled *pt*); and *cond_math_glue* denotes the '*\nonscript*' feature that cancels the glue node immediately following if it appears in a subscript.

```
define glue_node = 10 { type of node that points to a glue specification }
define cond_math_glue = 98 { special subtype to suppress glue in the next node }
define mu_glue = 99 { subtype for math glue }
define a_leaders = 100 { subtype for aligned leaders }
define c_leaders = 101 { subtype for centered leaders }
define x_leaders = 102 { subtype for expanded leaders }
define glue_ptr ≡ llink { pointer to a glue specification }
define leader_ptr ≡ rlink { pointer to box or rule node for leaders }
```

168. A glue specification has a halfword reference count in its first word, representing *null* plus the number of glue nodes that point to it (less one). Note that the reference count appears in the same position as the *link* field in list nodes; this is the field that is initialized to *null* when a node is allocated, and it is also the field that is flagged by *empty_flag* in empty nodes.

Glue specifications also contain three *scaled* fields, for the *width*, *stretch*, and *shrink* dimensions. Finally, there are two one-byte fields called *stretch_order* and *shrink_order*; these contain the orders of infinity (*normal*, *fil*, *fill*, or *filll*) corresponding to the stretch and shrink values.

```
define glue_spec_size = 4 { number of words to allocate for a glue specification }
define glue_ref_count(#) ≡ link(#) { reference count of a glue specification }
define stretch(#) ≡ mem[# + 2].sc { the stretchability of this glob of glue }
define shrink(#) ≡ mem[# + 3].sc { the shrinkability of this glob of glue }
define stretch_order ≡ type { order of infinity for stretching }
define shrink_order ≡ subtype { order of infinity for shrinking }
define fil = 1 { first-order infinity }
define fill = 2 { second-order infinity }
define filll = 3 { third-order infinity }

⟨ Types in the outer block 18 ⟩ +≡
glue_ord = normal .. filll; { infinity to the 0, 1, 2, or 3 power }
```

169. Here is a function that returns a pointer to a copy of a glue spec. The reference count in the copy is *null*, because there is assumed to be exactly one reference to the new specification.

```
function new_spec(p : pointer): pointer; { duplicates a glue specification }
  var q: pointer; { the new spec }
  begin q ← get_node(glue_spec_size);
    mem[q] ← mem[p]; glue_ref_count(q) ← null;
    width(q) ← width(p); stretch(q) ← stretch(p); shrink(q) ← shrink(p); new_spec ← q;
  end;
```

170. And here's a function that creates a glue node for a given parameter identified by its code number; for example, *new_param_glue(line_skip_code)* returns a pointer to a glue node for the current \lineskip.

```
function new_param_glue(n : small_number): pointer;
  var p: pointer; { the new node }
  q: pointer; { the glue specification }
  begin p ← get_node(small_node_size); type(p) ← glue_node; subtype(p) ← n + 1; leader_ptr(p) ← null;
    q ← ⟨Current mem equivalent of glue parameter number n 242⟩; glue_ptr(p) ← q;
    incr(glue_ref_count(q)); new_param_glue ← p;
  end;
```

171. Glue nodes that are more or less anonymous are created by *new_glue*, whose argument points to a glue specification.

```
function new_glue(q : pointer): pointer;
  var p: pointer; { the new node }
  begin p ← get_node(small_node_size); type(p) ← glue_node; subtype(p) ← normal;
    leader_ptr(p) ← null; glue_ptr(p) ← q; incr(glue_ref_count(q)); new_glue ← p;
  end;
```

172. Still another subroutine is needed: This one is sort of a combination of *new_param_glue* and *new_glue*. It creates a glue node for one of the current glue parameters, but it makes a fresh copy of the glue specification, since that specification will probably be subject to change, while the parameter will stay put. The global variable *temp_ptr* is set to the address of the new spec.

```
function new_skip_param(n : small_number): pointer;
  var p: pointer; { the new node }
  begin temp_ptr ← new_spec(⟨Current mem equivalent of glue parameter number n 242⟩);
    p ← new_glue(temp_ptr); glue_ref_count(temp_ptr) ← null; subtype(p) ← n + 1; new_skip_param ← p;
  end;
```

173. A *kern_node* has a *width* field to specify a (normally negative) amount of spacing. This spacing correction appears in horizontal lists between letters like A and V when the font designer said that it looks better to move them closer together or further apart. A kern node can also appear in a vertical list, when its '*width*' denotes additional spacing in the vertical direction. The *subtype* is either *normal* (for kerns inserted from font information or math mode calculations) or *explicit* (for kerns inserted from \kern and \v commands) or *acc.kern* (for kerns inserted from non-math accents) or *mu_glue* (for kerns inserted from \mkern specifications in math formulas).

```

define kern_node = 11 { type of a kern node }
define explicit = 1 { subtype of kern nodes from \kern and \v }
define acc_kern = 2 { subtype of kern nodes from accents }
define auto_kern = 3 { subtype of kern nodes created by get_auto_kern }

{ memory structure for marginal kerns }
define margin_kern_node = 40
define margin_kern_node_size = 3
define margin_char(#) $\equiv$ info(# + 2)
{ subtype of marginal kerns }
define left_side  $\equiv$  0
define right_side  $\equiv$  1
{ base for lp/rp/ef codes starts from 2: 0 for hyphen_char, 1 for skew_char }
define lp_code_base  $\equiv$  2
define rp_code_base  $\equiv$  3
define ef_code_base  $\equiv$  4
define tag_code  $\equiv$  5
define kn_bs_code_base  $\equiv$  7
define st_bs_code_base  $\equiv$  8
define sh_bs_code_base  $\equiv$  9
define kn_bc_code_base  $\equiv$  10
define kn_ac_code_base  $\equiv$  11
define no_lig_code  $\equiv$  6
define max_hlist_stack = 512 { maximum fill level for hlist_stack }
{ maybe good if larger than 2 * max_quarterword, so that box nesting level would overflow first }

```

174. The *new_kern* function creates a kern node having a given width.

```

function new_kern(w : scaled): pointer;
  var p: pointer; { the new node }
  begin p  $\leftarrow$  get_node(small_node_size); type(p)  $\leftarrow$  kern_node; subtype(p)  $\leftarrow$  normal; width(p)  $\leftarrow$  w;
  new_kern  $\leftarrow$  p;
  end;

```

175. A *penalty_node* specifies the penalty associated with line or page breaking, in its *penalty* field. This field is a fullword integer, but the full range of integer values is not used: Any penalty ≥ 10000 is treated as infinity, and no break will be allowed for such high values. Similarly, any penalty ≤ -10000 is treated as negative infinity, and a break will be forced.

```

define penalty_node = 12 { type of a penalty node }
define inf_penalty = inf_bad { "infinite" penalty value }
define eject_penalty = -inf_penalty { "negatively infinite" penalty value }
define penalty(#) $\equiv$ mem[# + 1].int { the added cost of breaking a list here }

```

176. Anyone who has been reading the last few sections of the program will be able to guess what comes next.

```
function new_penalty(m : integer): pointer;
  var p: pointer; { the new node }
begin p ← get_node(small_node_size); type(p) ← penalty_node; subtype(p) ← 0;
{ the subtype is not used }
penalty(p) ← m; new_penalty ← p;
end;
```

177. You might think that we have introduced enough node types by now. Well, almost, but there is one more: An *unset_node* has nearly the same format as an *hlist_node* or *vlist_node*; it is used for entries in \halign or \valign that are not yet in their final form, since the box dimensions are their “natural” sizes before any glue adjustment has been made. The *glue_set* word is not present; instead, we have a *glue_stretch* field, which contains the total stretch of order *glue_order* that is present in the hlist or vlist being boxed. Similarly, the *shift_amount* field is replaced by a *glue_shrink* field, containing the total shrink of order *glue_sign* that is present. The *subtype* field is called *span_count*; an unset box typically contains the data for $qo(span_count) + 1$ columns. Unset nodes will be changed to box nodes when alignment is completed.

```
define unset_node = 13 { type for an unset node }
define glue_stretch(#) ≡ mem[# + glue_offset].sc { total stretch in an unset node }
define glue_shrink ≡ shift_amount { total shrink in an unset node }
define span_count ≡ subtype { indicates the number of spanned columns }
```

178. In fact, there are still more types coming. When we get to math formula processing we will see that a *style_node* has *type* = 14; and a number of larger type codes will also be defined, for use in math mode only.

179. Warning: If any changes are made to these data structure layouts, such as changing any of the node sizes or even reordering the words of nodes, the *copy_node_list* procedure and the memory initialization code below may have to be changed. Such potentially dangerous parts of the program are listed in the index under ‘data structure assumptions’. However, other references to the nodes are made symbolically in terms of the WEB macro definitions above, so that format changes will leave T_EX’s other algorithms intact.

180. Memory layout. Some areas of *mem* are dedicated to fixed usage, since static allocation is more efficient than dynamic allocation when we can get away with it. For example, locations *mem_bot* to *mem_bot* + 3 are always used to store the specification for glue that is ‘0pt plus 0pt minus 0pt’. The following macro definitions accomplish the static allocation by giving symbolic names to the fixed positions. Static variable-size nodes appear in locations *mem_bot* through *lo_mem_stat_max*, and static single-word nodes appear in locations *hi_mem_stat_min* through *mem_top*, inclusive. It is harmless to let *lig_trick* and *garbage* share the same location of *mem*.

```

define zero_glue ≡ mem_bot { specification for 0pt plus 0pt minus 0pt }
define fil_glue ≡ zero_glue + glue_spec_size { 0pt plus 1fil minus 0pt }
define fill_glue ≡ fil_glue + glue_spec_size { 0pt plus 1fill minus 0pt }
define ss_glue ≡ fill_glue + glue_spec_size { 0pt plus 1fil minus 1fil }
define fil_neg_glue ≡ ss_glue + glue_spec_size { 0pt plus -1fil minus 0pt }
define lo_mem_stat_max ≡ fil_neg_glue + glue_spec_size - 1
                           { largest statically allocated word in the variable-size mem }

define page_ins_head ≡ mem_top { list of insertion data for current page }
define contrib_head ≡ mem_top - 1 { vlist of items not yet on current page }
define page_head ≡ mem_top - 2 { vlist for current page }
define temp_head ≡ mem_top - 3 { head of a temporary list of some kind }
define hold_head ≡ mem_top - 4 { head of a temporary list of another kind }
define adjust_head ≡ mem_top - 5 { head of adjustment list returned by hpack }
define active ≡ mem_top - 7 { head of active list in line_break, needs two words }
define align_head ≡ mem_top - 8 { head of preamble list for alignments }
define end_span ≡ mem_top - 9 { tail of spanned-width lists }
define omit_template ≡ mem_top - 10 { a constant token list }
define null_list ≡ mem_top - 11 { permanently empty list }
define lig_trick ≡ mem_top - 12 { a ligature masquerading as a char_node }
define garbage ≡ mem_top - 12 { used for scrap information }
define backup_head ≡ mem_top - 13 { head of token list built by scan_keyword }
define pre_adjust_head ≡ mem_top - 14 { head of pre-adjustment list returned by hpack }
define hi_mem_stat_min ≡ mem_top - 14 { smallest statically allocated word in the one-word mem }
define hi_mem_stat_usage = 15 { the number of one-word nodes always present }

```

181. The following code gets *mem* off to a good start, when TeX is initializing itself the slow way.

```

⟨ Local variables for initialization 19 ⟩ +≡
k: integer; { index into mem, eqtb, etc. }

```

182. ⟨ Initialize table entries (done by INITEX only) 182 ⟩ ≡

```

for  $k \leftarrow mem\_bot + 1$  to  $lo\_mem\_stat\_max$  do  $mem[k].sc \leftarrow 0$ ; { all glue dimensions are zeroed }
 $k \leftarrow mem\_bot$ ; while  $k \leq lo\_mem\_stat\_max$  do { set first words of glue specifications }
begin  $glue\_ref\_count(k) \leftarrow null + 1$ ;  $stretch\_order(k) \leftarrow normal$ ;  $shrink\_order(k) \leftarrow normal$ ;
 $k \leftarrow k + glue\_spec\_size$ ;
end;
 $stretch(fil\_glue) \leftarrow unity$ ;  $stretch\_order(fil\_glue) \leftarrow fil$ ;
 $stretch(fill\_glue) \leftarrow unity$ ;  $stretch\_order(fill\_glue) \leftarrow fill$ ;
 $stretch(ss\_glue) \leftarrow unity$ ;  $stretch\_order(ss\_glue) \leftarrow fil$ ;
 $shrink(ss\_glue) \leftarrow unity$ ;  $shrink\_order(ss\_glue) \leftarrow fil$ ;
 $stretch(fil\_neg\_glue) \leftarrow -unity$ ;  $stretch\_order(fil\_neg\_glue) \leftarrow fil$ ;
 $rover \leftarrow lo\_mem\_stat\_max + 1$ ;  $link(rover) \leftarrow empty\_flag$ ; { now initialize the dynamic memory }
 $node\_size(rover) \leftarrow 1000$ ; { which is a 1000-word available node }
 $llink(rover) \leftarrow rover$ ;  $rlink(rover) \leftarrow rover$ ;
 $lo\_mem\_max \leftarrow rover + 1000$ ;  $link(lo\_mem\_max) \leftarrow null$ ;  $info(lo\_mem\_max) \leftarrow null$ ;
for  $k \leftarrow hi\_mem\_stat\_min$  to  $mem\_top$  do  $mem[k] \leftarrow mem[lo\_mem\_max]$ ; { clear list heads }
⟨ Initialize the special list heads and constant nodes 966 ⟩;
 $avail \leftarrow null$ ;  $mem\_end \leftarrow mem\_top$ ;  $hi\_mem\_min \leftarrow hi\_mem\_stat\_min$ ;
{ initialize the one-word memory }
 $var\_used \leftarrow lo\_mem\_stat\_max + 1 - mem\_bot$ ;  $dyn\_used \leftarrow hi\_mem\_stat\_usage$ ; { initialize statistics }

```

See also sections 240, 246, 250, 258, 268, 277, 578, 672, 1064, 1123, 1128, 1394, 1479, 1616, 1653, 1818, and 1854.

This code is used in section 8.

183. If TeX is extended improperly, the *mem* array might get screwed up. For example, some pointers might be wrong, or some “dead” nodes might not have been freed when the last reference to them disappeared. Procedures *check_mem* and *search_mem* are available to help diagnose such problems. These procedures make use of two arrays called *free* and *was_free* that are present only if TeX’s debugging routines have been included. (You may want to decrease the size of *mem* while you are debugging.)

⟨ Global variables 13 ⟩ +≡

```

debug free: packed array [ $mem\_min .. mem\_max$ ] of boolean; { free cells }
was_free: packed array [ $mem\_min .. mem\_max$ ] of boolean; { previously free cells }
was_mem_end, was_lo_max, was_hi_min: pointer; { previous mem_end, lo_mem_max, and hi_mem_min }
panicking: boolean; { do we want to check memory constantly? }
gubed

```

184. ⟨ Set initial values of key variables 21 ⟩ +≡

```

debug was_mem_end  $\leftarrow mem\_min$ ; { indicate that everything was previously free }
was_lo_max  $\leftarrow mem\_min$ ; was_hi_min  $\leftarrow mem\_max$ ; panicking  $\leftarrow false$ ;
gubed

```

185. Procedure *check_mem* makes sure that the available space lists of *mem* are well formed, and it optionally prints out all locations that are reserved now but were free the last time this procedure was called.

```

debug procedure check_mem(print_locs : boolean);
label done1, done2; { loop exits }
var p, q: pointer; { current locations of interest in mem }
  clobbered: boolean; { is something amiss? }
begin for p  $\leftarrow$  mem_min to lo_mem_max do free[p]  $\leftarrow$  false; { you can probably do this faster }
  for p  $\leftarrow$  hi_mem_min to mem_end do free[p]  $\leftarrow$  false; { ditto }
  { Check single-word avail list 186 };
  { Check variable-size avail list 187 };
  { Check flags of unavailable nodes 188 };
  if print_locs then { Print newly busy locations 189 };
  for p  $\leftarrow$  mem_min to lo_mem_max do was_free[p]  $\leftarrow$  free[p];
  for p  $\leftarrow$  hi_mem_min to mem_end do was_free[p]  $\leftarrow$  free[p]; { was_free  $\leftarrow$  free might be faster }
  was_mem_end  $\leftarrow$  mem_end; was_lo_max  $\leftarrow$  lo_mem_max; was_hi_min  $\leftarrow$  hi_mem_min;
  end;
gubed
```

186. { Check single-word avail list 186 } \equiv

```

p  $\leftarrow$  avail; q  $\leftarrow$  null; clobbered  $\leftarrow$  false;
while p  $\neq$  null do
  begin if (p  $>$  mem_end)  $\vee$  (p  $<$  hi_mem_min) then clobbered  $\leftarrow$  true
  else if free[p] then clobbered  $\leftarrow$  true;
  if clobbered then
    begin print_nl("AVAIL_list_clobbered_at"); print_int(q); goto done1;
    end;
  free[p]  $\leftarrow$  true; q  $\leftarrow$  p; p  $\leftarrow$  link(q);
  end;
done1:
```

This code is used in section 185.

187. { Check variable-size avail list 187 } \equiv

```

p  $\leftarrow$  rover; q  $\leftarrow$  null; clobbered  $\leftarrow$  false;
repeat if (p  $\geq$  lo_mem_max)  $\vee$  (p  $<$  mem_min) then clobbered  $\leftarrow$  true
  else if (rlink(p)  $\geq$  lo_mem_max)  $\vee$  (rlink(p)  $<$  mem_min) then clobbered  $\leftarrow$  true
  else if  $\neg$ (is_empty(p))  $\vee$  (node_size(p)  $<$  2)  $\vee$  (p + node_size(p)  $>$  lo_mem_max)  $\vee$ 
    (llink(rlink(p))  $\neq$  p) then clobbered  $\leftarrow$  true;
  if clobbered then
    begin print_nl("Double-AVAIL_list_clobbered_at"); print_int(q); goto done2;
    end;
  for q  $\leftarrow$  p to p + node_size(p) - 1 do { mark all locations free }
    begin if free[q] then
      begin print_nl("Doubly-free_location_at"); print_int(q); goto done2;
      end;
    free[q]  $\leftarrow$  true;
    end;
    q  $\leftarrow$  p; p  $\leftarrow$  rlink(p);
  until p = rover;
done2:
```

This code is used in section 185.

188. ⟨Check flags of unavailable nodes 188⟩ ≡

```

 $p \leftarrow mem\_min;$ 
while  $p \leq lo\_mem\_max$  do { node  $p$  should not be empty }
  begin if  $is\_empty(p)$  then
    begin  $print\_nl("Bad\_\underline{f}\underline{l}\underline{a}\underline{g}\_\underline{a}\underline{t}\_\underline{")}; print\_int(p);$ 
    end;
    while  $(p \leq lo\_mem\_max) \wedge \neg free[p]$  do  $incr(p);$ 
    while  $(p \leq lo\_mem\_max) \wedge free[p]$  do  $incr(p);$ 
  end

```

This code is used in section 185.

189. ⟨Print newly busy locations 189⟩ ≡

```

begin  $print\_nl("New\_\underline{b}\underline{u}\underline{s}\underline{y}\underline{l}\underline{o}\underline{c}\underline{s}:\");$ 
for  $p \leftarrow mem\_min$  to  $lo\_mem\_max$  do
  if  $\neg free[p] \wedge ((p > was\_lo\_max) \vee was\_free[p])$  then
    begin  $print\_char(" \underline{\lrcorner} "); print\_int(p);$ 
    end;
for  $p \leftarrow hi\_mem\_min$  to  $mem\_end$  do
  if  $\neg free[p] \wedge ((p < was\_hi\_min) \vee (p > was\_mem\_end) \vee was\_free[p])$  then
    begin  $print\_char(" \underline{\lrcorner} "); print\_int(p);$ 
    end;
  end

```

This code is used in section 185.

190. The *search_mem* procedure attempts to answer the question “Who points to node p ?” In doing so, it fetches *link* and *info* fields of *mem* that might not be of type *two_halves*. Strictly speaking, this is undefined in Pascal, and it can lead to “false drops” (words that seem to point to p purely by coincidence). But for debugging purposes, we want to rule out the places that do *not* point to p , so a few false drops are tolerable.

```

debug procedure  $search\_mem(p : pointer);$  { look for pointers to  $p$  }
var  $q : integer;$  { current position being searched }
begin for  $q \leftarrow mem\_min$  to  $lo\_mem\_max$  do
  begin if  $link(q) = p$  then
    begin  $print\_nl("LINK("); print\_int(q); print\_char("));$ 
    end;
  if  $info(q) = p$  then
    begin  $print\_nl("INFO("); print\_int(q); print\_char("));$ 
    end;
  end;
for  $q \leftarrow hi\_mem\_min$  to  $mem\_end$  do
  begin if  $link(q) = p$  then
    begin  $print\_nl("LINK("); print\_int(q); print\_char("));$ 
    end;
  if  $info(q) = p$  then
    begin  $print\_nl("INFO("); print\_int(q); print\_char("));$ 
    end;
  end;
⟨ Search eqtb for equivalents equal to  $p$  273 };
⟨ Search save_stack for equivalents that point to  $p$  307 };
⟨ Search hyph_list for pointers to  $p$  1110 };
end;
gubed⟨ Declare procedures that need to be declared forward for pdfTeX 686 ⟩

```

191. Displaying boxes. We can reinforce our knowledge of the data structures just introduced by considering two procedures that display a list in symbolic form. The first of these, called *short_display*, is used in “overfull box” messages to give the top-level description of a list. The other one, called *show_node_list*, prints a detailed description of exactly what is in the data structure.

The philosophy of *short_display* is to ignore the fine points about exactly what is inside boxes, except that ligatures and discretionary breaks are expanded. As a result, *short_display* is a recursive procedure, but the recursion is never more than one level deep.

A global variable *font_in_short_display* keeps track of the font code that is assumed to be present when *short_display* begins; deviations from this font will be printed.

```
(Global variables 13) +≡
font_in_short_display: integer; { an internal font number }
```

192. Boxes, rules, inserts, whatsits, marks, and things in general that are sort of “complicated” are indicated only by printing ‘[]’.

```
procedure print_font_identifier(f: internal_font_number);
begin if pdf_font_blink[f] = null_font then print_esc(font_id_text(f))
else print_esc(font_id_text(pdf_font_blink[f]));
if pdf_tracing_fonts > 0 then
begin print("↳"); print(font_name[f]);
if font_size[f] ≠ font_dsize[f] then
begin print("⑧"); print_scaled(font_size[f]); print("pt");
end;
print(")");
end
else if pdf_font_expand_ratio[f] ≠ 0 then
begin print("↳");
if pdf_font_expand_ratio[f] > 0 then print("+");
print_int(pdf_font_expand_ratio[f]); print(")");
end;
end;
procedure short_display(p: integer); { prints highlights of list p }
var n: integer; { for replacement counts }
begin while p > mem_min do
begin if is_char_node(p) then
begin if p ≤ mem_end then
begin if font(p) ≠ font_in_short_display then
begin if (font(p) < font_base) ∨ (font(p) > font_max) then print_char("*")
else print_font_identifier(font(p));
print_char("↳"); font_in_short_display ← font(p);
end;
print_ASCII(qo(character(p)));
end;
end
else { Print a short indication of the contents of node p 193 };
p ← link(p);
end;
end;
```

193. ⟨ Print a short indication of the contents of node p 193 ⟩ ≡

```

case type( $p$ ) of
  hlist_node, vlist_node, ins_node, whatsit_node, mark_node, adjust_node, unset_node: print("[]");
  rule_node: print_char("I");
  glue_node: if glue_ptr( $p$ ) ≠ zero_glue then print_char("u");
  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 sections 192 and 674.

194. The *show_node_list* routine requires some auxiliary subroutines: one to print a font-and-character combination, one to print a token list without its reference count, and one to print a rule dimension.

```

procedure print_font_and_char( $p$  : integer); { prints char_node data }
  begin if  $p$  > mem_end then print_esc("CLOBBERED.");
  else begin if (font( $p$ ) < font_base) ∨ (font( $p$ ) > font_max) then print_char("*")
    else print_font_identifier(font( $p$ ));
    print_char("u"); print_ASCII(qo(character( $p$ )));
    end;
  end;

procedure print_mark( $p$  : integer); { prints token list data in braces }
  begin print_char("{");
  if ( $p$  < hi_mem_min) ∨ ( $p$  > mem_end) then print_esc("CLOBBERED.")
  else show_token_list(link( $p$ ), null, max_print_line - 10);
  print_char("}");
  end;

procedure print_rule_dimen( $d$  : scaled); { prints dimension in rule node }
  begin if is_running( $d$ ) then print_char("*")
  else print_scaled( $d$ );
  end;
```

195. Then there is a subroutine that prints glue stretch and shrink, possibly followed by the name of finite units:

```
procedure print_glue(d : scaled; order : integer; s : str_number); { prints a glue component }
  begin print_scaled(d);
  if (order < normal)  $\vee$  (order > fill) then print("foul")
  else if order > normal then
    begin print("fil");
    while order > fil do
      begin print_char("1"); decr(order);
      end;
    end
  else if s  $\neq$  0 then print(s);
  end;
```

196. The next subroutine prints a whole glue specification.

```
procedure print_spec(p : integer; s : str_number); { prints a glue specification }
  begin if (p < mem_min)  $\vee$  (p  $\geq$  lo_mem_max) then print_char("*")
  else begin print_scaled(width(p));
  if s  $\neq$  0 then print(s);
  if stretch(p)  $\neq$  0 then
    begin print("plus"); print_glue(stretch(p), stretch_order(p), s);
    end;
  if shrink(p)  $\neq$  0 then
    begin print("minus"); print_glue(shrink(p), shrink_order(p), s);
    end;
  end;
  end;
```

197. We also need to declare some procedures that appear later in this documentation.

⟨ Declare procedures needed for displaying the elements of mlists 867 ⟩
 ⟨ Declare the procedure called *print_skip_param* 243 ⟩

198. Since boxes can be inside of boxes, *show_node_list* is inherently recursive, up to a given maximum number of levels. The history of nesting is indicated by the current string, which will be printed at the beginning of each line; the length of this string, namely *cur_length*, is the depth of nesting.

Recursive calls on *show_node_list* therefore use the following pattern:

```
define node_list_display(#)  $\equiv$ 
  begin append_char("."); show_node_list(#); flush_char;
  end { str_room need not be checked; see show_box below }
```

199. A global variable called *depth_threshold* is used to record the maximum depth of nesting for which *show_node_list* will show information. If we have *depth_threshold* = 0, for example, only the top level information will be given and no sublists will be traversed. Another global variable, called *breadth_max*, tells the maximum number of items to show at each level; *breadth_max* had better be positive, or you won't see anything.

⟨ Global variables 13 ⟩ \equiv
depth_threshold: integer; { maximum nesting depth in box displays }
breadth_max: integer; { maximum number of items shown at the same list level }

200. Now we are ready for *show_node_list* itself. This procedure has been written to be “extra robust” in the sense that it should not crash or get into a loop even if the data structures have been messed up by bugs in the rest of the program. You can safely call its parent routine *show_box(p)* for arbitrary values of *p* when you are debugging TeX. However, in the presence of bad data, the procedure may fetch a *memory_word* whose variant is different from the way it was stored; for example, it might try to read *mem[p].hh* when *mem[p]* contains a scaled integer, if *p* is a pointer that has been clobbered or chosen at random.

```
procedure show_node_list(p : integer); { prints a node list symbolically }
label exit;
var n: integer; { the number of items already printed at this level }
g: real; { a glue ratio, as a floating point number }
begin if cur_length > depth_threshold then
  begin if p > null then print("[]"); { indicate that there's been some truncation }
  return;
end;
n ← 0;
while p > mem_min do
  begin print_ln; print_current_string; { display the nesting history }
  if p > mem_end then { pointer out of range }
    begin print("Bad_link, display_aborted."); return;
  end;
  incr(n);
  if n > breadth_max then { time to stop }
    begin print("etc."); return;
  end;
  ⟨ Display node p 201 ⟩;
  p ← link(p);
end;
exit: end;
```

201. ⟨ Display node p 201 ⟩ ≡
if *is_char_node(p)* then *print_font_and_char(p)*
else case *type(p)* of
hlist_node, vlist_node, unset_node: ⟨ Display box p 202 ⟩;
rule_node: ⟨ Display rule p 205 ⟩;
ins_node: ⟨ Display insertion p 206 ⟩;
whatsit_node: ⟨ Display the whatsit node p 1603 ⟩;
glue_node: ⟨ Display glue p 207 ⟩;
margin_kern_node: begin *print_esc("kern")*; *print_scaled(width(p))*;
 if *subtype(p) = left_side* then *print("left margin")*
 else *print("right margin")*;
 end;
kern_node: ⟨ Display kern p 209 ⟩;
math_node: ⟨ Display math node p 210 ⟩;
ligature_node: ⟨ Display ligature p 211 ⟩;
penalty_node: ⟨ Display penalty p 212 ⟩;
disc_node: ⟨ Display discretionary p 213 ⟩;
mark_node: ⟨ Display mark p 214 ⟩;
adjust_node: ⟨ Display adjustment p 215 ⟩;
⟨ Cases of *show_node_list* that arise in mlists only 866 ⟩
othercases *print("Unknown_node_type!")*
endcases

This code is used in section 200.

202. $\langle \text{Display box } p \text{ 202} \rangle \equiv$

```

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 203 ⟩
else begin ⟨ Display the value of glue_set(p) 204 ⟩;
  if shift_amount(p) ≠ 0 then
    begin print(",\u2028shifted\u2029"); print_scaled(shift_amount(p));
    end;
  if eTeX-ex then ⟨ Display if this box is never to be reversed 1704 ⟩;
  end;
node_list_display(list_ptr(p)); { recursive call }
end

```

This code is used in section 201.

203. $\langle \text{Display special fields of the unset node } p \text{ 203} \rangle \equiv$

```

begin if span_count(p) ≠ min_quarterword then
  begin print("\u2028"); print_int(qo(span_count(p)) + 1); print("\u2028columns");
  end;
if glue_stretch(p) ≠ 0 then
  begin print(",\u2028stretch\u2029"); print_glue(glue_stretch(p), glue_order(p), 0);
  end;
if glue_shrink(p) ≠ 0 then
  begin print(",\u2028shrink\u2029"); print_glue(glue_shrink(p), glue_sign(p), 0);
  end;
end

```

This code is used in section 202.

204. The code will have to change in this place if *glue_ratio* is a structured type instead of an ordinary *real*. Note that this routine should avoid arithmetic errors even if the *glue_set* field holds an arbitrary random value. The following code assumes that a properly formed nonzero *real* number has absolute value 2^{20} or more when it is regarded as an integer; this precaution was adequate to prevent floating point underflow on the author's computer.

$\langle \text{Display the value of glue_set(p) 204} \rangle \equiv$

```

g ← float(glue_set(p));
if (g ≠ float_constant(0)) ∧ (glue_sign(p) ≠ normal) then
  begin print(",\u2028glue\u2028set\u2029");
  if glue_sign(p) = shrinking then print("-\u2028");
  if abs(mem[p + glue_offset].int) < '4000000 then print("?.?")
  else if abs(g) > float_constant(20000) then
    begin if g > float_constant(0) then print_char(">")
    else print("\u2028-");
    print_glue(20000 * unity, glue_order(p), 0);
    end
  else print_glue(round(unity * g), glue_order(p), 0);
end

```

This code is used in section 202.

205. $\langle \text{Display rule } p \text{ 205} \rangle \equiv$

```
begin print_esc("rule("); print_rule_dimen(height(p)); print_char("+"); print_rule_dimen(depth(p));
print(")x"); print_rule_dimen(width(p));
end
```

This code is used in section 201.

206. $\langle \text{Display insertion } p \text{ 206} \rangle \equiv$

```
begin print_esc("insert"); print_int(go(subtype(p))); print(",\nnatural\nsize\n");
print_scaled(height(p)); print(";"); print_spec(split_top_ptr(p), 0); print_char(",");
print_scaled(depth(p)); print(");\nfloat\ncost\n"); print_int(float_cost(p)); node_list_display(ins_ptr(p));
{ recursive call }
end
```

This code is used in section 201.

207. $\langle \text{Display glue } p \text{ 207} \rangle \equiv$

```
if subtype(p) ≥ a_leaders then ⟨ Display leaders p 208 ⟩
else begin print_esc("glue");
if subtype(p) ≠ normal then
begin print_char("(");
if subtype(p) < cond_math_glue then print_skip_param(subtype(p) - 1)
else if subtype(p) = cond_math_glue then print_esc("nonscript")
else print_esc("mskip");
print_char(")");
end;
if subtype(p) ≠ cond_math_glue then
begin print_char("\n");
if subtype(p) < cond_math_glue then print_spec(glue_ptr(p), 0)
else print_spec(glue_ptr(p), "mu");
end;
end
```

This code is used in section 201.

208. $\langle \text{Display leaders } p \text{ 208} \rangle \equiv$

```
begin print_esc("");
if subtype(p) = c_leaders then print_char("c")
else if subtype(p) = x_leaders then print_char("x");
print("leaders\n"); print_spec(glue_ptr(p), 0); node_list_display(leader_ptr(p)); { recursive call }
end
```

This code is used in section 207.

209. An “explicit” kern value is indicated implicitly by an explicit space.

$\langle \text{Display kern } p \text{ 209} \rangle \equiv$

```
if subtype(p) ≠ mu_glue then
begin print_esc("kern");
if subtype(p) ≠ normal then print_char("\n");
print_scaled(width(p));
if subtype(p) = acc_kern then print("\n(for\naccent)\n");
if subtype(p) = auto_kern then print("\n(for\npdfprependkern/\n pdfappendkern)\n");
end
else begin print_esc("mkern"); print_scaled(width(p)); print("mu");
end
```

This code is used in section 201.

210. $\langle \text{Display math node } p \text{ 210} \rangle \equiv$

```

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 201.

211. $\langle \text{Display ligature } p \text{ 211} \rangle \equiv$

```

begin print_font_and_char(lig_char(p)); print(" „(ligature„");
if subtype(p) > 1 then print_char("I");
font_in_short_display ← font(lig_char(p)); short_display(lig_ptr(p));
if odd(subtype(p)) then print_char("I");
print_char(")");
end

```

This code is used in section 201.

212. $\langle \text{Display penalty } p \text{ 212} \rangle \equiv$

```

begin print_esc("penalty„"); print_int(penalty(p));
end

```

This code is used in section 201.

213. The *post-break* list of a discretionary node is indicated by a prefixed ‘|’ instead of the ‘.’ before the *pre-break* list.

$\langle \text{Display discretionary } p \text{ 213} \rangle \equiv$

```

begin print_esc("discretionary");
if replace_count(p) > 0 then
  begin print(" „replacing„"); print_int(replace_count(p));
  end;
node_list_display(pre_break(p)); { recursive call }
append_char("I"); show_node_list(post_break(p)); flush_char; { recursive call }
end

```

This code is used in section 201.

214. $\langle \text{Display mark } p \text{ 214} \rangle \equiv$

```

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 201.

215. $\langle \text{Display adjustment } p \text{ 215} \rangle \equiv$
begin *print_esc("vadjust");*
if *adjust_pre(p) ≠ 0 then print("pre")*;
node_list_display(adjust_ptr(p)); { recursive call}
end

This code is used in section 201.

216. The recursive machinery is started by calling *show_box*.

```
procedure show_box(p : pointer);  

begin { Assign the values depth_threshold  $\leftarrow$  show_box_depth and breadth_max  $\leftarrow$  show_box_breadth 254};  

if breadth_max  $\leq$  0 then breadth_max  $\leftarrow$  5;  

if pool_ptr + depth_threshold  $\geq$  pool_size then depth_threshold  $\leftarrow$  pool_size - pool_ptr - 1;  

    { now there's enough room for prefix string }  

show_node_list(p); { the show starts at p}  

print_ln;  

end;
```

217. Destroying boxes. When we are done with a node list, we are obliged to return it to free storage, including all of its sublists. The recursive procedure *flush_node_list* does this for us.

218. First, however, we shall consider two non-recursive procedures that do simpler tasks. The first of these, *delete_token_ref*, is called when a pointer to a token list's reference count is being removed. This means that the token list should disappear if the reference count was *null*, otherwise the count should be decreased by one.

```
define token_ref_count (#) ≡ info (#) { reference count preceding a token list }
procedure delete_token_ref (p : pointer);
    { p points to the reference count of a token list that is losing one reference }
begin if token_ref_count (p) = null then flush_list (p)
else decr (token_ref_count (p));
end;
```

219. Similarly, *delete_glue_ref* is called when a pointer to a glue specification is being withdrawn.

```
define fast_delete_glue_ref (#) ≡
begin if glue_ref_count (#) = null then free_node (#, glue_spec_size)
else decr (glue_ref_count (#));
end

procedure delete_glue_ref (p : pointer); { p points to a glue specification }
fast_delete_glue_ref (p);
```

220. Now we are ready to delete any node list, recursively. In practice, the nodes deleted are usually charnodes (about 2/3 of the time), and they are glue nodes in about half of the remaining cases.

```

procedure flush_node_list(p : pointer); { erase list of nodes starting at p }
label done; { go here when node p has been freed }
var q: pointer; { successor to node p }
begin while p ≠ null do
begin q ← link(p);
if is_char_node(p) then free_avail(p)
else begin case type(p) of
hlist_node, vlist_node, unset_node: begin flush_node_list(list_ptr(p)); free_node(p, box_node_size);
    goto done;
    end;
rule_node: begin free_node(p, rule_node_size); goto done;
    end;
ins_node: begin flush_node_list(ins_ptr(p)); delete_glue_ref(split_top_ptr(p));
    free_node(p, ins_node_size); goto done;
    end;
whatsit_node: { Wipe out the whatsit node p and goto done 1605 };
glue_node: begin fast_delete_glue_ref(glue_ptr(p));
    if leader_ptr(p) ≠ null then flush_node_list(leader_ptr(p));
    end;
kern_node, math_node, penalty_node: do_nothing;
margin_kern_node: begin free_avail(margin_char(p)); free_node(p, margin_kern_node_size);
    goto done;
    end;
ligature_node: flush_node_list(lig_ptr(p));
mark_node: delete_token_ref(mark_ptr(p));
disc_node: begin flush_node_list(pre_break(p)); flush_node_list(post_break(p));
    end;
adjust_node: flush_node_list(adjust_ptr(p));
{ Cases of flush_node_list that arise in mlists only 874 }
othercases confusion("flushing")
endcases;
free_node(p, small_node_size);
done: end;
p ← q;
end;
end;
```

221. Copying boxes. Another recursive operation that acts on boxes is sometimes needed: The procedure *copy_node_list* returns a pointer to another node list that has the same structure and meaning as the original. Note that since glue specifications and token lists have reference counts, we need not make copies of them. Reference counts can never get too large to fit in a halfword, since each pointer to a node is in a different memory address, and the total number of memory addresses fits in a halfword.

(Well, there actually are also references from outside *mem*; if the *save_stack* is made arbitrarily large, it would theoretically be possible to break TeX by overflowing a reference count. But who would want to do that?)

```
define add_token_ref (#) ≡ incr(token_ref_count(#)) { new reference to a token list }
define add_glue_ref (#) ≡ incr(glue_ref_count(#)) { new reference to a glue spec }
```

222. The copying procedure copies words en masse without bothering to look at their individual fields. If the node format changes—for example, if the size is altered, or if some link field is moved to another relative position—then this code may need to be changed too.

```
function copy_node_list(p : pointer): pointer;
    { makes a duplicate of the node list that starts at p and returns a pointer to the new list }
    var h: pointer; { temporary head of copied list }
    q: pointer; { previous position in new list }
    r: pointer; { current node being fabricated for new list }
    words: 0 .. 5; { number of words remaining to be copied }
begin h ← get_avail; q ← h;
while p ≠ null do
    begin ⟨ Make a copy of node p in node r 223 ⟩;
        link(q) ← r; q ← r; p ← link(p);
    end;
link(q) ← null; q ← link(h); free_avail(h); copy_node_list ← q;
end;
```

223. ⟨ Make a copy of node *p* in node *r* 223 ⟩ ≡
words ← 1; { this setting occurs in more branches than any other }
if *is_char_node*(*p*) then *r* ← *get_avail*
else ⟨ Case statement to copy different types and set *words* to the number of initial words not yet
copied 224 ⟩;
while *words* > 0 do
begin *decr*(*words*); *mem*[*r* + *words*] ← *mem*[*p* + *words*];
end

This code is used in section 222.

224. { Case statement to copy different types and set *words* to the number of initial words not yet copied 224 } \equiv

```

case type(p) of
  hlist_node, vlist_node, unset_node: begin r  $\leftarrow$  get_node(box_node_size); mem[r + 6]  $\leftarrow$  mem[p + 6];
    mem[r + 5]  $\leftarrow$  mem[p + 5]; { copy the last two words }
    list_ptr(r)  $\leftarrow$  copy_node_list(list_ptr(p)); { this affects mem[r + 5] }
    words  $\leftarrow$  5;
  end;
  rule_node: begin r  $\leftarrow$  get_node(rule_node_size); words  $\leftarrow$  rule_node_size;
  end;
  ins_node: begin r  $\leftarrow$  get_node(ins_node_size); mem[r + 4]  $\leftarrow$  mem[p + 4]; add_glue_ref(split_top_ptr(p));
    ins_ptr(r)  $\leftarrow$  copy_node_list(ins_ptr(p)); { this affects mem[r + 4] }
    words  $\leftarrow$  ins_node_size - 1;
  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 1604 };
  glue_node: begin r  $\leftarrow$  get_node(small_node_size); add_glue_ref(glue_ptr(p)); glue_ptr(r)  $\leftarrow$  glue_ptr(p);
    leader_ptr(r)  $\leftarrow$  copy_node_list(leader_ptr(p));
  end;
  kern_node, math_node, penalty_node: begin r  $\leftarrow$  get_node(small_node_size); words  $\leftarrow$  small_node_size;
  end;
  margin_kern_node: begin r  $\leftarrow$  get_node(margin_kern_node_size); fast_get_avail(margin_char(r));
    font(margin_char(r))  $\leftarrow$  font(margin_char(p));
    character(margin_char(r))  $\leftarrow$  character(margin_char(p)); words  $\leftarrow$  small_node_size;
  end;
  ligature_node: begin r  $\leftarrow$  get_node(small_node_size); mem[lig_char(r)]  $\leftarrow$  mem[lig_char(p)];
    { copy font and character }
    lig_ptr(r)  $\leftarrow$  copy_node_list(lig_ptr(p));
  end;
  disc_node: begin r  $\leftarrow$  get_node(small_node_size); pre_break(r)  $\leftarrow$  copy_node_list(pre_break(p));
    post_break(r)  $\leftarrow$  copy_node_list(post_break(p));
  end;
  mark_node: begin r  $\leftarrow$  get_node(small_node_size); add_token_ref(mark_ptr(p));
    words  $\leftarrow$  small_node_size;
  end;
  adjust_node: begin r  $\leftarrow$  get_node(small_node_size); adjust_ptr(r)  $\leftarrow$  copy_node_list(adjust_ptr(p));
  end; { words = 1 = small_node_size - 1 }
  othercases confusion("copying")
  endcases

```

This code is used in section 223.

225. The command codes. Before we can go any further, we need to define symbolic names for the internal code numbers that represent the various commands obeyed by TeX. These codes are somewhat arbitrary, but not completely so. For example, the command codes for character types are fixed by the language, since a user says, e.g., ‘\catcode `\\$ = 3’ to make \$ a math delimiter, and the command code *math_shift* is equal to 3. Some other codes have been made adjacent so that *case* statements in the program need not consider cases that are widely spaced, or so that *case* statements can be replaced by *if* statements.

At any rate, here is the list, for future reference. First come the “catcode” commands, several of which share their numeric codes with ordinary commands when the catcode cannot emerge from TeX’s scanning routine.

```
define escape = 0 { escape delimiter (called \ in The TeXbook) }
define relax = 0 { do nothing ( \relax ) }
define left_brace = 1 { beginning of a group ( { ) }
define right_brace = 2 { ending of a group ( } ) }
define math_shift = 3 { mathematics shift character ( $ ) }
define tab_mark = 4 { alignment delimiter ( &, \span ) }
define car_ret = 5 { end of line ( carriage_return, \cr, \crcr ) }
define out_param = 5 { output a macro parameter }
define mac_param = 6 { macro parameter symbol ( # ) }
define sup_mark = 7 { superscript ( ^ ) }
define sub_mark = 8 { subscript ( _ ) }
define ignore = 9 { characters to ignore ( ^^@ ) }
define endv = 9 { end of  $\langle v_j \rangle$  list in alignment template }
define spacer = 10 { characters equivalent to blank space ( \u ) }
define letter = 11 { characters regarded as letters ( A..Z, a..z ) }
define other_char = 12 { none of the special character types }
define active_char = 13 { characters that invoke macros ( ~ ) }
define par_end = 13 { end of paragraph ( \par ) }
define match = 13 { match a macro parameter }
define comment = 14 { characters that introduce comments ( % ) }
define end_match = 14 { end of parameters to macro }
define stop = 14 { end of job ( \end, \dump ) }
define invalid_char = 15 { characters that shouldn’t appear ( ^^? ) }
define delim_num = 15 { specify delimiter numerically ( \delimiter ) }
define max_char_code = 15 { largest catcode for individual characters }
```

226. 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, \splittdiscards ) }
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 ( \v ) }
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 ( \u ) }
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 }

```

227. 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 letterspace_font = 101 { letterspace a font ( \letterspacefont ) }
define pdf_copy_font = 102 { create a new font instance ( \pdfcopyfont ) }
define max_command = 102 { the largest command code seen at big_switch }

```

228. The remaining command codes are extra special, since they cannot get through T_EX'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 }
```

229. The semantic nest. TeX is typically in the midst of building many lists at once. For example, when a math formula is being processed, TeX is in math mode and working on an mlist; this formula has temporarily interrupted TeX from being in horizontal mode and building the hlist of a paragraph; and this paragraph has temporarily interrupted TeX from being in vertical mode and building the vlist for the next page of a document. Similarly, when a \vbox occurs inside of an \hbox, TeX is temporarily interrupted from working in restricted horizontal mode, and it enters internal vertical mode. The “semantic nest” is a stack that keeps track of what lists and modes are currently suspended.

At each level of processing we are in one of six modes:

vmode stands for vertical mode (the page builder);
hmode stands for horizontal mode (the paragraph builder);
mmode stands for displayed formula mode;
 $-vmode$ stands for internal vertical mode (e.g., in a \vbox);
 $-hmode$ stands for restricted horizontal mode (e.g., in an \hbox);
 $-mmode$ stands for math formula mode (not displayed).

The mode is temporarily set to zero while processing \write texts.

Numeric values are assigned to *vmode*, *hmode*, and *mmode* so that TeX’s “big semantic switch” can select the appropriate thing to do by computing the value $abs(mode) + cur_cmd$, where *mode* is the current mode and *cur_cmd* is the current command code.

```

define vmode = 1 { vertical mode }
define hmode = vmode + max_command + 1 { horizontal mode }
define mmode = hmode + max_command + 1 { math mode }

procedure print_mode(m : integer); { prints the mode represented by m }
begin if m > 0 then
  case m div (max_command + 1) of
    0: print("vertical");
    1: print("horizontal");
    2: print("display\u{math}");
  end
  else if m = 0 then print("no")
  else case (-m) div (max_command + 1) of
    0: print("internal\u{vertical}");
    1: print("restricted\u{horizontal}");
    2: print("math");
  end;
  print(" \u{mode}");
end;

```

230. 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 $\leq -1000\text{pt}$ 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 { magic dimension value to mean 'ignore me' }
⟨ 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;
```

231. `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 node 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_node ≡ 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 }`
`save_tail: pointer; { save tail so we can examine whether we have an auto kern before a glue }`
`prev_tail: pointer; { value of tail before the last call to tail_append }`

232. Here is a common way to make the current list grow:

```
define tail_append(#+) ≡
    begin prev_tail ← tail; link(tail) ← #; tail ← link(tail);
    end
define insert_before_tail(#+) ≡
    begin link(prev_tail) ← #; link(#+) ← tail; prev_tail ← #;
    end
```

233. 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; save_tail ← null; prev_depth ← ignore_depth; mode_line ← 0; prev_graf ← 0;`
`shown_mode ← 0; { Start a new current page 1168 };`

234. 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;
```

235. Conversely, when T_EX is finished on the current level, the former state is restored by calling *pop_nest*. This routine will never be called at the lowest semantic level, nor will it be called unless *head* is a node that should be returned to free memory.

```
procedure pop_nest; { leave a semantic level, re-enter the old }
begin free_avail(head); decr(nest_ptr); cur_list ← nest[nest_ptr];
end;
```

236. Here is a procedure that displays what T_EX is working on, at all levels.

```
procedure print_totals; forward;
procedure show_activities;
var p: 0 .. nest_size; { index into nest }
m: -mmode .. mmode; { mode }
a: memory_word; { auxiliary }
q, r: pointer; { for showing the current page }
t: integer; { ditto }
begin nest[nest_ptr] ← cur_list; { put the top level into the array }
print_nl(""); print_ln;
for p ← nest_ptr downto 0 do
begin m ← nest[p].mode_field; a ← nest[p].aux_field; print_nl("###");
print("entered at line"); print_int(abs(nest[p].ml_field));
if m = hmode then
if nest[p].pg_field ≠ '40600000 then
begin print("(\language"); print_int(nest[p].pg_field mod '200000); print(":hyphenmin");
print_int((nest[p].pg_field div '20000000) mod '100); print_char(")");
end;
if nest[p].ml_field < 0 then print("\outputroutine");
if p = 0 then
begin ⟨Show the status of the current page 1163⟩;
if link(contrib_head) ≠ null then print_nl("### recent contributions:");
end;
show_box(link(nest[p].head_field)); ⟨Show the auxiliary field, a 237⟩;
end;
end;
```

237. \langle Show the auxiliary field, a [237](#) $\rangle \equiv$

```
case abs(m) div (max_command + 1) of
  0: begin print_nl("prevdepth\u");
    if a.sc ≤ pdf_ignored_dimen then print("ignored")
    else print_scaled(a.sc);
    if nest[p].pg_field ≠ 0 then
      begin print(",\u prevgraf\u"); print_int(nest[p].pg_field); print("\u line");
        if nest[p].pg_field ≠ 1 then print_char("s");
      end;
    end;
  end;
  1: begin print_nl("spacefactor\u"); print_int(a.hh.lh);
    if m > 0 then if a.hh.rh > 0 then
      begin print(",\u current\u language\u"); print_int(a.hh.rh); end;
    end;
  end;
  2: if a.int ≠ null then
    begin print("this\u will\u begin\u denominator\u of :"); show_box(a.int); end;
end { there are no other cases }
```

This code is used in section [236](#).

238. The table of equivalents. Now that we have studied the data structures for T_EX's semantic routines, we ought to consider the data structures used by its syntactic routines. In other words, our next concern will be the tables that T_EX looks at when it is scanning what the user has written.

The biggest and most important such table is called *eqtb*. It holds the current "equivalents" of things; i.e., it explains what things mean or what their current values are, for all quantities that are subject to the nesting structure provided by T_EX's grouping mechanism. There are six parts to *eqtb*:

- 1) *eqtb[active_base .. (hash_base - 1)]* holds the current equivalents of single-character control sequences.
- 2) *eqtb[hash_base .. (glue_base - 1)]* holds the current equivalents of multiletter control sequences.
- 3) *eqtb[glue_base .. (local_base - 1)]* holds the current equivalents of glue parameters like the current baselineskip.
- 4) *eqtb[local_base .. (int_base - 1)]* holds the current equivalents of local halfword quantities like the current box registers, the current "catcodes," the current font, and a pointer to the current paragraph shape.
- 5) *eqtb[int_base .. (dimen_base - 1)]* holds the current equivalents of fullword integer parameters like the current hyphenation penalty.
- 6) *eqtb[dimen_base .. eqtb_size]* holds the current equivalents of fullword dimension parameters like the current hsize or amount of hanging indentation.

Note that, for example, the current amount of baselineskip glue is determined by the setting of a particular location in region 3 of *eqtb*, while the current meaning of the control sequence '\baselineskip' (which might have been changed by \def or \let) appears in region 2.

239. Each entry in *eqtb* is a *memory_word*. Most of these words are of type *two_halves*, and subdivided into three fields:

- 1) The *eq_level* (a quarterword) is the level of grouping at which this equivalent was defined. If the level is *level_zero*, the equivalent has never been defined; *level_one* refers to the outer level (outside of all groups), and this level is also used for global definitions that never go away. Higher levels are for equivalents that will disappear at the end of their group.
- 2) The *eq_type* (another quarterword) specifies what kind of entry this is. There are many types, since each T_EX primitive like \hbox, \def, etc., has its own special code. The list of command codes above includes all possible settings of the *eq_type* field.
- 3) The *equiv* (a halfword) is the current equivalent value. This may be a font number, a pointer into *mem*, or a variety of other things.

```
define eq_level_field(#) ≡ #.hh.b1
define eq_type_field(#) ≡ #.hh.b0
define equiv_field(#) ≡ #.hh.rh
define eq_level(#) ≡ eq_level_field(eqtb[#]) { level of definition }
define eq_type(#) ≡ eq_type_field(eqtb[#]) { command code for equivalent }
define equiv(#) ≡ equiv_field(eqtb[#]) { equivalent value }
define level_zero = min_quarterword { level for undefined quantities }
define level_one = level_zero + 1 { outermost level for defined quantities }
```

240. Many locations in *eqtb* have symbolic names. The purpose of the next paragraphs is to define these names, and to set up the initial values of the equivalents.

In the first region we have 256 equivalents for “active characters” that act as control sequences, followed by 256 equivalents for single-character control sequences.

Then comes region 2, which corresponds to the hash table that we will define later. The maximum address in this region is used for a dummy control sequence that is perpetually undefined. There also are several locations for control sequences that are perpetually defined (since they are used in error recovery).

```

define active_base = 1 { beginning of region 1, for active character equivalents }
define single_base = active_base + 256 { equivalents of one-character control sequences }
define null_cs = single_base + 256 { equivalent of \csname\endcsname }
define hash_base = null_cs + 1 { beginning of region 2, for the hash table }
define frozen_control_sequence = hash_base + hash_size { for error recovery }
define frozen_protection = frozen_control_sequence { inaccessible but definable }
define frozen_cr = frozen_control_sequence + 1 { permanent '\cr' }
define frozen_end_group = frozen_control_sequence + 2 { permanent '\endgroup' }
define frozen_right = frozen_control_sequence + 3 { permanent '\right' }
define frozen_fi = frozen_control_sequence + 4 { permanent '\fi' }
define frozen_end_template = frozen_control_sequence + 5 { permanent '\endtemplate' }
define frozen_endv = frozen_control_sequence + 6 { second permanent '\endtemplate' }
define frozen_relax = frozen_control_sequence + 7 { permanent '\relax' }
define end_write = frozen_control_sequence + 8 { permanent '\endwrite' }
define frozen_dont_expand = frozen_control_sequence + 9 { permanent '\notexpanded:' }
define prim_size = 2100 { maximum number of primitives }
define frozen_null_font = frozen_control_sequence + 10 { permanent '\nullfont' }
define frozen_primitive = frozen_control_sequence + 11 { permanent '\pdfprimitive' }
define prim_eqtb_base = frozen_primitive + 1
define font_id_base = frozen_null_font - font_base { begins table of 257 permanent font identifiers }
define undefined_control_sequence = frozen_null_font + 257 { dummy location }
define glue_base = undefined_control_sequence + 1 { beginning of region 3 }

{ Initialize table entries (done by INITEX only) 182 } +≡
eq_type(undefined_control_sequence) ← undefined_cs; equiv(undefined_control_sequence) ← null;
eq_level(undefined_control_sequence) ← level_zero;
for k ← active_base to undefined_control_sequence - 1 do eqtb[k] ← eqtb[undefined_control_sequence];

```

241. Here is a routine that displays the current meaning of an *eqtb* entry in region 1 or 2. (Similar routines for the other regions will appear below.)

```

{ Show equivalent n, in region 1 or 2 241 } ≡
begin sprint_cs(n); print_char("="); print_cmd_chr(eq_type(n), equiv(n));
if eq_type(n) ≥ call then
  begin print_char(":"); show_token_list(link(equiv(n)), null, 32);
  end;
end

```

This code is used in section 270.

242. Region 3 of *eqtb* contains the 256 \skip registers, as well as the glue parameters defined here. It is important that the “muskip” parameters have larger numbers than the others.

```

define line_skip_code = 0 { interline glue if baseline_skip is infeasible }
define baseline_skip_code = 1 { desired glue between baselines }
define par_skip_code = 2 { extra glue just above a paragraph }
define above_display_skip_code = 3 { extra glue just above displayed math }
define below_display_skip_code = 4 { extra glue just below displayed math }
define above_display_short_skip_code = 5 { glue above displayed math following short lines }
define below_display_short_skip_code = 6 { glue below displayed math following short lines }
define left_skip_code = 7 { glue at left of justified lines }
define right_skip_code = 8 { glue at right of justified lines }
define top_skip_code = 9 { glue at top of main pages }
define split_top_skip_code = 10 { glue at top of split pages }
define tab_skip_code = 11 { glue between aligned entries }
define space_skip_code = 12 { glue between words (if not zero_glue) }
define xspace_skip_code = 13 { glue after sentences (if not zero_glue) }
define par_fill_skip_code = 14 { glue on last line of paragraph }
define thin_mu_skip_code = 15 { thin space in math formula }
define med_mu_skip_code = 16 { medium space in math formula }
define thick_mu_skip_code = 17 { thick space in math formula }
define glue_pars = 18 { total number of glue parameters }
define skip_base = glue_base + glue_pars { table of 256 “skip” registers }
define mu_skip_base = skip_base + 256 { table of 256 “muskip” registers }
define local_base = mu_skip_base + 256 { beginning of region 4 }

define skip(#) $\equiv$ equiv(skip_base + #) { mem location of glue specification }
define mu_skip(#) $\equiv$ equiv(mu_skip_base + #) { mem location of math glue spec }
define glue_par(#) $\equiv$ equiv(glue_base + #) { mem location of glue specification }
define line_skip $\equiv$ glue_par(line_skip_code)
define baseline_skip $\equiv$ glue_par(baseline_skip_code)
define par_skip $\equiv$ glue_par(par_skip_code)
define above_display_skip $\equiv$ glue_par(above_display_skip_code)
define below_display_skip $\equiv$ glue_par(below_display_skip_code)
define above_display_short_skip $\equiv$ glue_par(above_display_short_skip_code)
define below_display_short_skip $\equiv$ glue_par(below_display_short_skip_code)
define left_skip $\equiv$ glue_par(left_skip_code)
define right_skip $\equiv$ glue_par(right_skip_code)
define top_skip $\equiv$ glue_par(top_skip_code)
define split_top_skip $\equiv$ glue_par(split_top_skip_code)
define tab_skip $\equiv$ glue_par(tab_skip_code)
define space_skip $\equiv$ glue_par(space_skip_code)
define xspace_skip $\equiv$ glue_par(xspace_skip_code)
define par_fill_skip $\equiv$ glue_par(par_fill_skip_code)
define thin_mu_skip $\equiv$ glue_par(thin_mu_skip_code)
define med_mu_skip $\equiv$ glue_par(med_mu_skip_code)
define thick_mu_skip $\equiv$ glue_par(thick_mu_skip_code)

<Current mem equivalent of glue parameter number n 242> $\equiv$ 
glue_par(n)

```

This code is used in sections 170 and 172.

243. Sometimes we need to convert TeX's internal code numbers into symbolic form. The *print_skip_param* routine gives the symbolic name of a glue parameter.

⟨ Declare the procedure called *print_skip_param* 243 ⟩ ≡

```
procedure print_skip_param(n : integer);
begin case n of
line_skip_code: print_esc("lineskip");
baseline_skip_code: print_esc("baselineskip");
par_skip_code: print_esc("parskip");
above_display_skip_code: print_esc("abovedisplayskip");
below_display_skip_code: print_esc("belowdisplayskip");
above_display_short_skip_code: print_esc("abovedisplayshortskip");
below_display_short_skip_code: print_esc("belowdisplayshortskip");
left_skip_code: print_esc("leftskip");
right_skip_code: print_esc("rightskip");
top_skip_code: print_esc("topskip");
split_top_skip_code: print_esc("splittopskip");
tab_skip_code: print_esc("tabskip");
space_skip_code: print_esc("spaceskip");
xspace_skip_code: print_esc("xspaceskip");
par_fill_skip_code: print_esc("parfillskip");
thin_mu_skip_code: print_esc("thinmuskip");
med_mu_skip_code: print_esc("medmuskip");
thick_mu_skip_code: print_esc("thickmuskip");
othercases print(" [unknown glue parameter!] ")
endcases;
end;
```

This code is used in section 197.

244. The symbolic names for glue parameters are put into TeX's hash table by using the routine called *primitive*, defined below. Let us enter them now, so that we don't have to list all those parameter names anywhere else.

(Put each of TeX's primitives into the hash table 244) \equiv

```
primitive("lineskip", assign_glue, glue_base + line_skip_code);
primitive("baselineskip", assign_glue, glue_base + baseline_skip_code);
primitive("parskip", assign_glue, glue_base + par_skip_code);
primitive("abovedisplayskip", assign_glue, glue_base + above_display_skip_code);
primitive("belowdisplayskip", assign_glue, glue_base + below_display_skip_code);
primitive("abovedisplayshortskip", assign_glue, glue_base + above_display_short_skip_code);
primitive("belowdisplayshortskip", assign_glue, glue_base + below_display_short_skip_code);
primitive("leftskip", assign_glue, glue_base + left_skip_code);
primitive("rightskip", assign_glue, glue_base + right_skip_code);
primitive("topskip", assign_glue, glue_base + top_skip_code);
primitive("splittopskip", assign_glue, glue_base + split_top_skip_code);
primitive("tabskip", assign_glue, glue_base + tab_skip_code);
primitive("spaceskip", assign_glue, glue_base + space_skip_code);
primitive("xspaceskip", assign_glue, glue_base + xspace_skip_code);
primitive("parfillskip", assign_glue, glue_base + par_fill_skip_code);
primitive("thinmuskip", assign_mu_glue, glue_base + thin_mu_skip_code);
primitive("medmuskip", assign_mu_glue, glue_base + med_mu_skip_code);
primitive("thickmuskip", assign_mu_glue, glue_base + thick_mu_skip_code);
```

See also sections 248, 256, 266, 287, 356, 402, 410, 437, 442, 494, 513, 517, 579, 956, 1160, 1230, 1236, 1249, 1266, 1285, 1292, 1319, 1334, 1347, 1356, 1366, 1386, 1397, 1400, 1408, 1428, 1432, 1440, 1450, 1455, 1464, 1469, and 1524.

This code is used in section 1516.

245. *(Cases of print_cmd_chr for symbolic printing of primitives 245) \equiv*

```
assign_glue, assign_mu_glue: if chr_code < skip_base then print_skip_param(chr_code - glue_base)
else if chr_code < mu_skip_base then
begin print_esc("skip"); print_int(chr_code - skip_base);
end
else begin print_esc("muskip"); print_int(chr_code - mu_skip_base);
end;
```

See also sections 249, 257, 267, 288, 357, 403, 411, 438, 443, 495, 514, 518, 957, 1161, 1231, 1237, 1250, 1267, 1286, 1293, 1321, 1335, 1348, 1357, 1367, 1387, 1398, 1401, 1409, 1429, 1433, 1439, 1441, 1451, 1456, 1465, 1470, 1473, and 1526.

This code is used in section 320.

246. All glue parameters and registers are initially '0pt plus0pt minus0pt'.

(Initialize table entries (done by INITEX only) 182) $+ \equiv$

```
equiv(glue_base)  $\leftarrow$  zero_glue; eq_level(glue_base)  $\leftarrow$  level_one; eq_type(glue_base)  $\leftarrow$  glue_ref;
for k  $\leftarrow$  glue_base + 1 to local_base - 1 do eqtb[k]  $\leftarrow$  eqtb[glue_base];
glue_ref_count(zero_glue)  $\leftarrow$  glue_ref_count(zero_glue) + local_base - glue_base;
```

247. \langle Show equivalent n , in region 3 247 $\rangle \equiv$

```
if  $n < skip\_base$  then
  begin print_skip_param( $n - glue\_base$ ); print_char("=");
  if  $n < glue\_base + thin\_mu\_skip\_code$  then print_spec(equiv( $n$ ), "pt")
  else print_spec(equiv( $n$ ), "mu");
  end
else if  $n < mu\_skip\_base$  then
  begin print_esc("skip"); print_int( $n - skip\_base$ ); print_char("=");
  print_spec(equiv( $n$ ), "pt");
  end
else begin print_esc("muskip"); print_int( $n - mu\_skip\_base$ ); print_char("=");
  print_spec(equiv( $n$ ), "mu");
  end
```

This code is used in section 270.

248. 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 pdftex_first_loc = tex_toks { base for pdfTeX's token list parameters }
define pdf_pages_attr_loc = pdftex_first_loc + 0 { points to token list for \pdfpagesattr }
define pdf_page_attr_loc = pdftex_first_loc + 1 { points to token list for \pdfpageattr }
define pdf_page_resources_loc = pdftex_first_loc + 2 { points to token list for \pdfpageresources }
define pdf_pk_mode_loc = pdftex_first_loc + 3 { points to token list for \pdfpkmode }
define pdf_toks = pdftex_first_loc + 4 { end of pdfTeX's token list parameters }

define etex_toks_base = pdf_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 pdf_pages_attr ≡ equiv(pdf_pages_attr_loc)

```

```

define pdf_page_attr ≡ equiv(pdf_page_attr_loc)
define pdf_page_resources ≡ equiv(pdf_page_resources_loc)
define pdf_pk_mode ≡ equiv(pdf_pk_mode_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 244 ⟩ +≡
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);
primitive("pdfpagesattr", assign_toks, pdf_pages_attr_loc);
primitive("pdfpageattr", assign_toks, pdf_page_attr_loc);
primitive("pdfpageresources", assign_toks, pdf_page_resources_loc);
primitive("pdfpkmode", assign_toks, pdf_pk_mode_loc);

```

249. ⟨ Cases of print_cmd_chr for symbolic printing of primitives 245 ⟩ +≡

```

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 1658 ⟩
  pdf_pages_attr_loc: print_esc("pdfpagesattr");
  pdf_page_attr_loc: print_esc("pdfpageattr");
  pdf_page_resources_loc: print_esc("pdfpageresources");
  pdf_pk_mode_loc: print_esc("pdfpkmode");
  othercases print_esc("errhelp");
endcases;

```

250. 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) 182 ⟩ +≡
  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;

```

251. \langle Show equivalent n , in region 4 [251](#) $\rangle \equiv$

```

if ( $n = par\_shape\_loc$ )  $\vee ((n \geq etex\_pen\_base) \wedge (n < etex\_pens))$  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  $equiv(n) \neq null$  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  $equiv(n) \neq null$  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  $\leftarrow 0$ ; breadth_max  $\leftarrow 1$ ; show_node_list( $equiv(n)$ );
    end;
  end
else if  $n < cat\_code\_base$  then ⟨ Show the font identifier in  $eqtb[n]$  252 ⟩
  else ⟨ Show the halfword code in  $eqtb[n]$  253 ⟩

```

This code is used in section [270](#).

252. ⟨ Show the font identifier in $eqtb[n]$ [252](#) ⟩ \equiv

```

begin if  $n = cur\_font\_loc$  then print("current↳font")
else if  $n < math\_font\_base + 16$  then
  begin print_esc("textfont"); print_int( $n - math\_font\_base$ );
  end
else if  $n < math\_font\_base + 32$  then
  begin print_esc("scriptfont"); print_int( $n - math\_font\_base - 16$ );
  end
else begin print_esc("scriptscriptfont"); print_int( $n - math\_font\_base - 32$ );
  end;
print_char("=");
print_esc(hash[ $font\_id\_base + equiv(n)$ ].rh); { that's  $font\_id\_text(equiv(n))$  }
end

```

This code is used in section [251](#).

253. ⟨ Show the halfword code in $eqtb[n]$ 253 ⟩ ≡
if $n < math_code_base$ then
begin if $n < lc_code_base$ then
begin print_esc("catcode"); print_int($n - cat_code_base$);
end
else if $n < uc_code_base$ then
begin print_esc("lccode"); print_int($n - lc_code_base$);
end
else if $n < sf_code_base$ then
begin print_esc("uccode"); print_int($n - uc_code_base$);
end
else begin print_esc("sfcde"); print_int($n - sf_code_base$);
end;
print_char('='); print_int(equiv(n));
end
else begin print_esc("mathcode"); print_int($n - math_code_base$); print_char('=');
print_int(ho(equiv(n)));
end

This code is used in section 251.

254. 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 held over 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 pdftex_first_integer_code = tex_int_pars { base for pdftEX's integer parameters }
define pdf_output_code = pdftex_first_integer_code + 0 { switch on PDF output if positive }
define pdf_compress_level_code = pdftex_first_integer_code + 1 { compress level of streams }
define pdf_decimal_digits_code = pdftex_first_integer_code + 2
    { digits after the decimal point of numbers }
define pdf_move_chars_code = pdftex_first_integer_code + 3 { move chars 0..31 to higher area if possible }
define pdf_image_resolution_code = pdftex_first_integer_code + 4 { default image resolution }
define pdf_pk_resolution_code = pdftex_first_integer_code + 5 { default resolution of PK font }
define pdf_unique_resname_code = pdftex_first_integer_code + 6 { generate unique names for resources }
define pdf_option_always_use_pdffpagebox_code = pdftex_first_integer_code + 7
    { if the PDF inclusion should always use a specific PDF page box }
define pdf_option_pdf_inclusion_errorlevel_code = pdftex_first_integer_code + 8
    { if the PDF inclusion should treat pdfs newer than pdf_minor_version as an error }
define pdf_major_version_code = pdftex_first_integer_code + 9
    { integer part of the PDF version produced }
define pdf_minor_version_code = pdftex_first_integer_code + 10
    { fractional part of the PDF version produced }
define pdf_force_pagebox_code = pdftex_first_integer_code + 11
    { if the PDF inclusion should always use a specific PDF page box }
define pdf_pagebox_code = pdftex_first_integer_code + 12 { default pagebox to use for PDF inclusion }
define pdf_inclusion_errorlevel_code = pdftex_first_integer_code + 13
    { if the PDF inclusion should treat pdfs newer than pdf_minor_version as an error }
define pdf_gamma_code = pdftex_first_integer_code + 14
define pdf_image_gamma_code = pdftex_first_integer_code + 15
define pdf_image_hicolor_code = pdftex_first_integer_code + 16
define pdf_image_apply_gamma_code = pdftex_first_integer_code + 17
define pdf_adjust_spacing_code = pdftex_first_integer_code + 18 { level of spacing adjusting }
define pdf_protrude_chars_code = pdftex_first_integer_code + 19
    { protrude chars at left/right edge of paragraphs }
define pdf_tracing_fonts_code = pdftex_first_integer_code + 20 { level of font detail in log }
define pdf_objcompresslevel_code = pdftex_first_integer_code + 21 { activate object streams }
define pdf_adjust_interword_glue_code = pdftex_first_integer_code + 22 { adjust interword glue? }
define pdf_prepend_kern_code = pdftex_first_integer_code + 23 { prepend kern before certain characters? }
define pdf_append_kern_code = pdftex_first_integer_code + 24 { append kern before certain characters? }
define pdf_gen_tounicode_code = pdftex_first_integer_code + 25 { generate ToUnicode for fonts? }
define pdf_draftmode_code = pdftex_first_integer_code + 26 { switch on draftmode if positive }
define pdf_inclusion_copy_font_code = pdftex_first_integer_code + 27 { generate ToUnicode for fonts? }
define pdf_suppress_warning_dup_dest_code = pdftex_first_integer_code + 28
    { suppress warning about duplicated destinations }
define pdf_suppress_warning_dup_map_code = pdftex_first_integer_code + 29
    { suppress warning about duplicated map lines }
define pdf_suppress_warning_page_group_code = pdftex_first_integer_code + 30
    { suppress warning about multiple pdfs with page group }

```

```

define pdf_info OMIT_DATE_CODE = pdftex_first_integer_code + 31
    { omit generating CreationDate and ModDate }
define pdf_suppress_ptex_info_code = pdftex_first_integer_code + 32
    { suppress /PTEX.* entries in PDF dictionaries }
define pdf OMIT_CHARSET_CODE = pdftex_first_integer_code + 33 { omit CharSet in Font dict }
define pdf OMIT_INFO_DICT_CODE = pdftex_first_integer_code + 34 { omit Info dict }
define pdf OMIT_PROCSET_CODE = pdftex_first_integer_code + 35 { omit ProcSet in resources dict }
define pdf_PTEX_USE_UNDERSCORE_CODE = pdftex_first_integer_code + 36 { use underscore for PTEX prefix }
define pdf_int_pars = pdftex_first_integer_code + 37 { total number of pdfTeX's integer parameters }
define etex_int_base = pdf_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 pdf_adjust_spacing ≡ int_par(pdf_adjust_spacing_code)
define pdf_protrude_chars ≡ int_par(pdf_protrude_chars_code)
define pdf_tracing_fonts ≡ int_par(pdf_tracing_fonts_code)
define pdf_adjust_interword_glue ≡ int_par(pdf_adjust_interword_glue_code)
define pdf_prepend_kern ≡ int_par(pdf_prepend_kern_code)
define pdf_append_kern ≡ int_par(pdf_append_kern_code)
define pdf_gen_tounicode ≡ int_par(pdf_gen_tounicode_code)
define pdf_output ≡ int_par(pdf_output_code)
define pdf_compress_level ≡ int_par(pdf_compress_level_code)
define pdf_objcompresslevel ≡ int_par(pdf_objcompresslevel_code)
define pdf_decimal_digits ≡ int_par(pdf_decimal_digits_code)
define pdf_move_chars ≡ int_par(pdf_move_chars_code)
define pdf_image_resolution ≡ int_par(pdf_image_resolution_code)
define pdf_pk_resolution ≡ int_par(pdf_pk_resolution_code)
define pdf_unique_resname ≡ int_par(pdf_unique_resname_code)
define pdf_option_always_use_pdffpagebox ≡ int_par(pdf_option_always_use_pdffpagebox_code)
define pdf_option_pdf_inclusion_errorlevel ≡ int_par(pdf_option_pdf_inclusion_errorlevel_code)
define pdf_major_version ≡ int_par(pdf_major_version_code)
define pdf_minor_version ≡ int_par(pdf_minor_version_code)
define pdf_force_pagebox ≡ int_par(pdf_force_pagebox_code)
define pdf_pagebox ≡ int_par(pdf_pagebox_code)

```

```
define pdf_inclusion_errorlevel ≡ int_par(pdf_inclusion_errorlevel_code)
define pdf_gamma ≡ int_par(pdf_gamma_code)
define pdf_image_gamma ≡ int_par(pdf_image_gamma_code)
define pdf_image_hicolor ≡ int_par(pdf_image_hicolor_code)
define pdf_image_apply_gamma ≡ int_par(pdf_image_apply_gamma_code)
define pdf_draftmode ≡ int_par(pdf_draftmode_code)
define pdf_inclusion_copy_font ≡ int_par(pdf_inclusion_copy_font_code)
define pdf_suppress_warning_dup_dest ≡ int_par(pdf_suppress_warning_dup_dest_code)
define pdf_suppress_warning_dup_map ≡ int_par(pdf_suppress_warning_dup_map_code)
define pdf_suppress_warning_page_group ≡ int_par(pdf_suppress_warning_page_group_code)
define pdf_info OMIT_date ≡ int_par(pdf_info OMIT_date_code)
define pdf_suppress_ptex_info ≡ int_par(pdf_suppress_ptex_info_code)
define pdf OMIT_charset ≡ int_par(pdf OMIT_charset_code)
define pdf OMIT_info_dict ≡ int_par(pdf OMIT_info_dict_code)
define pdf OMIT_procset ≡ int_par(pdf OMIT_procset_code)
define pdf_ptex_use_underscore ≡ int_par(pdf_ptex_use_underscore_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 254 ⟩ ≡
depth_threshold ← show_box_depth; breadth_max ← show_box_breadth
```

This code is used in section 216.

255. 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");

pdf_output_code: print_esc("pdfoutput");
pdf_compress_level_code: print_esc("pdfcompresslevel");
pdf_objcompresslevel_code: print_esc("pdfobjcompresslevel");
pdf_decimal_digits_code: print_esc("pdfdecimaldigits");
pdf_move_chars_code: print_esc("pdfmovechars");
pdf_image_resolution_code: print_esc("pdfimageresolution");
pdf_pk_resolution_code: print_esc("pdfpkresolution");
pdf_unique_resname_code: print_esc("pdfuniqueresname");
pdf_option_always_use_pdfpagebox_code: print_esc("pdfoptionalwaysusepdfpagebox");
pdf_option_pdf_inclusion_errorlevel_code: print_esc("pdfoptionpdfinclusionerrorlevel");
pdf_major_version_code: print_esc("pdfmajorversion");
pdf_minor_version_code: print_esc("pdfminorversion");
pdf_force_pagebox_code: print_esc("pdfforcepagebox");
pdf_pagebox_code: print_esc("pdfpagebox");
pdf_inclusion_errorlevel_code: print_esc("pdfinclusionerrorlevel");
pdf_gamma_code: print_esc("pdfgamma");
pdf_image_gamma_code: print_esc("pdfimagegamma");
pdf_image_hicolor_code: print_esc("pdfimagehicolor");
pdf_image_apply_gamma_code: print_esc("pdfimageapplygamma");
pdf_adjust_spacing_code: print_esc("pdfadjustspacing");
pdf_protrude_chars_code: print_esc("pdfprotrudechars");
pdf_tracing_fonts_code: print_esc("pdftracingfonts");
pdf_adjust_interword_glue_code: print_esc("pdfadjustinterwordglue");
pdf_prepend_kern_code: print_esc("pdfprependkern");
pdf_append_kern_code: print_esc("pdfappendkern");
pdf_gen_tounicode_code: print_esc("pdfgentounicode");
pdf_draftmode_code: print_esc("pdfdraftmode");
pdf_inclusion_copy_font_code: print_esc("pdfinclusioncopyfonts");
pdf_suppress_warning_dup_dest_code: print_esc("pdfsuppresswarningdupdest");
pdf_suppress_warning_dup_map_code: print_esc("pdfsuppresswarningdupmap");
pdf_suppress_warning_page_group_code: print_esc("pdfsuppresswarningpagegroup");
pdf_info OMIT_date_code: print_esc("pdfinfoomitdate");
pdf_suppress_ptex_info_code: print_esc("pdfsuppressptexinfo");
pdf OMIT_charset_code: print_esc("pdfomitcharset");
pdf OMIT_info_dict_code: print_esc("pdfomitinfodict");
pdf OMIT_procset_code: print_esc("pdfomitprocset");
pdf_ptex_use_underscore_code: print_esc("pdfptexuseunderscore");
  {Cases for print-param 1659}
othercases print(" [unknown\integer\parameter!] ")
endcases;
end;

```

256. The integer parameter names must be entered into the hash table.

```
< Put each of TeX's primitives into the hash table 244 > +≡
primitive("pretolerance", assign_int, int_base + pretolerance_code);
primitive("tolerance", assign_int, int_base + tolerance_code);
primitive("linepenalty", assign_int, int_base + line_penalty_code);
primitive("hyphenpenalty", assign_int, int_base + hyphen_penalty_code);
primitive("exhyphenpenalty", assign_int, int_base + ex_hyphen_penalty_code);
primitive("clubpenalty", assign_int, int_base + club_penalty_code);
primitive("widowpenalty", assign_int, int_base + widow_penalty_code);
primitive("displaywidowpenalty", assign_int, int_base + display_widow_penalty_code);
primitive("brokenpenalty", assign_int, int_base + broken_penalty_code);
primitive("binoppenalty", assign_int, int_base + bin_op_penalty_code);
primitive("relpenalty", assign_int, int_base + rel_penalty_code);
primitive("predisplaypenalty", assign_int, int_base + pre_display_penalty_code);
primitive("postdisplaypenalty", assign_int, int_base + post_display_penalty_code);
primitive("interlinepenalty", assign_int, int_base + inter_line_penalty_code);
primitive("doublehyphendemerits", assign_int, int_base + double_hyphen_demerits_code);
primitive("finalhyphendemerits", assign_int, int_base + final_hyphen_demerits_code);
primitive("adjdemerits", assign_int, int_base + adj_demerits_code);
primitive("mag", assign_int, int_base + mag_code);
primitive("delimiterfactor", assign_int, int_base + delimiter_factor_code);
primitive("looseness", assign_int, int_base + looseness_code);
primitive("time", assign_int, int_base + time_code);
primitive("day", assign_int, int_base + day_code);
primitive("month", assign_int, int_base + month_code);
primitive("year", assign_int, int_base + year_code);
primitive("showboxbreadth", assign_int, int_base + show_box_breadth_code);
primitive("showboxdepth", assign_int, int_base + show_box_depth_code);
primitive("hbadness", assign_int, int_base + hbadness_code);
primitive("vbadness", assign_int, int_base + vbadness_code);
primitive("pausing", assign_int, int_base + pausing_code);
primitive("tracingonline", assign_int, int_base + tracing_online_code);
primitive("tracingmacros", assign_int, int_base + tracing_macros_code);
primitive("tracingstats", assign_int, int_base + tracing_stats_code);
primitive("tracingparagraphs", assign_int, int_base + tracing_paragraphs_code);
primitive("tracingpages", assign_int, int_base + tracing_pages_code);
primitive("tracingoutput", assign_int, int_base + tracing_output_code);
primitive("tracinglostchars", assign_int, int_base + tracing_lost_chars_code);
primitive("tracingcommands", assign_int, int_base + tracing_commands_code);
primitive("tracingrestores", assign_int, int_base + tracing_restores_code);
primitive("uchyph", assign_int, int_base + uc_yph_code);
primitive("outputpenalty", assign_int, int_base + output_penalty_code);
primitive("maxdeadcycles", assign_int, int_base + max_dead_cycles_code);
primitive("hangafter", assign_int, int_base + hang_after_code);
primitive("floatingpenalty", assign_int, int_base + floating_penalty_code);
primitive("globaldefs", assign_int, int_base + global_defs_code);
primitive("fam", assign_int, int_base + cur_fam_code);
primitive("escapechar", assign_int, int_base + escape_char_code);
primitive("defaulthyphenchar", assign_int, int_base + default_hyphen_char_code);
primitive("defaultskewchar", assign_int, int_base + default_skew_char_code);
primitive("endlinechar", assign_int, int_base + end_line_char_code);
primitive("newlinechar", assign_int, int_base + new_line_char_code);
```

```

primitive("language", assign_int, int_base + language_code);
primitive("lefthyphenmin", assign_int, int_base + left_hyphen_min_code);
primitive("righthyphenmin", assign_int, int_base + right_hyphen_min_code);
primitive("holdinginserts", assign_int, int_base + holding_inserts_code);
primitive("errorcontextlines", assign_int, int_base + error_context_lines_code);
primitive("pdfoutput", assign_int, int_base + pdf_output_code);
primitive("pdfcompresslevel", assign_int, int_base + pdf_compress_level_code);
primitive("pdfobjcompresslevel", assign_int, int_base + pdf_objcompresslevel_code);
primitive("pdfdecimaldigits", assign_int, int_base + pdf_decimal_digits_code);
primitive("pdfmovechars", assign_int, int_base + pdf_move_chars_code);
primitive("pdfimageresolution", assign_int, int_base + pdf_image_resolution_code);
primitive("pdfpkresolution", assign_int, int_base + pdf_pk_resolution_code);
primitive("pdfuniqueresname", assign_int, int_base + pdf_unique_resname_code);
primitive("pdfoptionpdfminorversion", assign_int, int_base + pdf_minor_version_code);
primitive("pdfoptionalwaysusepdfpagebox", assign_int,
          int_base + pdf_option_always_use_pdfpagebox_code);
primitive("pdfoptionpdfinclusionerrorlevel", assign_int,
          int_base + pdf_option_pdf_inclusion_errorlevel_code);
primitive("pdfmajorversion", assign_int, int_base + pdf_major_version_code);
primitive("pdfminorversion", assign_int, int_base + pdf_minor_version_code);
primitive("pdfforcepagebox", assign_int, int_base + pdf_force_pagebox_code);
primitive("pdfpagebox", assign_int, int_base + pdf_pagebox_code);
primitive("pdfinclusionerrorlevel", assign_int, int_base + pdf_inclusion_errorlevel_code);
primitive("pdfgamma", assign_int, int_base + pdf_gamma_code);
primitive("pdfimagegamma", assign_int, int_base + pdf_image_gamma_code);
primitive("pdfimagehicolor", assign_int, int_base + pdf_image_hicolor_code);
primitive("pdfimageapplygamma", assign_int, int_base + pdf_image_apply_gamma_code);
primitive("pdfadjustspacing", assign_int, int_base + pdf_adjust_spacing_code);
primitive("pdfprotrudechars", assign_int, int_base + pdf_protrude_chars_code);
primitive("pdftracingfonts", assign_int, int_base + pdf_tracing_fonts_code);
primitive("pdfadjustinterwordglue", assign_int, int_base + pdf_adjust_interword_glue_code);
primitive("pdfprependkern", assign_int, int_base + pdf_prepend_kern_code);
primitive("pdfappendkern", assign_int, int_base + pdf_append_kern_code);
primitive("pdfgentounicode", assign_int, int_base + pdf_gen_tounicode_code);
primitive("pdfdraftmode", assign_int, int_base + pdf_draftmode_code);
primitive("pdfinclusioncopyfonts", assign_int, int_base + pdf_inclusion_copy_font_code);
primitive("pdfsuppresswarningdupdest", assign_int, int_base + pdf_suppress_warning_dup_dest_code);
primitive("pdfsuppresswarningdupmap", assign_int, int_base + pdf_suppress_warning_dup_map_code);
primitive("pdfsuppresswarningpagegroup", assign_int, int_base + pdf_suppress_warning_page_group_code);
primitive("pdfinfoomitdate", assign_int, int_base + pdf_info_omit_date_code);
primitive("pdfsuppressptexinfo", assign_int, int_base + pdf_suppress_ptex_info_code);
primitive("pdfomitcharset", assign_int, int_base + pdf_omit_charset_code);
primitive("pdfomitinfodict", assign_int, int_base + pdf_omit_info_dict_code);
primitive("pdfomitprocset", assign_int, int_base + pdf_omit_procset_code);
primitive("pdfptexuseunderscore", assign_int, int_base + pdf_ptex_use_underscore_code);

```

257. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡

```

assign_int: if chr_code < count_base then print_param(chr_code - int_base)
else begin print_esc("count"); print_int(chr_code - count_base);
end;

```

258. The integer parameters should really be initialized by a macro package; the following initialization does the minimum to keep T_EX from complete failure.

⟨ Initialize table entries (done by INITEX only) 182 ⟩ ≡

```
for k ← int_base to del_code_base - 1 do eqtb[k].int ← 0;
mag ← 1000; tolerance ← 10000; hang_after ← 1; max_dead_cycles ← 25; escape_char ← "\";
end_line_char ← carriage_return;
for k ← 0 to 255 do del_code(k) ← -1;
del_code(".".) ← 0; { this null delimiter is used in error recovery }
```

259. The following procedure, which is called just before T_EX initializes its input and output, establishes the initial values of the date and time. Since standard Pascal cannot provide such information, something special is needed. The program here simply assumes that suitable values appear in the global variables *sys_time*, *sys_day*, *sys_month*, and *sys_year* (which are initialized to noon on 4 July 1776, in case the implementor is careless).

```
procedure fix_date_and_time;
begin sys_time ← 12 * 60; sys_day ← 4; sys_month ← 7; sys_year ← 1776; { self-evident truths }
time ← sys_time; { minutes since midnight }
day ← sys_day; { day of the month }
month ← sys_month; { month of the year }
year ← sys_year; { Anno Domini }
end;
```

260. ⟨ Show equivalent *n*, in region 5 260 ⟩ ≡

```
begin if n < count_base then print_param(n - int_base)
else if n < del_code_base then
begin print_esc("count"); print_int(n - count_base);
end
else begin print_esc("delcode"); print_int(n - del_code_base);
end;
print_char("=");
print_int(eqtb[n].int);
end
```

This code is used in section 270.

261. ⟨ Set variable *c* to the current escape character 261 ⟩ ≡

```
c ← escape_char
```

This code is used in section 63.

262. ⟨ Character *s* is the current new-line character 262 ⟩ ≡

```
s = new_line_char
```

This code is used in sections 58 and 59.

263. TeX is occasionally supposed to print diagnostic information that goes only into the transcript file, unless *tracing_online* is positive. Here are two routines that adjust the destination of print commands:

```
procedure begin_diagnostic; { prepare to do some tracing }
  begin old_setting ← selector;
  if (tracing_online ≤ 0) ∧ (selector = term_and_log) then
    begin decr(selector);
    if history = spotless then history ← warning_issued;
    end;
  end;

procedure end_diagnostic(blank_line : boolean); { restore proper conditions after tracing }
  begin print_nl("");
  if blank_line then print_ln;
  selector ← old_setting;
  end;
```

264. Of course we had better declare a few more global variables, if the previous routines are going to work.

```
{Global variables 13} +≡
old_setting: 0 .. max_selector;
sys_time, sys_day, sys_month, sys_year: integer; { date and time supplied by external system }
```

265. The final region of *eqtb* contains the dimension parameters defined here, and the 256 \dimen registers.

```

define par_indent_code = 0 { indentation of paragraphs }
define math_surround_code = 1 { space around math in text }
define line_skip_limit_code = 2 { threshold for line_skip instead of baseline_skip }
define hsize_code = 3 { line width in horizontal mode }
define vsize_code = 4 { page height in vertical mode }
define max_depth_code = 5 { maximum depth of boxes on main pages }
define split_max_depth_code = 6 { maximum depth of boxes on split pages }
define box_max_depth_code = 7 { maximum depth of explicit vboxes }
define hfuzz_code = 8 { tolerance for overfull hbox messages }
define vfuzz_code = 9 { tolerance for overfull vbox messages }
define delimiter_shortfall_code = 10 { maximum amount uncovered by variable delimiters }
define null_delimiter_space_code = 11 { blank space in null delimiters }
define script_space_code = 12 { extra space after subscript or superscript }
define pre_display_size_code = 13 { length of text preceding a display }
define display_width_code = 14 { length of line for displayed equation }
define display_indent_code = 15 { indentation of line for displayed equation }
define overfull_rule_code = 16 { width of rule that identifies overfull hboxes }
define hang_indent_code = 17 { amount of hanging indentation }
define h_offset_code = 18 { amount of horizontal offset when shipping pages out }
define v_offset_code = 19 { amount of vertical offset when shipping pages out }
define emergency_stretch_code = 20 { reduces badnesses on final pass of line-breaking }
define pdftex_first_dimen_code = 21 { first number defined in this section }
define pdf_h_origin_code = pdftex_first_dimen_code + 0 { horigin of the PDF output }
define pdf_v_origin_code = pdftex_first_dimen_code + 1 { vorigin of the PDF output }
define pdf_page_width_code = pdftex_first_dimen_code + 2 { page width of the PDF output }
define pdf_page_height_code = pdftex_first_dimen_code + 3 { page height of the PDF output }
define pdf_link_margin_code = pdftex_first_dimen_code + 4 { link margin in the PDF output }
define pdf_dest_margin_code = pdftex_first_dimen_code + 5 { dest margin in the PDF output }
define pdf_thread_margin_code = pdftex_first_dimen_code + 6 { thread margin in the PDF output }
define pdf_first_line_height_code = pdftex_first_dimen_code + 7
define pdf_last_line_depth_code = pdftex_first_dimen_code + 8
define pdf_each_line_height_code = pdftex_first_dimen_code + 9
define pdf_each_line_depth_code = pdftex_first_dimen_code + 10
define pdf_ignored_dimen_code = pdftex_first_dimen_code + 11
define pdf_px_dimen_code = pdftex_first_dimen_code + 12
define pdftex_last_dimen_code = pdftex_first_dimen_code + 12 { last number defined in this section }
define dimen_pars = pdftex_last_dimen_code + 1 { total number of dimension parameters }
define scaled_base = dimen_base + dimen_pars { table of 256 user-defined \dimen registers }
define eqtb_size = scaled_base + 255 { largest subscript of eqtb }

define dimen(#) $\equiv$  eqtb[scaled_base + #].sc
define dimen_par(#) $\equiv$  eqtb[dimen_base + #].sc { a scaled quantity }
define par_indent  $\equiv$  dimen_par(par_indent_code)
define math_surround  $\equiv$  dimen_par(math_surround_code)
define line_skip_limit  $\equiv$  dimen_par(line_skip_limit_code)
define hsize  $\equiv$  dimen_par(hsize_code)
define vsize  $\equiv$  dimen_par(vsize_code)
define max_depth  $\equiv$  dimen_par(max_depth_code)
define split_max_depth  $\equiv$  dimen_par(split_max_depth_code)
define box_max_depth  $\equiv$  dimen_par(box_max_depth_code)
define hfuzz  $\equiv$  dimen_par(hfuzz_code)
define vfuzz  $\equiv$  dimen_par(vfuzz_code)

```

```

define delimiter_shortfall ≡ dimen_par(delimiter_shortfall_code)
define null_delimiter_space ≡ dimen_par(null_delimiter_space_code)
define script_space ≡ dimen_par(script_space_code)
define pre_display_size ≡ dimen_par(pre_display_size_code)
define display_width ≡ dimen_par(display_width_code)
define display_indent ≡ dimen_par(display_indent_code)
define overfull_rule ≡ dimen_par(overfull_rule_code)
define hang_indent ≡ dimen_par(hang_indent_code)
define h_offset ≡ dimen_par(h_offset_code)
define v_offset ≡ dimen_par(v_offset_code)
define emergency_stretch ≡ dimen_par(emergency_stretch_code)
define pdf_h_origin ≡ dimen_par(pdf_h_origin_code)
define pdf_v_origin ≡ dimen_par(pdf_v_origin_code)
define pdf_page_width ≡ dimen_par(pdf_page_width_code)
define pdf_page_height ≡ dimen_par(pdf_page_height_code)
define pdf_link_margin ≡ dimen_par(pdf_link_margin_code)
define pdf_dest_margin ≡ dimen_par(pdf_dest_margin_code)
define pdf_thread_margin ≡ dimen_par(pdf_thread_margin_code)
define pdf_first_line_height ≡ dimen_par(pdf_first_line_height_code)
define pdf_last_line_depth ≡ dimen_par(pdf_last_line_depth_code)
define pdf_each_line_height ≡ dimen_par(pdf_each_line_height_code)
define pdf_each_line_depth ≡ dimen_par(pdf_each_line_depth_code)
define pdf_ignored_dimen ≡ dimen_par(pdf_ignored_dimen_code)
define pdf_px_dimen ≡ dimen_par(pdf_px_dimen_code)

procedure print_length_param(n : integer);
begin case n of
  par_indent_code: print_esc("parindent");
  math_surround_code: print_esc("mathsurround");
  line_skip_limit_code: print_esc("lineskiplimit");
  hsize_code: print_esc("hsize");
  vsiz_code: print_esc("vsiz");
  max_depth_code: print_esc("maxdepth");
  split_max_depth_code: print_esc("splitmaxdepth");
  box_max_depth_code: print_esc("boxmaxdepth");
  hfuzz_code: print_esc("hfuzz");
  vfuzz_code: print_esc("vfuzz");
  delimiter_shortfall_code: print_esc("delimitershrtfall");
  null_delimiter_space_code: print_esc("nulldelimiterspace");
  script_space_code: print_esc("scriptspace");
  pre_display_size_code: print_esc("predisplaysize");
  display_width_code: print_esc("displaywidth");
  display_indent_code: print_esc("displayindent");
  overfull_rule_code: print_esc("overfullrule");
  hang_indent_code: print_esc("hangindent");
  h_offset_code: print_esc("hoffset");
  v_offset_code: print_esc("voffset");
  emergency_stretch_code: print_esc("emergencystretch");
  pdf_h_origin_code: print_esc("pdfhorigin");
  pdf_v_origin_code: print_esc("pdfvorigin");
  pdf_page_width_code: print_esc("pdfpagewidth");
  pdf_page_height_code: print_esc("pdfpageheight");
  pdf_link_margin_code: print_esc("pdflinkmargin");

```

```

pdf_dest_margin_code: print_esc("pdfdestmargin");
pdf_thread_margin_code: print_esc("pdfthreadmargin");
pdf_first_line_height_code: print_esc("pdffirstlineheight");
pdf_last_line_depth_code: print_esc("pdflastlinedepth");
pdf_each_line_height_code: print_esc("pdfeachlineheight");
pdf_each_line_depth_code: print_esc("pdfeachlinedepth");
pdf_ignored_dimen_code: print_esc("pdfignoreddimen");
pdf_px_dimen_code: print_esc("pdfpxdimen");
othercases print("[unknown_dimen_parameter!] ")
endcases;
end;

```

266. ⟨ Put each of TeX's primitives into the hash table 244 ⟩ +≡

```

primitive("parindent", assign_dimen, dimen_base + par_indent_code);
primitive("mathsurround", assign_dimen, dimen_base + math_surround_code);
primitive("lineskiplimit", assign_dimen, dimen_base + line_skip_limit_code);
primitive("hsize", assign_dimen, dimen_base + hsize_code);
primitive("vsize", assign_dimen, dimen_base + vsize_code);
primitive("maxdepth", assign_dimen, dimen_base + max_depth_code);
primitive("splitmaxdepth", assign_dimen, dimen_base + split_max_depth_code);
primitive("boxmaxdepth", assign_dimen, dimen_base + box_max_depth_code);
primitive("hfuzz", assign_dimen, dimen_base + hfuzz_code);
primitive("vfuzz", assign_dimen, dimen_base + vfuzz_code);
primitive("delimitershortfall", assign_dimen, dimen_base + delimiter_shortfall_code);
primitive("nulldelimiterspace", assign_dimen, dimen_base + null_delimiter_space_code);
primitive("scriptspace", assign_dimen, dimen_base + script_space_code);
primitive("predisplaysize", assign_dimen, dimen_base + pre_display_size_code);
primitive("displaywidth", assign_dimen, dimen_base + display_width_code);
primitive("displayindent", assign_dimen, dimen_base + display_indent_code);
primitive("overfullrule", assign_dimen, dimen_base + overfull_rule_code);
primitive("hangindent", assign_dimen, dimen_base + hang_indent_code);
primitive("hoffset", assign_dimen, dimen_base + h_offset_code);
primitive("voffset", assign_dimen, dimen_base + v_offset_code);
primitive("emergencystretch", assign_dimen, dimen_base + emergency_stretch_code);
primitive("pdfhorigin", assign_dimen, dimen_base + pdf_h_origin_code);
primitive("pdfvorigin", assign_dimen, dimen_base + pdf_v_origin_code);
primitive("pdfpagewidth", assign_dimen, dimen_base + pdf_page_width_code);
primitive("pdfpageheight", assign_dimen, dimen_base + pdf_page_height_code);
primitive("pdflinkmargin", assign_dimen, dimen_base + pdf_link_margin_code);
primitive("pdfdestmargin", assign_dimen, dimen_base + pdf_dest_margin_code);
primitive("pdfthreadmargin", assign_dimen, dimen_base + pdf_thread_margin_code);
primitive("pdffirstlineheight", assign_dimen, dimen_base + pdf_first_line_height_code);
primitive("pdflastlinedepth", assign_dimen, dimen_base + pdf_last_line_depth_code);
primitive("pdfeachlineheight", assign_dimen, dimen_base + pdf_each_line_height_code);
primitive("pdfeachlinedepth", assign_dimen, dimen_base + pdf_each_line_depth_code);
primitive("pdfignoreddimen", assign_dimen, dimen_base + pdf_ignored_dimen_code);
primitive("pdfpxdimen", assign_dimen, dimen_base + pdf_px_dimen_code);

```

267. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 245 ⟩ +≡

```

assign_dimen: if chr_code < scaled_base then print_length_param(chr_code - dimen_base)
else begin print_esc("dimen"); print_int(chr_code - scaled_base);
end;

```

268. ⟨ Initialize table entries (done by INITEX only) 182 ⟩ +≡
for $k \leftarrow \text{dimen_base}$ **to** eqtb_size **do** $\text{eqtb}[k].sc \leftarrow 0;$

269. ⟨ Show equivalent n , in region 6 269 ⟩ ≡
begin if $n < \text{scaled_base}$ **then** $\text{print_length_param}(n - \text{dimen_base})$
else begin $\text{print_esc}(\text{"dimen"})$; $\text{print_int}(n - \text{scaled_base})$;
**end};
 $\text{print_char}("=")$; $\text{print_scaled}(\text{eqtb}[n].sc)$; $\text{print}(\text{"pt"})$;
end**

This code is used in section 270.

270. Here is a procedure that displays the contents of $\text{eqtb}[n]$ symbolically.

⟨ Declare the procedure called print_cmd_chr 320 ⟩
stat procedure $\text{show_eqtb}(n : \text{pointer})$;
begin if $n < \text{active_base}$ **then** $\text{print_char}("?)$ { this can't happen }
else if $n < \text{glue_base}$ **then** ⟨ Show equivalent n , in region 1 or 2 241 ⟩
else if $n < \text{local_base}$ **then** ⟨ Show equivalent n , in region 3 247 ⟩
else if $n < \text{int_base}$ **then** ⟨ Show equivalent n , in region 4 251 ⟩
else if $n < \text{dimen_base}$ **then** ⟨ Show equivalent n , in region 5 260 ⟩
else if $n \leq \text{eqtb_size}$ **then** ⟨ Show equivalent n , in region 6 269 ⟩
else $\text{print_char}("?)$; { this can't happen either }
**end};
tats**

271. The last two regions of eqtb have fullword values instead of the three fields eq_level , eq_type , and equiv . An eq_type is unnecessary, but TeX needs to store the eq_level information in another array called xeq_level .

⟨ Global variables 13 ⟩ +≡
 eqtb : **array** [$\text{active_base} \dots \text{eqtb_size}$] **of** memory_word ;
 xeq_level : **array** [$\text{int_base} \dots \text{eqtb_size}$] **of** quarterword ;

272. ⟨ Set initial values of key variables 21 ⟩ +≡
for $k \leftarrow \text{int_base}$ **to** eqtb_size **do** $\text{xeq_level}[k] \leftarrow \text{level_one}$;

273. When the debugging routine search_mem is looking for pointers having a given value, it is interested only in regions 1 to 3 of eqtb , and in the first part of region 4.

⟨ Search eqtb for equivalents equal to p 273 ⟩ ≡
for $q \leftarrow \text{active_base}$ **to** $\text{box_base} + 255$ **do**
begin if $\text{equiv}(q) = p$ **then**
begin $\text{print_nl}(\text{"EQUIV("})$; $\text{print_int}(q)$; $\text{print_char}(")")$;
**end};
end**

This code is used in section 190.

274. The hash table. Control sequences are stored and retrieved by means of a fairly standard hash table algorithm called the method of “coalescing lists” (cf. Algorithm 6.4C in *The Art of Computer Programming*). Once a control sequence enters the table, it is never removed, because there are complicated situations involving \gdef where the removal of a control sequence at the end of a group would be a mistake preventable only by the introduction of a complicated reference-count mechanism.

The actual sequence of letters forming a control sequence identifier is stored in the *str_pool* array together with all the other strings. An auxiliary array *hash* consists of items with two halfword fields per word. The first of these, called *next(p)*, points to the next identifier belonging to the same coalesced list as the identifier corresponding to *p*; and the other, called *text(p)*, points to the *str_start* entry for *p*'s identifier. If position *p* of the hash table is empty, we have *text(p) = 0*; if position *p* is either empty or the end of a coalesced hash list, we have *next(p) = 0*. An auxiliary pointer variable called *hash_used* is maintained in such a way that all locations *p ≥ hash_used* are nonempty. The global variable *cs_count* tells how many multiletter control sequences have been defined, if statistics are being kept.

A global boolean variable called *no_new_control_sequence* is set to *true* during the time that new hash table entries are forbidden.

```
define next(#) ≡ hash[#].lh { link for coalesced lists }
define text(#) ≡ hash[#].rh { string number for control sequence name }
define hash_is_full ≡ (hash_used = hash_base) { test if all positions are occupied }
define font_id_text(#) ≡ text(font_id_base + #) { a frozen font identifier's name }

⟨ Global variables 13 ⟩ +≡
hash: array [hash_base .. undefined_control_sequence - 1] of two_halves; { the hash table }
hash_used: pointer; { allocation pointer for hash }
no_new_control_sequence: boolean; { are new identifiers legal? }
cs_count: integer; { total number of known identifiers }
```

275. Primitive support needs a few extra variables and definitions

```
define prim_prime = 1777 { about 85% of primitive_size }
define prim_base = 1
define prim_next(#) ≡ prim[#].lh { link for coalesced lists }
define prim_text(#) ≡ prim[#].rh { string number for control sequence name, plus one }
define prim_is_full ≡ (prim_used = prim_base) { test if all positions are occupied }
define prim_eq_level_field(#) ≡ #.hh.b1
define prim_eq_type_field(#) ≡ #.hh.b0
define prim_equiv_field(#) ≡ #.hh.rh
define prim_eq_level(#) ≡ prim_eq_level_field(eqtb[prim_eqtb_base + #]) { level of definition }
define prim_eq_type(#) ≡ prim_eq_type_field(eqtb[prim_eqtb_base + #]) { command code for equivalent }
define prim_equiv(#) ≡ prim_equiv_field(eqtb[prim_eqtb_base + #]) { equivalent value }
define undefined_primitive = 0
define biggest_char = 255 { 65535 in XeTeX }

⟨ Global variables 13 ⟩ +≡
prim: array [0 .. prim_size] of two_halves; { the primitives table }
prim_used: pointer; { allocation pointer for prim }
```

276. ⟨ Set initial values of key variables 21 ⟩ +≡

```
no_new_control_sequence ← true; { new identifiers are usually forbidden }
prim_next(0) ← 0; prim_text(0) ← 0;
for k ← 1 to prim_size do prim[k] ← prim[0];
next(hash_base) ← 0; text(hash_base) ← 0;
for k ← hash_base + 1 to undefined_control_sequence - 1 do hash[k] ← hash[hash_base];
```

277. ⟨ Initialize table entries (done by INITEX only) 182 ⟩ +≡
 $\text{prim_used} \leftarrow \text{prim_size}; \quad \{ \text{nothing is used} \}$
 $\text{hash_used} \leftarrow \text{frozen_control_sequence}; \quad \{ \text{nothing is used} \}$
 $\text{cs_count} \leftarrow 0; \quad \text{eq_type}(\text{frozen_dont_expand}) \leftarrow \text{dont_expand};$
 $\text{text}(\text{frozen_dont_expand}) \leftarrow \text{"notexpanded:"}; \quad \text{eq_type}(\text{frozen_primitive}) \leftarrow \text{ignore_spaces};$
 $\text{equiv}(\text{frozen_primitive}) \leftarrow 1; \quad \text{eq_level}(\text{frozen_primitive}) \leftarrow \text{level_one};$
 $\text{text}(\text{frozen_primitive}) \leftarrow \text{"pdfprimitive"};$

278. Here is the subroutine that searches the hash table for an identifier that matches a given string of length $l > 1$ appearing in $\text{buffer}[j \dots (j + l - 1)]$. If the identifier is found, the corresponding hash table address is returned. Otherwise, if the global variable $\text{no_new_control_sequence}$ is true, the dummy address $\text{undefined_control_sequence}$ is returned. Otherwise the identifier is inserted into the hash table and its location is returned.

```
function id_lookup(j, l : integer): pointer; { search the hash table }
label found; { go here if you found it }
var h: integer; { hash code }
      d: integer; { number of characters in incomplete current string }
      p: pointer; { index in hash array }
      k: pointer; { index in buffer array }
begin {Compute the hash code h 280};
      p ← h + hash_base; { we start searching here; note that  $0 \leq h < \text{hash\_prime}$  }
loop begin if text(p) > 0 then
      if length(text(p)) = l then
          if str_eq_buf(text(p), j) then goto found;
      if next(p) = 0 then
          begin if no_new_control_sequence then p ← undefined_control_sequence
          else { Insert a new control sequence after p, then make p point to it 279 };
          goto found;
      end;
      p ← next(p);
  end;
found: id_lookup ← p;
end;
```

279. ⟨ Insert a new control sequence after *p*, then make *p* point to it 279 ⟩ ≡
begin if *text(p)* > 0 then
 begin repeat if *hash_is_full* then *overflow("hash_size", hash_size)*;
 decr(hash_used);
 until *text(hash_used)* = 0; { search for an empty location in hash }
 next(p) ← *hash_used*; *p* ← *hash_used*;
 end;
 str_room(l); *d* ← *cur_length*;
 while *pool_ptr* > *str_start[str_ptr]* do
 begin *decr(pool_ptr)*; *str_pool[pool_ptr + l]* ← *str_pool[pool_ptr]*;
 end; { move current string up to make room for another }
 for *k* ← *j* to *j + l - 1* do *append_char(buffer[k])*;
 text(p) ← *make_string*; *pool_ptr* ← *pool_ptr + d*;
 stat *incr(cs_count)*; tats
end

This code is used in section 278.

280. The value of *hash_prime* should be roughly 85% of *hash_size*, and it should be a prime number. The theory of hashing tells us to expect fewer than two table probes, on the average, when the search is successful. [See J. S. Vitter, *Journal of the ACM* **30** (1983), 231–258.]

⟨Compute the hash code *h* 280⟩ ≡

```

h ← buffer[j]
for k ← j + 1 to j + l – 1 do
  begin h ← h + h + buffer[k]
  while h ≥ hash_prime do h ← h – hash_prime;
  end
```

This code is used in section 278.

281. Here is the subroutine that searches the primitive table for an identifier:

```

function prim_lookup(s : str_number): pointer; { search the primitives table }
  label found; { go here if you found it }
  var h: integer; { hash code }
    p: pointer; { index in hash array }
    k: pointer; { index in string pool }
    j, l: integer;
  begin if s ≤ biggest_char then
    begin if s < 0 then
      begin p ← undefined_primitive; goto found;
      end
    else p ← (s mod prim_prime) + prim_base; { we start searching here }
    end
  else begin j ← str_start[s]
    if s = str_ptr then l ← cur_length
    else l ← length(s);
    ⟨Compute the primitive code h 283⟩;
    p ← h + prim_base; { we start searching here; note that 0 ≤ h < prim_prime }
    end;
  loop begin if prim_text(p) > 1 + biggest_char then { p points a multi-letter primitive }
    begin if length(prim_text(p) – 1) = l then
      if str_eq_str(prim_text(p) – 1, s) then goto found;
      end
    else if prim_text(p) = 1 + s then goto found; { p points a single-letter primitive }
    if prim_next(p) = 0 then
      begin if no_new_control_sequence then p ← undefined_primitive
      else ⟨Insert a new primitive after p, then make p point to it 282⟩;
      goto found;
      end;
    p ← prim_next(p);
    end;
  found: prim_lookup ← p;
  end;
```

282. ⟨ Insert a new primitive after p , then make p point to it 282 ⟩ ≡

```

begin if prim_text( $p$ ) > 0 then
  begin repeat if prim_is_full then overflow("primitive_size", prim_size);
    decr(prim_used);
  until prim_text(prim_used) = 0; { search for an empty location in prim }
  prim_next( $p$ ) ← prim_used;  $p$  ← prim_used;
end;
prim_text( $p$ ) ←  $s + 1$ ;
end

```

This code is used in section 281.

283. The value of $prim_prime$ should be roughly 85% of $prim_size$, and it should be a prime number.

⟨ Compute the primitive code h 283 ⟩ ≡

```

 $h \leftarrow str\_pool[j]$ ;
for  $k \leftarrow j + 1$  to  $j + l - 1$  do
  begin  $h \leftarrow h + h + str\_pool[k]$ ;
  while  $h \geq prim\_prime$  do  $h \leftarrow h - prim\_prime$ ;
end

```

This code is used in section 281.

284. Single-character control sequences do not need to be looked up in a hash table, since we can use the character code itself as a direct address. The procedure $print_cs$ prints the name of a control sequence, given a pointer to its address in $eqtb$. A space is printed after the name unless it is a single nonletter or an active character. This procedure might be invoked with invalid data, so it is “extra robust.” The individual characters must be printed one at a time using $print$, since they may be unprintable.

⟨ Basic printing procedures 57 ⟩ +≡

```

procedure print_cs( $p$ : integer); { prints a purported control sequence }
begin if  $p < hash\_base$  then { single character }
  if  $p \geq single\_base$  then
    if  $p = null\_cs$  then
      begin print_esc("csname"); print_esc("endcsname"); print_char("□");
    end
    else begin print_esc( $p - single\_base$ );
      if cat_code( $p - single\_base$ ) = letter then print_char("□");
    end
  else if  $p < active\_base$  then print_esc("IMPOSSIBLE .")
    else print( $p - active\_base$ )
  else if  $p \geq undefined\_control\_sequence$  then print_esc("IMPOSSIBLE .")
  else if ( $text(p) < 0$ )  $\vee$  ( $text(p) \geq str\_ptr$ ) then print_esc("NONEXISTENT .")
  else begin if ( $p \geq prim\_eqtb\_base$ )  $\wedge$  ( $p < frozen\_null\_font$ ) then
    print_esc(prim_text( $p - prim\_eqtb\_base$ ) - 1)
    else print_esc(text( $p$ ));
    print_char("□");
  end;
end;

```

285. Here is a similar procedure; it avoids the error checks, and it never prints a space after the control sequence.

```
⟨ Basic printing procedures 57 ⟩ +≡  
procedure sprint_cs(p : pointer); { prints a control sequence }  
begin if p < hash_base then  
    if p < single_base then print(p − active_base)  
    else if p < null_cs then print_esc(p − single_base)  
    else begin print_esc("csname"); print_esc("endcsname");  
            end  
    else if (p ≥ prim_eqtb_base) ∧ (p < frozen_null_font) then print_esc(prim_text(p − prim_eqtb_base) − 1)  
        else print_esc(text(p));  
    end;  
end;
```

286. 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.

Until pdfTeX 1.40.19 (released in 2018), a bug in primitive handling caused, e.g., `\pdfprimitive\ \q` to swallow the `\q` instead of giving an undefined control sequence error. The original report was posted by Hironori Kitagawa (tug.org/pipermail/tex-k/2017-October/002816.html). Largely quoting from that message:

The cause was *cur_tok* not being set in the "Cases of *main_control...*" module, because *back_input* unscans the token, but only looks at *cur_tok*, which represents the internalized `\pdfprimitive` at that time. So `\pdfprimitive\vrule\q` becomes "(internalized `\pdfprimitive`)"`\q`, hence no error (and `\vrule` disappears).

Hironori's explanation of the previous behavior and fix continues (off-list):

- * *back_input* (and similar routine `<Insert token p into TeX's input>`) only stores *cur_tok* to a token list.
- * When TeX gets input from a token list (at module `(Input from token list, goto restart ...)`), TeX looks at the saved *cur_tok* value *t*, and recover the command code (*cur_cmd*) and its modifier (*cur_chr*) from it:
 - If $t \geq cs_token_flag$, *t* points to an *eqtb* location $t - cs_token_flag$.
 - If $t < cs_token_flag$, *cur_cmd* and *cur_chr* are set with $cur_cmd \leftarrow t \text{div}'400$; $cur_chr \leftarrow t \text{mod}'400$.
 - This *t* is used to display the token (`show_token_list`).
- * pdfTeX defines *cs_token_flag* as "FFF". So simply using $cur_tok \leftarrow (cur_cmd * '400) + cur_chr$ by `\pdfprimitive` does not work correctly with primitives whose command codes *cur_cmd* ≥ 16 .

Increasing *cs_token_flag* to "FFFF or somewhat higher might suffice for fixing this situation in pdfTeX. However, this approach does not seem good, because

- 1) an (indirect) mapping from *cur_tok* to control sequence name is needed anyway, for displaying the token, and
- 2) this does not work in Japanese e-(u)pTeX.

Thus, we now put *prim_eqtb* entries into the end of region 2 of *eqtb* (which contains some frozen primitives, such as "frozen `\fi`" and "frozen `\cr`"), thus treating *prim_eqtb* entries as a permanent location for primitives.

```

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 }
  prim_val: integer; { needed to fill prim_eqtb }

begin if s < 256 then
  begin cur_val  $\leftarrow s + single\_base$ ; prim_val  $\leftarrow prim\_lookup(s)$ ;
  end
else begin k  $\leftarrow str\_start[s]$ ; l  $\leftarrow str\_start[s + 1] - k$ ;
  { we will move s into the (possibly non-empty) buffer }
  if first + l > buf_size + 1 then overflow("buffer_size", buf_size);
  for j  $\leftarrow 0$  to l - 1 do buffer[first + j]  $\leftarrow so(str\_pool[k + j])$ ;
  cur_val  $\leftarrow id\_lookup(first, l)$ ; { no_new_control_sequence is false }
  flush_string; text(cur_val)  $\leftarrow s$ ; { we don't want to have the string twice }
  prim_val  $\leftarrow prim\_lookup(s)$ ;
  end;
  eq_level(cur_val)  $\leftarrow level\_one$ ; eq_type(cur_val)  $\leftarrow c$ ; equiv(cur_val)  $\leftarrow o$ ;
  prim_eq_level(prim_val)  $\leftarrow level\_one$ ; prim_eq_type(prim_val)  $\leftarrow c$ ; prim_equiv(prim_val)  $\leftarrow o$ ;
end;
tini

```

287. 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 244 ⟩ +≡

```

primitive("□", ex_space, 0);
primitive("/", italic_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("letterspacefont", letterspace_font, 0);
primitive("pdfcopyfont", pdf_copy_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("pdfprimitive", no_expand, 1);
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);

```

288. 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 245 ⟩ +≡
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");
letterspace_font: print_esc("letterspacefont");
pdf_copy_font: print_esc("pdfcopyfont");
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 1763 ⟩;
halign: print_esc("halign");
hrule: print_esc("hrule");
ignore_spaces: if chr_code = 0 then print_esc("ignorespaces")
    else print_esc("pdfprimitive");
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: if chr_code = 0 then print_esc("noexpand")
    else print_esc("pdfprimitive");
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 1760 ⟩;
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 1865 ⟩
    end; { there are no other cases }
the: if chr_code = 0 then print_esc("the") ⟨ Cases of the for print_cmd_chr 1687 ⟩;

```

```
toks_register: ⟨ Cases of toks_register for print_cmd_chr 1833 ⟩;  
vadjust: print_esc("vadjust");  
valign: if chr_code = 0 then print_esc("valign")  
    ⟨ Cases of valign for print_cmd_chr 1702 ⟩;  
vcenter: print_esc("vcenter");  
vrule: print_esc("vrule");
```

289. We will deal with the other primitives later, at some point in the program where their *eq_type* and *equiv* values are more meaningful. For example, the primitives for math mode will be loaded when we consider the routines that deal with formulas. It is easy to find where each particular primitive was treated by looking in the index at the end; for example, the section where "radical" entered *eqtb* is listed under '\radical primitive'. (Primitives consisting of a single nonalphabetic character, like '\'', are listed under 'Single-character primitives'.)

Meanwhile, this is a convenient place to catch up on something we were unable to do before the hash table was defined:

290. 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 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 306 ⟩

291. Here are the group codes that are used to discriminate between different kinds of groups. They allow TeX to decide what special actions, if any, should be performed when a group ends.

Some groups are not supposed to be ended by right braces. For example, the '\$' that begins a math formula causes a *math-shift-group* to be started, and this should be terminated by a matching '\$'. Similarly, a group that starts with \left should end with \right, and one that starts with \begingroup should end with \endgroup.

```
define bottom_level = 0 { group code for the outside world }
define simple_group = 1 { group code for local structure only }
define hbox_group = 2 { code for '\hbox{...}' }
define adjusted_hbox_group = 3 { code for '\hbox{...}' in vertical mode }
define vbox_group = 4 { code for '\vbox{...}' }
define vtop_group = 5 { code for '\vtop{...}' }
define align_group = 6 { code for '\halign{...}', '\valign{...}' }
define no_align_group = 7 { code for '\noalign{...}' }
define output_group = 8 { code for output routine }
define math_group = 9 { code for, e.g., '^{...}' }
define disc_group = 10 { code for '\discretionary{...}{...}{...}' }
define insert_group = 11 { code for '\insert{...}', '\vadjust{...}' }
define vcenter_group = 12 { code for '\vcenter{...}' }
define math_choice_group = 13 { code for '\mathchoice{...}{...}{...}{...}' }
define semi_simple_group = 14 { code for '\begingroup... \endgroup' }
define math_shift_group = 15 { code for '$...$' }
define math_left_group = 16 { code for '\left... \right' }
define max_group_code = 16

⟨ Types in the outer block 18 ⟩ +≡
group_code = 0 .. max_group_code; { save_level for a level boundary }
```

292. The global variable *cur_group* keeps track of what sort of group we are currently in. Another global variable, *cur_boundary*, points to the topmost *level_boundary* word. And *cur_level* is the current depth of nesting. The routines are designed to preserve the condition that no entry in the *save_stack* or in *eqtb* ever has a level greater than *cur_level*.

293. ⟨ Global variables 13 ⟩ +≡

```
save_stack: array [0 .. save_size] of memory_word;
save_ptr: 0 .. save_size; { first unused entry on save_stack }
max_save_stack: 0 .. save_size; { maximum usage of save stack }
cur_level: quarterword; { current nesting level for groups }
cur_group: group_code; { current group type }
cur_boundary: 0 .. save_size; { where the current level begins }
```

294. At this time it might be a good idea for the reader to review the introduction to *eqtb* that was given above just before the long lists of parameter names. Recall that the “outer level” of the program is *level_one*, since undefined control sequences are assumed to be “defined” at *level_zero*.

⟨ Set initial values of key variables 21 ⟩ +≡

```
save_ptr ← 0; cur_level ← level_one; cur_group ← bottom_level; cur_boundary ← 0; max_save_stack ← 0;
```

295. 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
```

296. 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;
```

297. 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 1834 }
    othercases do_nothing
  endcases;
  end;
```

298. To save a value of $eqtb[p]$ that was established at level l , we can use the following subroutine.

```
procedure eq_save(p : pointer; l : quarterword); { saves eqtb[p]}
begin check_full_save_stack;
if l = level_zero then save_type(save_ptr) ← restore_zero
else begin save_stack[save_ptr] ← eqtb[p]; incr(save_ptr); save_type(save_ptr) ← restore_old_value;
end;
save_level(save_ptr) ← l; save_index(save_ptr) ← p; incr(save_ptr);
end;
```

299. 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 ∧ (eq_type(p) = t) ∧ (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) ← cur_level; eq_type(p) ← t; equiv(p) ← e; assign_trace(p, "into")
exit: end;
```

300. The counterpart of *eq_define* for the remaining (fullword) positions in *eqtb* is called *eq_word_define*. Since $x_{eq_level}[p] \geq level_one$ 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 ∧ (eqtb[p].int = w) then
begin assign_trace(p, "reassigning")
return;
end;
assign_trace(p, "changing")
if x_{eq\_level}[p] ≠ cur_level then
begin eq_save(p, x_{eq\_level}[p]); x_{eq\_level}[p] ← cur_level;
end;
eqtb[p].int ← w; assign_trace(p, "into")
exit: end;
```

301. 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) ← level_one; eq_type(p) ← t; equiv(p) ← 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 ← w; xeq_level[p] ← level_one;
  end; assign_trace(p, "into")
  end;
```

302. Subroutine *save_for_after* puts a token on the stack for save-keeping.

```
procedure save_for_after(t : halfword);
  begin if cur_level > level_one then
    begin check_full_save_stack; save_type(save_ptr) ← insert_token; save_level(save_ptr) ← level_zero;
    save_index(save_ptr) ← t; incr(save_ptr);
    end;
  end;
```

303. The *unsafe* 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 unsafe; { 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 304 };
    end
  else confusion("curlevel"); { unsafe is not used when cur_group = bottom_level }
  end;
```

304. \langle Clear off top level from *save_stack* 304 $\rangle \equiv$

```

loop begin decr(save_ptr);
  if save_type(save_ptr) = level_boundary then goto done;
  p  $\leftarrow$  save_index(save_ptr);
  if save_type(save_ptr) = insert_token then  $\langle$  Insert token p into TeX's input 348  $\rangle$ 
  else if save_type(save_ptr) = restore_sa then
    begin sa_restore; sa_chain  $\leftarrow$  p; sa_level  $\leftarrow$  save_level(save_ptr);
    end
  else begin if save_type(save_ptr) = restore_old_value then
    begin l  $\leftarrow$  save_level(save_ptr); decr(save_ptr);
    end
    else save_stack[save_ptr]  $\leftarrow$  eqtb[undefined_control_sequence];
    {Store save_stack[save_ptr] in eqtb[p] unless eqtb[p] holds a global value 305};
    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  $\leftarrow$  save_level(save_ptr); cur_boundary  $\leftarrow$  save_index(save_ptr);
  if eTeX_ex then decr(save_ptr)

```

This code is used in section 303.

305. A global definition, which sets the level to *level_one*, will not be undone by *unsafe*. If at least one global definition of *eqtb*[*p*] has been carried out within the group that just ended, the last such definition will therefore survive.

\langle Store *save_stack*[*save_ptr*] in *eqtb*[*p*], unless *eqtb*[*p*] holds a global value 305 $\rangle \equiv$

```

if p < int_base then
  if eq_level(p) = level_one then
    begin eq_destroy(save_stack[save_ptr]); {destroy the saved value}
    stat if tracing_restores > 0 then restore_trace(p, "retaining");
    tats
    end
  else begin eq_destroy(eqtb[p]); {destroy the current value}
    eqtb[p]  $\leftarrow$  save_stack[save_ptr]; {restore the saved value}
    stat if tracing_restores > 0 then restore_trace(p, "restoring");
    tats
    end
  else if xeq_level[p]  $\neq$  level_one then
    begin eqtb[p]  $\leftarrow$  save_stack[save_ptr]; x eq_level[p]  $\leftarrow$  l;
    stat if tracing_restores > 0 then restore_trace(p, "restoring");
    tats
    end
  else begin stat if tracing_restores > 0 then restore_trace(p, "retaining");
    tats
    end

```

This code is used in section 304.

306. \langle Declare ε -TeX procedures for tracing and input 306 $\rangle \equiv$

```
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 1661, 1662, 1756, 1757, 1774, 1776, 1777, 1821, 1823, 1837, 1838, 1839, 1840, and 1841.

This code is used in section 290.

307. When looking for possible pointers to a memory location, it is helpful to look for references from *eqtb* that might be waiting on the save stack. Of course, we might find spurious pointers too; but this routine is merely an aid when debugging, and at such times we are grateful for any scraps of information, even if they prove to be irrelevant.

\langle Search *save_stack* for equivalents that point to *p* 307 $\rangle \equiv$

```
if save_ptr > 0 then
  for q  $\leftarrow$  0 to save_ptr - 1 do
    begin if equiv_field(save_stack[q]) = p then
      begin print_nl("SAVE("); print_int(q); print_char(")");
      end;
    end
```

This code is used in section 190.

308. Most of the parameters kept in *eqtb* can be changed freely, but there's an exception: The magnification should not be used with two different values during any TeX job, since a single magnification is applied to an entire run. The global variable *mag_set* is set to the current magnification whenever it becomes necessary to "freeze" it at a particular value.

\langle Global variables 13 $\rangle +\equiv$
mag_set: integer; { if nonzero, this magnification should be used henceforth }

309. \langle Set initial values of key variables 21 $\rangle +\equiv$
mag_set \leftarrow 0;

310. The *prepare_mag* subroutine is called whenever TeX wants to use *mag* for magnification.

```
procedure prepare_mag;
begin if (mag_set > 0)  $\wedge$  (mag  $\neq$  mag_set) then
  begin print_err("Incompatible magnification"); print_int(mag); print(");");
  print_nl("the previous value will be retained");
  help2("I can handle only one magnification ratio per job. So I've")
  ("reverted to the magnification you used earlier on this run.");
  int_error(mag_set); geq_word_define(int_base + mag_code, mag_set); { mag  $\leftarrow$  mag_set }
end;
if (mag  $\leq$  0)  $\vee$  (mag > 32768) then
  begin print_err("Illegal magnification has been changed to 1000");
  help1("The magnification ratio must be between 1 and 32768."); int_error(mag);
  geq_word_define(int_base + mag_code, 1000);
end;
mag_set  $\leftarrow$  mag;
end;
```

311. Token lists. A T_EX 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 $1 \leq m \leq 14$. (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 $left_brace_limit \leq t < right_brace_limit$; and it represents a *match* or *end_match* command if and only if $match_token \leq t \leq end_match_token$. 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 { 28 · left_brace }
define left_brace_limit = '1000 { 28 · (left_brace + 1) }
define right_brace_token = '1000 { 28 · right_brace }
define right_brace_limit = '1400 { 28 · (right_brace + 1) }
define math_shift_token = '1400 { 28 · math.shift }
define tab_token = '2000 { 28 · tab_mark }
define out_param_token = '2400 { 28 · out_param }
define space_token = '5040 { 28 · spacer + " " }
define letter_token = '5400 { 28 · letter }
define other_token = '6000 { 28 · other_char }
define match_token = '6400 { 28 · match }
define end_match_token = '7000 { 28 · end.match }
define protected_token = '7001 { 28 · end.match + 1 }
```

312. ⟨Check the “constant” values for consistency 14⟩ +≡
if $cs_token_flag + undefined_control_sequence > max_halfword$ then $bad \leftarrow 21$;

313. A token list is a singly linked list of one-word nodes in *mem*, where each word contains a token and a link. Macro definitions, output-routine definitions, marks, \write texts, and a few other things are remembered by TeX in the form of token lists, usually preceded by a node with a reference count in its *token_ref_count* field. The token stored in location *p* is called *info(p)*.

Three special commands appear in the token lists of macro definitions. When *m* = *match*, it means that TeX should scan a parameter for the current macro; when *m* = *end_match*, it means that parameter matching should end and TeX should start reading the macro text; and when *m* = *out_param*, it means that TeX should insert parameter number *c* into the text at this point.

The enclosing { and } characters of a macro definition are omitted, but an output routine will be enclosed in braces.

Here is an example macro definition that illustrates these conventions. After TeX processes the text

```
\def\mac a#1#2 \b {#1-a ##1#2 #2}
```

the definition of \mac is represented as a token list containing

```
(reference count), letter a, match #, match #, spacer \, \b, end_match,
out_param 1, \-, letter a, spacer \, mac_param #, other_char 1,
out_param 2, spacer \, out_param 2.
```

The procedure *scan_toks* builds such token lists, and *macro_call* does the parameter matching.

Examples such as

```
\def\m{\def\m{a}\b}
```

explain why reference counts would be needed even if TeX had no \let operation: When the token list for \m is being read, the redefinition of \m changes the *eqtb* entry before the token list has been fully consumed, so we dare not simply destroy a token list when its control sequence is being redefined.

If the parameter-matching part of a definition ends with '#{' , the corresponding token list will have '{' just before the 'end_match' and also at the very end. The first '{' is used to delimit the parameter; the second one keeps the first from disappearing.

314. The procedure *show_token_list*, which prints a symbolic form of the token list that starts at a given node *p*, illustrates these conventions. The token list being displayed should not begin with a reference count. However, the procedure is intended to be robust, so that if the memory links are awry or if *p* is not really a pointer to a token list, nothing catastrophic will happen.

An additional parameter *q* is also given; this parameter is either null or it points to a node in the token list where a certain magic computation takes place that will be explained later. (Basically, *q* is non-null when we are printing the two-line context information at the time of an error message; *q* marks the place corresponding to where the second line should begin.)

For example, if *p* points to the node containing the first a in the token list above, then *show_token_list* will print the string

```
'a#1#2\b->#1\~-a##1#2#2';
```

and if *q* points to the node containing the second a, the magic computation will be performed just before the second a is printed.

The generation will stop, and '\ETC.' will be printed, if the length of printing exceeds a given limit *l*. Anomalous entries are printed in the form of control sequences that are not followed by a blank space, e.g., '\BAD.'; this cannot be confused with actual control sequences because a real control sequence named BAD would come out '\BAD'.

```
< Declare the procedure called show_token_list 314 > ≡
procedure show_token_list(p, q : integer; l : integer);
label exit;
var m, c: integer; { pieces of a token }
    match_chr: ASCII_code; { character used in a 'match' }
    n: ASCII_code; { the highest parameter number, as an ASCII digit }
begin match_chr ← "#"; n ← "0"; tally ← 0;
while (p ≠ null) ∧ (tally < l) do
    begin if p = q then < Do magic computation 342 >;
        < Display token p, and return if there are problems 315 >;
        p ← link(p);
    end;
    if p ≠ null then print_esc("ETC.");
exit: end;
```

This code is used in section 137.

315. < Display token *p*, and return if there are problems 315 > ≡

```
if (p < hi_mem_min) ∨ (p > mem_end) then
    begin print_esc("CLOBBERED."); return;
end;
if info(p) ≥ cs_token_flag then print_cs(info(p) - cs_token_flag)
else begin m ← info(p) div '400; c ← info(p) mod '400;
    if info(p) < 0 then print_esc("BAD.")
    else < Display the token (m, c) 316 >;
end
```

This code is used in section 314.

316. 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) 316 ) ≡
  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 ≤ 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 315.

317. Here’s the way we sometimes want to display a token list, given a pointer to its reference count; the pointer may be null.

```
procedure token_show(p : pointer);
begin if p ≠ null then show_token_list(link(p), null, 10000000);
end;
```

318. 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;
```

319. Introduction to the syntactic routines. Let's pause a moment now and try to look at the Big Picture. The TeX program consists of three main parts: syntactic routines, semantic routines, and output routines. The chief purpose of the syntactic routines is to deliver the user's input to the semantic routines, one token at a time. The semantic routines act as an interpreter responding to these tokens, which may be regarded as commands. And the output routines are periodically called on to convert box-and-glue lists into a compact set of instructions that will be sent to a typesetter. We have discussed the basic data structures and utility routines of TeX, so we are good and ready to plunge into the real activity by considering the syntactic routines.

Our current goal is to come to grips with the *get_next* procedure, which is the keystone of TeX's input mechanism. Each call of *get_next* sets the value of three variables *cur_cmd*, *cur_chr*, and *cur_cs*, representing the next input token.

cur_cmd denotes a command code from the long list of codes given above;
cur_chr denotes a character code or other modifier of the command code;
cur_cs is the *eqtb* location of the current control sequence,
 if the current token was a control sequence, otherwise it's zero.

Underlying this external behavior of *get_next* is all the machinery necessary to convert from character files to tokens. At a given time we may be only partially finished with the reading of several files (for which *\input* was specified), and partially finished with the expansion of some user-defined macros and/or some macro parameters, and partially finished with the generation of some text in a template for *\halign*, and so on. When reading a character file, special characters must be classified as math delimiters, etc.; comments and extra blank spaces must be removed, paragraphs must be recognized, and control sequences must be found in the hash table. Furthermore there are occasions in which the scanning routines have looked ahead for a word like 'plus' but only part of that word was found, hence a few characters must be put back into the input and scanned again.

To handle these situations, which might all be present simultaneously, TeX uses various stacks that hold information about the incomplete activities, and there is a finite state control for each level of the input mechanism. These stacks record the current state of an implicitly recursive process, but the *get_next* procedure is not recursive. Therefore it will not be difficult to translate these algorithms into low-level languages that do not support recursion.

⟨ Global variables 13 ⟩ +≡
cur_cmd: *eight_bits*; { current command set by *get_next* }
cur_chr: *halfword*; { operand of current command }
cur_cs: *pointer*; { control sequence found here, zero if none found }
cur_tok: *halfword*; { packed representative of *cur_cmd* and *cur_chr* }

320. 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 320 ⟩ ≡
procedure print_cmd_chr(cmd : quarterword; chr_code : halfword);
  var n: integer; { temp variable }
  begin case cmd of
    left_brace: chr_cmd("begin-group\ucharacter\u");
    right_brace: chr_cmd("end-group\ucharacter\u");
    math_shift: chr_cmd("math\ushift\ucharacter\u");
    mac_param: chr_cmd("macro\uparameter\ucharacter\u");
    sup_mark: chr_cmd("superscript\ucharacter\u");
    sub_mark: chr_cmd("subscript\ucharacter\u");
    endv: print("end\uof\ualignment\utemplate");
    spacer: chr_cmd("blank\u space\u");
    letter: chr_cmd("the\uletter\u");
    other_char: chr_cmd("the\ucharacter\u");
    ⟨ Cases of print_cmd_chr for symbolic printing of primitives 245 ⟩
    othercases print("[unknown\ucommand\ucode!]")
  endcases;
  end;
```

This code is used in section 270.

321. 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;

```

322. Input stacks and states. This implementation of TeX uses two different conventions for representing sequential stacks.

- 1) If there is frequent access to the top entry, and if the stack is essentially never empty, then the top entry is kept in a global variable (even better would be a machine register), and the other entries appear in the array $stack[0 \dots (ptr - 1)]$. For example, the semantic stack described above is handled this way, and so is the input stack that we are about to study.
- 2) If there is infrequent top access, the entire stack contents are in the array $stack[0 \dots (ptr - 1)]$. For example, the $save_stack$ is treated this way, as we have seen.

The state of TeX's input mechanism appears in the input stack, whose entries are records with six fields, called $state$, $index$, $start$, loc , $limit$, and $name$. This stack is maintained with convention (1), so it is declared in the following way:

```
<Types in the outer block 18> +≡
in_state_record = record state_field, index_field: quarterword;
                     start_field, loc_field, limit_field, name_field: halfword;
end;
```

323. < Global variables 13 > +≡

```
input_stack: array [0 .. stack_size] of in_state_record;
input_ptr: 0 .. stack_size; { first unused location of input_stack }
max_in_stack: 0 .. stack_size; { largest value of input_ptr when pushing }
cur_input: in_state_record; { the "top" input state, according to convention (1) }
```

324. We've already defined the special variable $loc \equiv cur_input.loc_field$ in our discussion of basic input-output routines. The other components of cur_input are defined in the same way:

```
define state ≡ cur_input.state_field { current scanner state }
define index ≡ cur_input.index_field { reference for buffer information }
define start ≡ cur_input.start_field { starting position in buffer }
define limit ≡ cur_input.limit_field { end of current line in buffer }
define name ≡ cur_input.name_field { name of the current file }
```

325. 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 $0 \leq n \leq 16$. (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 $18 \leq name \leq 19$ 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 }
```

326. Additional information about the current line is available via the *index* variable, which counts how many lines of characters are present in the buffer below the current level. We have *index* = 0 when reading from the terminal and prompting the user for each line; then if the user types, e.g., ‘\input paper’, we will have *index* = 1 while reading the file *paper.tex*. However, it does not follow that *index* is the same as the input stack pointer, since many of the levels on the input stack may come from token lists. For example, the instruction ‘\input paper’ might occur in a token list.

The global variable *in_open* is equal to the *index* value of the highest non-token-list level. Thus, the number of partially read lines in the buffer is *in_open* + 1, and we have *in_open* = *index* when we are not reading a token list.

If we are not currently reading from the terminal, or from an input stream, we are reading from the file variable *input_file[index]*. We use the notation *terminal_input* as a convenient abbreviation for *name* = 0, and *cur_file* as an abbreviation for *input_file[index]*.

The global variable *line* contains the line number in the topmost open file, for use in error messages. If we are not reading from the terminal, *line_stack[index]* holds the line number for the enclosing level, so that *line* can be restored when the current file has been read. Line numbers should never be negative, since the negative of the current line number is used to identify the user’s output routine in the *mode_line* field of the semantic nest entries.

If more information about the input state is needed, it can be included in small arrays like those shown here. For example, the current page or segment number in the input file might be put into a variable *page*, maintained for enclosing levels in ‘*page_stack: array [1 .. max_in_open] of integer*’ by analogy with *line_stack*.

```
define terminal_input ≡ (name = 0) { are we reading from the terminal? }
define cur_file ≡ input_file[index] { the current alpha_file variable }

⟨ Global variables 13 ⟩ +≡
in_open: 0 .. max_in_open; { the number of lines in the buffer, less one }
open_parens: 0 .. max_in_open; { the number of open text files }
input_file: array [1 .. max_in_open] of alpha_file;
line: integer; { current line number in the current source file }
line_stack: array [1 .. max_in_open] of integer;
```

327. Users of TeX sometimes forget to balance left and right braces properly, and one of the ways TeX tries to spot such errors is by considering an input file as broken into subfiles by control sequences that are declared to be `\outer`.

A variable called `scanner_status` tells TeX whether or not to complain when a subfile ends. This variable has six possible values:

normal, means that a subfile can safely end here without incident.

skipping, means that a subfile can safely end here, but not a file, because we're reading past some conditional text that was not selected.

defining, means that a subfile shouldn't end now because a macro is being defined.

matching, means that a subfile shouldn't end now because a macro is being used and we are searching for the end of its arguments.

aligning, means that a subfile shouldn't end now because we are not finished with the preamble of an `\halign` or `\valign`.

absorbing, means that a subfile shouldn't end now because we are reading a balanced token list for `\message`, `\write`, etc.

If the `scanner_status` is not *normal*, the variable `warning_index` points to the `eqtb` location for the relevant control sequence name to print in an error message.

```
define skipping = 1 { scanner_status when passing conditional text }
define defining = 2 { scanner_status when reading a macro definition }
define matching = 3 { scanner_status when reading macro arguments }
define aligning = 4 { scanner_status when reading an alignment preamble }
define absorbing = 5 { scanner_status when reading a balanced text }

⟨ Global variables 13 ⟩ +≡
scanner_status: normal .. absorbing; { can a subfile end now? }
warning_index: pointer; { identifier relevant to non-normal scanner status }
def_ref: pointer; { reference count of token list being defined }
```

328. Here is a procedure that uses `scanner_status` to print a warning message when a subfile has ended, and at certain other crucial times:

```
⟨ Declare the procedure called runaway 328 ⟩ ≡
procedure runaway;
  var p: pointer; { head of runaway list }
  begin if scanner_status > skipping then
    begin print_nl("Runaway");
    case scanner_status of
      defining: begin print("definition"); p ← def_ref;
      end;
      matching: begin print("argument"); p ← temp_head;
      end;
      aligning: begin print("preamble"); p ← hold_head;
      end;
      absorbing: begin print("text"); p ← def_ref;
      end;
    end; { there are no other cases }
    print_char(?); print_ln; show_token_list(link(p), null, error_line - 10);
  end;
end;
```

This code is used in section 137.

329. 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* \geq *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 }

```

330. The *param_stack* is an auxiliary array used to hold pointers to the token lists for parameters at the current level and subsidiary levels of input. This stack is maintained with convention (2), and it grows at a different rate from the others.

```

⟨ Global variables 13 ⟩ +≡
param_stack: array [0 .. param_size] of pointer; { token list pointers for parameters }
param_ptr: 0 .. param_size; { first unused entry in param_stack }
max_param_stack: integer; { largest value of param_ptr, will be ≤ param_size + 9 }

```

331. The input routines must also interact with the processing of *\halign* and *\valign*, since the appearance of tab marks and *\cr* in certain places is supposed to trigger the beginning of special $\langle v_j \rangle$ template text in the scanner. This magic is accomplished by an *align_state* variable that is increased by 1 when a ‘{’ is scanned and decreased by 1 when a ‘}’ is scanned. The *align_state* is nonzero during the $\langle u_j \rangle$ template, after which it is set to zero; the $\langle v_j \rangle$ template begins when a tab mark or *\cr* occurs at a time that *align_state* = 0.

```

⟨ Global variables 13 ⟩ +≡
align_state: integer; { group level with respect to current alignment }

```

332. Thus, the “current input state” can be very complicated indeed; there can be many levels and each level can arise in a variety of ways. The *show_context* procedure, which is used by TeX’s error-reporting routine to print out the current input state on all levels down to the most recent line of characters from an input file, illustrates most of these conventions. The global variable *base_ptr* contains the lowest level that was displayed by this procedure.

```

⟨ Global variables 13 ⟩ +≡
base_ptr: 0 .. stack_size; { shallowest level shown by show_context }

```

333. 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 337 }
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 334 }
  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;
```

334. { Display the current context 334 } ≡

```

begin if (base_ptr = input_ptr) ∨ (state ≠ token_list) ∨ (token_type ≠ backed_up) ∨ (loc ≠ null) then
  { we omit backed-up token lists that have already been read }
  begin tally ← 0; { get ready to count characters }
  old_setting ← selector;
  if state ≠ token_list then
    begin { Print location of current line 335 };
    { Pseudoprint the line 340 };
    end
  else begin { Print type of token list 336 };
    { Pseudoprint the token list 341 };
    end;
  selector ← old_setting; { stop pseudoprinting }
  { Print two lines using the tricky pseudoprinted information 339 };
  incr(nn);
end;
end
```

This code is used in section 333.

335. 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.

\langle Print location of current line 335 $\rangle \equiv$

```

if name  $\leq$  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("1.");
    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 334.

336. \langle Print type of token list 336 $\rangle \equiv$

```

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 334.

337. Here it is necessary to explain a little trick. We don't want to store a long string that corresponds to a token list, because that string might take up lots of memory; and we are printing during a time when an error message is being given, so we dare not do anything that might overflow one of \TeX 's tables. So 'pseudoprinting' is the answer: We enter a mode of printing that stores characters into a buffer of length error_line , where character $k + 1$ is placed into $\text{trick_buf}[k \bmod \text{error_line}]$ if $k < \text{trick_count}$, otherwise character k is dropped. Initially we set $\text{tally} \leftarrow 0$ and $\text{trick_count} \leftarrow 1000000$; then when we reach the point where transition from line 1 to line 2 should occur, we set $\text{first_count} \leftarrow \text{tally}$ and $\text{trick_count} \leftarrow \max(\text{error_line}, \text{tally} + 1 + \text{error_line} - \text{half_error_line})$. At the end of the pseudoprinting, the values of first_count , tally , and trick_count give us all the information we need to print the two lines, and all of the necessary text is in trick_buf .

Namely, let l be the length of the descriptive information that appears on the first line. The length of the context information gathered for that line is $k = \text{first_count}$, and the length of the context information gathered for line 2 is $m = \min(\text{tally}, \text{trick_count}) - k$. If $l + k \leq h$, where $h = \text{half_error_line}$, we print $\text{trick_buf}[0 \dots k - 1]$ after the descriptive information on line 1, and set $n \leftarrow l + k$; here n is the length of line 1. If $l + k > h$, some cropping is necessary, so we set $n \leftarrow h$ and print '...' followed by

$\text{trick_buf}[(l + k - h + 3) \dots k - 1]$

where subscripts of trick_buf are circular modulo error_line . The second line consists of n spaces followed by $\text{trick_buf}[k \dots (k + m - 1)]$, unless $n + m > \text{error_line}$; in the latter case, further cropping is done. This is easier to program than to explain.

```

⟨Local variables for formatting calculations 337⟩ ≡
i: 0 .. buf_size; { index into buffer }
j: 0 .. buf_size; { end of current line in buffer }
l: 0 .. half_error_line; { length of descriptive information on line 1 }
m: integer; { context information gathered for line 2 }
n: 0 .. error_line; { length of line 1 }
p: integer; { starting or ending place in trick_buf }
q: integer; { temporary index }
```

This code is used in section 333.

338. The following code sets up the print routines so that they will gather the desired information.

```

define begin_pseudoprint ≡
  begin l ← tally; tally ← 0; selector ← pseudo; trick_count ← 1000000;
  end
define set_trick_count ≡
  begin first_count ← tally; trick_count ← tally + 1 + error_line - half_error_line;
  if trick_count < error_line then trick_count ← error_line;
  end
```

339. And the following code uses the information after it has been gathered.

```
< Print two lines using the tricky pseudoprinted information 339 > ≡
  if trick_count = 1000000 then set_trick_count; { set_trick_count must be performed }
  if tally < trick_count then m ← tally - first_count
  else m ← trick_count - first_count; { context on line 2 }
  if l + first_count ≤ half_error_line then
    begin p ← 0; n ← l + first_count;
    end
  else begin print("..."); p ← l + first_count - half_error_line + 3; n ← half_error_line;
    end;
  for q ← p to first_count - 1 do print_char(trick_buf[q mod error_line]);
  print_ln;
  for q ← 1 to n do print_char(" "); { print n spaces to begin line 2 }
  if m + n ≤ error_line then p ← first_count + m
  else p ← first_count + (error_line - n - 3);
  for q ← first_count to p - 1 do print_char(trick_buf[q mod error_line]);
  if m + n > error_line then print("...")
```

This code is used in section 334.

340. But the trick is distracting us from our current goal, which is to understand the input state. So let's concentrate on the data structures that are being pseudoprinted as we finish up the *show_context* procedure.

```
< Pseudoprint the line 340 > ≡
  begin_pseudoprint;
  if buffer[limit] = end_line_char then j ← limit
  else j ← limit + 1; { determine the effective end of the line }
  if j > 0 then
    for i ← start to j - 1 do
      begin if i = loc then set_trick_count;
      print(buffer[i]);
      end
```

This code is used in section 334.

341. < Pseudoprint the token list 341 > ≡
 begin_pseudoprint;
 if token_type < macro then show_token_list(start, loc, 100000)
 else show_token_list(link(start), loc, 100000) { avoid reference count }

This code is used in section 334.

342. Here is the missing piece of *show_token_list* that is activated when the token beginning line 2 is about to be shown:

```
< Do magic computation 342 > ≡
  set_trick_count
```

This code is used in section 314.

343. Maintaining the input stacks. The following subroutines change the input status in commonly needed ways.

First comes *push_input*, which stores the current state and creates a new level (having, initially, the same properties as the old).

```
define push_input ≡ { enter a new input level, save the old }
begin if input_ptr > max_in_stack then
  begin max_in_stack ← input_ptr;
  if input_ptr = stack_size then overflow("input_stack_size", stack_size);
  end;
  input_stack[input_ptr] ← cur_input; { stack the record }
  incr(input_ptr);
end
```

344. And of course what goes up must come down.

```
define pop_input ≡ { leave an input level, re-enter the old }
begin decr(input_ptr); cur_input ← input_stack[input_ptr];
end
```

345. Here is a procedure that starts a new level of token-list input, given a token list *p* and its type *t*. If *t* = *macro*, the calling routine should set *name* and *loc*.

```
define back_list(♯) ≡ begin_token_list(♯, backed_up) { backs up a simple token list }
define ins_list(♯) ≡ begin_token_list(♯, inserted) { inserts a simple token list }

procedure begin_token_list(p : pointer; t : quarterword);
begin push_input; state ← token_list; start ← p; token_type ← t;
if t ≥ macro then { the token list starts with a reference count }
  begin add_token_ref(p);
  if t = macro then param_start ← param_ptr
  else begin loc ← link(p);
    if tracing_macros > 1 then
      begin begin_diagnostic; print_nl("");
      case t of
        mark_text: print_esc("mark");
        write_text: print_esc("write");
        othercases print_cmd_chr(assign_toks, t - output_text + output_routine_loc)
        endcases;
        print("->"); token_show(p); end_diagnostic(false);
      end;
    end;
  end
else loc ← p;
end;
```

346. When a token list has been fully scanned, the following computations should be done as we leave that level of input. The *token_type* tends to be equal to either *backed_up* or *inserted* about 2/3 of the time.

```
procedure end_token_list; { leave a token-list input level }
begin if token_type ≥ backed_up then { token list to be deleted }
begin if token_type ≤ inserted then flush_list(start)
else begin delete_token_ref(start); { update reference count }
if token_type = macro then { parameters must be flushed }
while param_ptr > param_start do
begin decr(param_ptr); flush_list(param_stack[param_ptr]);
end;
end;
end
else if token_type = u_template then
if align_state > 500000 then align_state ← 0
else fatal_error("(interwoven_alignment_preambles_are_not_allowed)");
pop_input; check_interrupt;
end;
```

347. Sometimes TeX has read too far and wants to “unscan” what it has seen. The *back_input* procedure takes care of this by putting the token just scanned back into the input stream, ready to be read again. This procedure can be used only if *cur_tok* represents the token to be replaced. Some applications of TeX use this procedure a lot, so it has been slightly optimized for speed.

```
procedure back_input; { undoes one token of input }
var p: pointer; { a token list of length one }
begin while (state = token_list) ∧ (loc = null) ∧ (token_type ≠ v_template) do end_token_list;
{ conserve stack space }
p ← get_avail; info(p) ← cur_tok;
if cur_tok < right_brace_limit then
if cur_tok < left_brace_limit then decr(align_state)
else incr(align_state);
push_input; state ← token_list; start ← p; token_type ← backed_up; loc ← p;
{ that was back_list(p), without procedure overhead }
end;
```

348. ⟨Insert token *p* into TeX’s input 348⟩ ≡

```
begin t ← cur_tok; cur_tok ← p;
if a then
begin p ← get_avail; info(p) ← cur_tok; link(p) ← loc; loc ← p; start ← 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 ← eTeX_ex;
end;
cur_tok ← t;
end
```

This code is used in section 304.

349. The *back_error* routine is used when we want to replace an offending token just before issuing an error message. This routine, like *back_input*, requires that *cur_tok* has been set. We disable interrupts during the call of *back_input* so that the help message won't be lost.

```
procedure back_error; { back up one token and call error }
begin OK_to_interrupt ← false; back_input; OK_to_interrupt ← true; error;
end;

procedure ins_error; { back up one inserted token and call error }
begin OK_to_interrupt ← false; back_input; token_type ← inserted; OK_to_interrupt ← true; error;
end;
```

350. 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 ← in_open; eof_seen[index] ← false;
grp_stack[index] ← cur_boundary; if_stack[index] ← cond_ptr; line_stack[index] ← line; start ← first;
state ← mid_line; name ← 0; { terminal_input is now true }
end;
```

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

```
procedure end_file_reading;
begin first ← start; line ← line_stack[index];
if (name = 18) ∨ (name = 19) then pseudo_close
else if name > 17 then a_close(cur_file); { forget it }
pop_input; decr(in_open);
end;
```

352. In order to keep the stack from overflowing during a long sequence of inserted ‘\show’ commands, the following routine removes completed error-inserted lines from memory.

```
procedure clear_for_error_prompt;
begin while (state ≠ token_list) ∧ terminal_input ∧ (input_ptr > 0) ∧ (loc > limit) do end_file_reading;
print_ln; clear_terminal;
end;
```

353. To get TeX's whole input mechanism going, we perform the following actions.

⟨ Initialize the input routines 353 ⟩ ≡

```
begin input_ptr ← 0; max_in_stack ← 0; in_open ← 0; open_parens ← 0; max_buf_stack ← 0;
grp_stack[0] ← 0; if_stack[0] ← null; param_ptr ← 0; max_param_stack ← 0; first ← buf_size;
repeat buffer[first] ← 0; decr(first);
until first = 0;
scanner_status ← normal; warning_index ← null; first ← 1; state ← new_line; start ← 1; index ← 0;
line ← 0; name ← 0; force_eof ← false; align_state ← 1000000;
if ¬init_terminal then goto final_end;
limit ← last; first ← last + 1; { init_terminal has set loc and last }
end
```

This code is used in section 1517.

354. Getting the next token. The heart of TeX's input mechanism is the *get_next* procedure, which we shall develop in the next few sections of the program. Perhaps we shouldn't actually call it the "heart," however, because it really acts as TeX's eyes and mouth, reading the source files and gobbling them up. And it also helps TeX to regurgitate stored token lists that are to be processed again.

The main duty of *get_next* is to input one token and to set *cur_cmd* and *cur_chr* to that token's command code and modifier. Furthermore, if the input token is a control sequence, the *eqtb* location of that control sequence is stored in *cur_cs*; otherwise *cur_cs* is set to zero.

Underlying this simple description is a certain amount of complexity because of all the cases that need to be handled. However, the inner loop of *get_next* is reasonably short and fast.

When *get_next* is asked to get the next token of a \read line, it sets *cur_cmd* = *cur_chr* = *cur_cs* = 0 in the case that no more tokens appear on that line. (There might not be any tokens at all, if the *end_line_char* has *ignore* as its catcode.)

355. The value of *par_loc* is the *eqtb* address of '\par'. This quantity is needed because a blank line of input is supposed to be exactly equivalent to the appearance of \par; we must set *cur_cs* ← *par_loc* when detecting a blank line.

```
(Global variables 13) +≡
par_loc: pointer; {location of '\par' in eqtb}
par_token: halfword; {token representing '\par'}
```

356. {Put each of TeX's primitives into the hash table 244} +≡

```
primitive("par", par_end, 256); {cf. scan_file_name}
par_loc ← cur_val; par_token ← cs_token_flag + par_loc;
```

357. {Cases of *print_cmd_chr* for symbolic printing of primitives 245} +≡

```
par_end: print_esc("par");
```

358. Before getting into *get_next*, let's consider the subroutine that is called when an '\outer' control sequence has been scanned or when the end of a file has been reached. These two cases are distinguished by *cur_cs*, which is zero at the end of a file.

```
procedure check_outer_validity;
var p: pointer; {points to inserted token list}
q: pointer; {auxiliary pointer}
begin if scanner_status ≠ normal then
begin deletions_allowed ← false; {Back up an outer control sequence so that it can be reread 359};
if scanner_status > skipping then {Tell the user what has run away and try to recover 360}
else begin print_err("Incomplete"); print_cmd_chr(if_test, cur_if);
print(";\uall\uwas\uignored\uafter\u"; print_int(skip_line);
help3("A\uforbidden\ucontrol\usequence\uoccurred\uin\u skipped\u text.");
("This\ukind\uof\uerror\uhappens\uwhen\uyou\u say\u`\\if...\uand\uforget");
("the\u matching\u`\\fi`\uI've\uinserted\ua\u`\\fi`\u; this\u might\u work.");
if cur_cs ≠ 0 then cur_cs ← 0
else help_line[2] ← "The\ufile\uended\uwhile\uI\uwas\u skipping\uconditional\u text.";
cur_tok ← cs_token_flag + frozen_if; ins_error;
end;
deletions_allowed ← true;
end;
end;
```

359. An outer control sequence that occurs in a `\read` will not be reread, since the error recovery for `\read` is not very powerful.

```
< Back up an outer control sequence so that it can be reread 359 > ≡
if cur_cs ≠ 0 then
  begin if (state = token_list) ∨ (name < 1) ∨ (name > 17) then
    begin p ← get_avail; info(p) ← cs_token_flag + cur_cs; back_list(p);
      { prepare to read the control sequence again }
    end;
  cur_cmd ← spacer; cur_chr ← " "; { replace it by a space }
end
```

This code is used in section 358.

360. { Tell the user what has run away and try to recover 360 } ≡

```
begin runaway; { print a definition, argument, or preamble }
if cur_cs = 0 then print_err("File ended")
else begin cur_cs ← 0; print_err("Forbidden control sequence found");
  end;
print("while scanning"); { Print either 'definition' or 'use' or 'preamble' or 'text', and insert
  tokens that should lead to recovery 361 };
print("of"); sprint_cs(warning_index);
help4("I suspect you have forgotten a `}', causing me"
("to read past where you wanted me to stop.")
("I'll try to recover; but if the error is serious,"
("you'd better type `E' or `X' now and fix your file."));
error;
end
```

This code is used in section 358.

361. The recovery procedure can't be fully understood without knowing more about the TeX routines that should be aborted, but we can sketch the ideas here: For a runaway definition or a runaway balanced text we will insert a right brace; for a runaway preamble, we will insert a special `\cr` token and a right brace; and for a runaway argument, we will set `long_state` to `outer_call` and insert `\par`.

```
< Print either 'definition' or 'use' or 'preamble' or 'text', and insert tokens that should lead to
  recovery 361 > ≡
p ← get_avail;
case scanner_status of
  defining: begin print("definition"); info(p) ← right_brace_token + "}";
  end;
  matching: begin print("use"); info(p) ← par_token; long_state ← outer_call;
  end;
  aligning: begin print("preamble"); info(p) ← right_brace_token + "}";
  q ← p; p ← get_avail;
  link(p) ← q; info(p) ← cs_token_flag + frozen_cr; align_state ← -1000000;
  end;
  absorbing: begin print("text"); info(p) ← right_brace_token + "}";
  end;
end; { there are no other cases }
ins_list(p)
```

This code is used in section 360.

362. We need to mention a procedure here that may be called by `get_next`.

```
procedure firm_up_the_line; forward;
```

363. Now we're ready to take the plunge into *get_next* itself. Parts of this routine are executed more often than any other instructions of T_EX.

```

define switch = 25 { a label in get_next }
define start_cs = 26 { another }

procedure get_next; { sets cur_cmd, cur_chr, cur_cs to next token }
label restart, { go here to get the next input token }
  switch, { go here to eat the next character from a file }
  reswitch, { go here to digest it again }
  start_cs, { go here to start looking for a control sequence }
  found, { go here when a control sequence has been found }
  exit; { go here when the next input token has been got }
var k: 0 .. buf_size; { an index into buffer }
  t: halfword; { a token }
  cat: 0 .. max_char_code; { cat_code(cur_chr), usually }
  c, cc: ASCII_code; { constituents of a possible expanded code }
  d: 2 .. 3; { number of excess characters in an expanded code }
begin restart: cur_cs ← 0;
  if state ≠ token_list then { Input from external file, goto restart if no input found 365 }
  else { Input from token list, goto restart if end of list or if a parameter needs to be expanded 379 };
    { If an alignment entry has just ended, take appropriate action 364 };
  exit: end;

```

364. An alignment entry ends when a tab or \cr occurs, provided that the current level of braces is the same as the level that was present at the beginning of that alignment entry; i.e., provided that *align_state* has returned to the value it had after the $\langle u_j \rangle$ template for that entry.

```

{ If an alignment entry has just ended, take appropriate action 364 } ≡
  if cur_cmd ≤ car_ret then
    if cur_cmd ≥ tab_mark then
      if align_state = 0 then { Insert the  $\langle v_j \rangle$  template and goto restart 965 }

```

This code is used in section 363.

```

365. { Input from external file, goto restart if no input found 365 } ≡
  begin switch: if loc ≤ limit then { current line not yet finished }
    begin cur_chr ← buffer[loc]; incr(loc);
    reswitch: cur_cmd ← cat_code(cur_chr); { Change state if necessary, and goto switch if the current
      character should be ignored, or goto reswitch if the current character changes to another 366 };
    end
  else begin state ← new_line;
    { Move to next line of file, or goto restart if there is no next line, or return if a \read line has
      finished 382 };
    check_interrupt; goto switch;
  end;
end

```

This code is used in section 363.

366. The following 48-way switch accomplishes the scanning quickly, assuming that a decent Pascal compiler has translated the code. Note that the numeric values for *mid_line*, *skip_blanks*, and *new_line* are spaced apart from each other by *max_char_code* + 1, so we can add a character's command code to the state to get a single number that characterizes both.

```
define any_state_plus(#) ≡ mid_line + #, skip_blanks + #, new_line + #

⟨ Change state if necessary, and goto switch if the current character should be ignored, or goto reswitch if
    the current character changes to another 366 ⟩ ≡
case state + cur_cmd of
    ⟨ Cases where character is ignored 367 ⟩: goto switch;
    any_state_plus(escape): ⟨ Scan a control sequence and set state ← skip_blanks or mid_line 376 ⟩;
    any_state_plus(active_char): ⟨ Process an active-character control sequence and set state ← mid_line 375 ⟩;
    any_state_plus(sup_mark): ⟨ If this sup_mark starts an expanded character like ^A or ^df, then goto
        reswitch, otherwise set state ← mid_line 374 ⟩;
    any_state_plus(invalid_char): ⟨ Decry the invalid character and goto restart 368 ⟩;
    ⟨ Handle situations involving spaces, braces, changes of state 369 ⟩
    othercases do_nothing
endcases
```

This code is used in section 365.

367. ⟨ Cases where character is ignored 367 ⟩ ≡
any_state_plus(ignore), *skip_blanks + spacer*, *new_line + spacer*

This code is used in section 366.

368. We go to *restart* instead of to *switch*, because *state* might equal *token_list* after the error has been dealt with (cf. *clear_for_error_prompt*).

```
⟨ Decry the invalid character and goto restart 368 ⟩ ≡
begin print_err("Text_line_contains_an_invalid_character");
help2("AFunnySymbolthatIcan'treadhasjustbeeninput.");
("Continue, and I'llforgetthatiteverhappened.");
deletions_allowed ← false; error; deletions_allowed ← true; goto restart;
end
```

This code is used in section 366.

369. define add_delims_to(#) ≡ # + *math_shift*, # + *tab_mark*, # + *mac_param*, # + *sub_mark*, # + *letter*,
 # + *other_char*

```
⟨ Handle situations involving spaces, braces, changes of state 369 ⟩ ≡
mid_line + spacer: ⟨ Enter skip_blanks state, emit a space 371 ⟩;
mid_line + car_ret: ⟨ Finish line, emit a space 370 ⟩;
skip_blanks + car_ret, any_state_plus(comment): ⟨ Finish line, goto switch 372 ⟩;
new_line + car_ret: ⟨ Finish line, emit a \par 373 ⟩;
mid_line + left_brace: incr(align_state);
skip_blanks + left_brace, new_line + left_brace: begin state ← mid_line; incr(align_state);
    end;
mid_line + right_brace: decr(align_state);
skip_blanks + right_brace, new_line + right_brace: begin state ← mid_line; decr(align_state);
    end;
add_delims_to(skip_blanks), add_delims_to(new_line): state ← mid_line;
```

This code is used in section 366.

370. When a character of type *spacer* gets through, its character code is changed to " \sqcup " = $'40$. This means that the ASCII codes for tab and space, and for the space inserted at the end of a line, will be treated alike when macro parameters are being matched. We do this since such characters are indistinguishable on most computer terminal displays.

```
<Finish line, emit a space 370> ≡
begin loc ← limit + 1; cur_cmd ← spacer; cur_chr ← "sqcup";
end
```

This code is used in section 369.

371. The following code is performed only when $cur_cmd = spacer$.

```
<Enter skip_blanks state, emit a space 371> ≡
begin state ← skip_blanks; cur_chr ← "sqcup";
end
```

This code is used in section 369.

372. `<Finish line, goto switch 372>` ≡
`begin loc ← limit + 1; goto switch;`
`end`

This code is used in section 369.

373. `<Finish line, emit a \par 373>` ≡
`begin loc ← limit + 1; cur_cs ← par_loc; cur_cmd ← eq_type(cur_cs); cur_chr ← equiv(cur_cs);`
`if cur_cmd ≥ outer_call then check_outer_validity;`
`end`

This code is used in section 369.

374. Notice that a code like $\wedge\wedge 8$ becomes x if not followed by a hex digit.

```
define is_hex(#) ≡ (((# ≥ "0") ∧ (# ≤ "9")) ∨ ((# ≥ "a") ∧ (# ≤ "f")))
define hex_to_cur_chr ≡
  if c ≤ "9" then cur_chr ← c - "0" else cur_chr ← c - "a" + 10;
  if cc ≤ "9" then cur_chr ← 16 * cur_chr + cc - "0"
  else cur_chr ← 16 * cur_chr + cc - "a" + 10

<If this sup_mark starts an expanded character like  $\wedge\wedge A$  or  $\wedge\wedge df$ , then goto reswitch, otherwise set
state ← mid_line 374> ≡
begin if cur_chr = buffer[loc] then
  if loc < limit then
    begin c ← buffer[loc + 1]; if c < '200' then {yes we have an expanded char}
      begin loc ← loc + 2;
      if is_hex(c) then
        if loc ≤ limit then
          begin cc ← buffer[loc]; if is_hex(cc) then
            begin incr(loc); hex_to_cur_chr; goto reswitch;
            end;
          end;
        end;
      if c < '100' then cur_chr ← c + '100' else cur_chr ← c - '100';
      goto reswitch;
      end;
    end;
  state ← mid_line;
end
```

This code is used in section 366.

375. ⟨ Process an active-character control sequence and set $state \leftarrow mid_line$ 375 ⟩ ≡
begin $cur_cs \leftarrow cur_chr + active_base$; $cur_cmd \leftarrow eq_type(cur_cs)$; $cur_chr \leftarrow equiv(cur_cs)$;
 $state \leftarrow mid_line$;
if $cur_cmd \geq outer_call$ **then** $check_outer_validity$;
end

This code is used in section 366.

376. Control sequence names are scanned only when they appear in some line of a file; once they have been scanned the first time, their $eqtb$ location serves as a unique identification, so TeX doesn't need to refer to the original name any more except when it prints the equivalent in symbolic form.

The program that scans a control sequence has been written carefully in order to avoid the blowups that might otherwise occur if a malicious user tried something like '\catcode`15=0'. The algorithm might look at $buffer[limit + 1]$ but it never looks at $buffer[limit + 2]$.

If expanded characters like '^A' or '^df' appear in or just following a control sequence name, they are converted to single characters in the buffer and the process is repeated, slowly but surely.

⟨ Scan a control sequence and set $state \leftarrow skip_blanks$ or mid_line 376 ⟩ ≡
begin **if** $loc > limit$ **then** $cur_cs \leftarrow null_cs$ { $state$ is irrelevant in this case }
else begin $start_cs: k \leftarrow loc$; $cur_chr \leftarrow buffer[k]$; $cat \leftarrow cat_code(cur_chr)$; $incr(k)$;
if $cat = letter$ **then** $state \leftarrow skip_blanks$
else if $cat = spacer$ **then** $state \leftarrow skip_blanks$
else $state \leftarrow mid_line$;
if ($cat = letter$) $\wedge (k \leq limit)$ **then** ⟨ Scan ahead in the buffer until finding a nonletter; if an expanded code is encountered, reduce it and **goto** $start_cs$; otherwise if a multiletter control sequence is found, adjust cur_cs and loc , and **goto** $found$ 378 ⟩
else ⟨ If an expanded code is present, reduce it and **goto** $start_cs$ 377 ⟩;
 $cur_cs \leftarrow single_base + buffer[loc]$; $incr(loc)$;
end;
found: $cur_cmd \leftarrow eq_type(cur_cs)$; $cur_chr \leftarrow equiv(cur_cs)$;
if $cur_cmd \geq outer_call$ **then** $check_outer_validity$;
end

This code is used in section 366.

377. Whenever we reach the following piece of code, we will have $cur_chr = buffer[k - 1]$ and $k \leq limit + 1$ and $cat = cat_code(cur_chr)$. If an expanded code like $\wedge\wedge A$ or $\wedge\wedge df$ appears in $buffer[(k - 1) \dots (k + 1)]$ or $buffer[(k - 1) \dots (k + 2)]$ we will store the corresponding code in $buffer[k - 1]$ and shift the rest of the buffer left two or three places.

```

⟨ If an expanded code is present, reduce it and goto start_cs 377 ⟩ ≡
begin if buffer[k] = cur_chr then if cat = sup_mark then if k < limit then
    begin c ← buffer[k + 1]; if c < '200 then { yes, one is indeed present }
        begin d ← 2;
        if is_hex(c) then if k + 2 ≤ limit then
            begin cc ← buffer[k + 2]; if is_hex(cc) then incr(d);
            end;
        if d > 2 then
            begin hex_to_cur_chr; buffer[k - 1] ← cur_chr;
            end
        else if c < '100 then buffer[k - 1] ← c + '100
        else buffer[k - 1] ← c - '100;
        limit ← limit - d; first ← first - d;
    while k ≤ limit do
        begin buffer[k] ← buffer[k + d]; incr(k);
        end;
    goto start_cs;
    end;
end;
end;
end

```

This code is used in sections 376 and 378.

378. ⟨ Scan ahead in the buffer until finding a nonletter; if an expanded code is encountered, reduce it and goto start_cs; otherwise if a multiletter control sequence is found, adjust cur_cs and loc, and goto found 378 ⟩ ≡

```

begin repeat cur_chr ← buffer[k]; cat ← cat_code(cur_chr); incr(k);
until (cat ≠ letter) ∨ (k > limit);
⟨ If an expanded code is present, reduce it and goto start_cs 377 ⟩;
if cat ≠ letter then decr(k); { now k points to first nonletter }
if k > loc + 1 then { multiletter control sequence has been scanned }
    begin cur_cs ← id_lookup(loc, k - loc); loc ← k; goto found;
    end;
end

```

This code is used in section 376.

379. Let's consider now what happens when *get_next* is looking at a token list.

```

⟨ Input from token list, goto restart if end of list or if a parameter needs to be expanded 379 ⟩ ≡
  if loc ≠ null then { list not exhausted }
    begin t ← info(loc); loc ← link(loc); { move to next }
    if t ≥ cs_token_flag then { a control sequence token }
      begin cur_cs ← t - cs_token_flag; cur_cmd ← eq_type(cur_cs); cur_chr ← equiv(cur_cs);
      if cur_cmd ≥ outer_call then
        if cur_cmd = dont_expand then ⟨ Get the next token, suppressing expansion 380 ⟩
        else check_outer_validity;
      end
    else begin cur_cmd ← t div '400; cur_chr ← t mod '400;
      case cur_cmd of
        left_brace: incr(align_state);
        right_brace: decr(align_state);
        out_param: ⟨ Insert macro parameter and goto restart 381 ⟩;
        othercases do_nothing
      endcases;
    end;
  end
else begin { we are done with this token list }
  end_token_list; goto restart; { resume previous level }
end

```

This code is used in section 363.

380. The present point in the program is reached only when the *expand* routine has inserted a special marker into the input. In this special case, *info(loc)* is known to be a control sequence token, and *link(loc) = null*.

```

define no_expand_flag = 257 { this characterizes a special variant of relax }
⟨ Get the next token, suppressing expansion 380 ⟩ ≡
  begin cur_cs ← info(loc) - cs_token_flag; loc ← null;
  cur_cmd ← eq_type(cur_cs); cur_chr ← equiv(cur_cs);
  if cur_cmd > max_command then
    begin cur_cmd ← relax; cur_chr ← no_expand_flag;
    end;
  end

```

This code is used in section 379.

381. ⟨ Insert macro parameter and goto restart 381 ⟩ ≡

```

begin begin_token_list(param_stack[param_start + cur_chr - 1], parameter); goto restart;
end

```

This code is used in section 379.

382. All of the easy branches of *get_next* have now been taken care of. There is one more branch.

```

define end_line_char_inactive ≡ (end_line_char < 0) ∨ (end_line_char > 255)
⟨Move to next line of file, or goto restart if there is no next line, or return if a \read line has
finished 382⟩ ≡
if name > 17 then ⟨Read next line of file into buffer, or goto restart if the file has ended 384⟩
else begin if ¬terminal_input then { \read line has ended }
  begin cur_cmd ← 0; cur_chr ← 0; return;
  end;
  if input_ptr > 0 then { text was inserted during error recovery }
    begin end_file_reading; goto restart; { resume previous level }
    end;
  if selector < log_only then open_log_file;
  if interaction > nonstop_mode then
    begin if end_line_char_inactive then incr(limit);
    if limit = start then { previous line was empty }
      print_nl("Please_type_a_command_or_say_\`end`");
      print_ln; first ← start; prompt_input("*"); { input on-line into buffer }
      limit ← last;
    if end_line_char_inactive then decr(limit)
    else buffer[limit] ← end_line_char;
    first ← limit + 1; loc ← start;
    end
  else fatal_error("***_job_aborted,_no_legal_\end_found");
    { nonstop mode, which is intended for overnight batch processing, never waits for on-line input }
  end
```

This code is used in section 365.

383. The global variable *force_eof* is normally *false*; it is set *true* by an \endinput command.

```

⟨Global variables 13⟩ +≡
force_eof: boolean; { should the next \input be aborted early? }
```

384. ⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 384 ⟩ ≡

```

begin incr(line); first ← start;
if  $\neg force\_eof$  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)  $\wedge \neg eof\_seen[index]$  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)  $\wedge \neg eof\_seen[index]$  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)  $\vee$  (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 382.

385. If the user has set the *pausing* parameter to some positive value, and if nonstop mode has not been selected, each line of input is displayed on the terminal and the transcript file, followed by ' $=>$ '. TeX waits for a response. If the response is simply *carriage_return*, the line is accepted as it stands, otherwise the line typed is used instead of the line in the file.

```
procedure firm_up_the_line;
  var k: 0 .. buf_size; { an index into buffer }
  begin limit ← last;
  if pausing > 0 then
    if interaction > nonstop_mode then
      begin wake_up_terminal; print_ln;
      if start < limit then
        for k ← start to limit - 1 do print(buffer[k]);
      first ← limit; prompt_input("=>"); { wait for user response }
      if last > first then
        begin for k ← first to last - 1 do { move line down in buffer }
          buffer[k + start - first] ← buffer[k];
        limit ← start + last - first;
        end;
      end;
    end;
  end;
```

386. Since *get_next* is used so frequently in TeX, it is convenient to define three related procedures that do a little more:

get_token not only sets *cur_cmd* and *cur_chr*, it also sets *cur_tok*, a packed halfword version of the current token.

get_x_token, meaning "get an expanded token," is like *get_token*, but if the current token turns out to be a user-defined control sequence (i.e., a macro call), or a conditional, or something like \topmark or \expandafter or \csname, it is eliminated from the input by beginning the expansion of the macro or the evaluation of the conditional.

x_token is like *get_x_token* except that it assumes that *get_next* has already been called.

In fact, these three procedures account for almost every use of *get_next*.

387. No new control sequences will be defined except during a call of *get_token*, or when \csname compresses a token list, because *no_new_control_sequence* is always *true* at other times.

```
procedure get_token; { sets cur_cmd, cur_chr, cur_tok }
  begin no_new_control_sequence ← false; get_next; no_new_control_sequence ← true;
  if cur_cs = 0 then cur_tok ← (cur_cmd * '400) + cur_chr
  else cur_tok ← cs_token_flag + cur_cs;
  end;
```

388. 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 415 ⟩
⟨ Declare the procedure called insert_relax 405 ⟩
⟨ Declare ε-TeX procedures for expanding 1752 ⟩
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” }
b: boolean; { keep track of nested csnames }
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  $\leftarrow$  cur_val; cvt_backup  $\leftarrow$  cur_val_level; radix_backup  $\leftarrow$  radix; co_backup  $\leftarrow$  cur_order;
backup_backup  $\leftarrow$  link(backup_head);
reswitch: if cur_cmd  $<$  call then ⟨ Expand a nonmacro 391 ⟩
else if cur_cmd  $<$  end_template then macro_call
else ⟨ Insert a token containing frozen_endv 401 ⟩;
cur_val  $\leftarrow$  cv_backup; cur_val_level  $\leftarrow$  cvt_backup; radix  $\leftarrow$  radix_backup; cur_order  $\leftarrow$  co_backup;
link(backup_head)  $\leftarrow$  backup_backup;
end;

```

389. ⟨ Global variables 13 ⟩ +≡

is_in_cname: boolean;

390. ⟨ Set initial values of key variables 21 ⟩ +≡

is_in_cname \leftarrow false;

391. $\langle \text{Expand a nonmacro } 391 \rangle \equiv$

```

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 412 ⟩;
  expand_after: if cur_chr = 0 then ⟨ Expand the token after the next token 392 ⟩
    else ⟨ Negate a boolean conditional and goto reswitch 1765 ⟩;
  no_expand: if cur_chr = 0 then ⟨ Suppress expansion of the next token 393 ⟩
    else ⟨ Implement \pdfprimitive 394 ⟩;
  cs_name: ⟨ Manufacture a control sequence name 398 ⟩;
  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 536 ⟩;
  input: ⟨ Initiate or terminate input from a file 404 ⟩;
  othercases ⟨ Complain about an undefined macro 396 ⟩
endcases;
end

```

This code is used in section 388.

392. It takes only a little shuffling to do what T_EX calls \expandafter.

$\langle \text{Expand the token after the next token } 392 \rangle \equiv$

```

begin get_token; t ← cur_tok; get_token;
if cur_cmd > max_command then expand else back_input;
cur_tok ← t; back_input;
end

```

This code is used in section 391.

393. The implementation of \noexpand is a bit trickier, because it is necessary to insert a special ‘*dont_expand*’ marker into T_EX’s reading mechanism. This special marker is processed by *get_next*, but it does not slow down the inner loop.

Since \outer macros might arise here, we must also clear the *scanner_status* temporarily.

$\langle \text{Suppress expansion of the next token } 393 \rangle \equiv$

```

begin save_scanner_status ← scanner_status; scanner_status ← normal; get_token;
scanner_status ← save_scanner_status; t ← cur_tok; back_input;
{ now start and loc point to the backed-up token t }
if t ≥ cs_token_flag then
  begin p ← get_avail; info(p) ← cs_token_flag + frozen_dont_expand; link(p) ← loc; start ← p;
  loc ← p;
  end;
end

```

This code is used in section 391.

394. The `\pdfprimitive` handling. If the primitive meaning of the next token is an expandable command, it suffices to replace the current token with the primitive one and restart `expand`.

Otherwise, the token we just read has to be pushed back, as well as a token matching the internal form of `\pdfprimitive`, that is sneaked in as an alternate form of `ignore_spaces`.

Simply pushing back a token that matches the correct internal command does not work, because approach would not survive roundtripping to a temporary file.

```
(Implement \pdfprimitive 394) ≡
begin save_scanner_status ← scanner_status; scanner_status ← normal; get_token;
scanner_status ← save_scanner_status;
if cur_cs < hash_base then cur_cs ← prim_lookup(cur_cs - single_base)
else cur_cs ← prim_lookup(text(cur_cs));
if cur_cs ≠ undefined_primitive then
begin t ← prim_eq_type(cur_cs);
if t > max_command then
begin cur_cmd ← t; cur_chr ← prim_equiv(cur_cs); cur_tok ← (cur_cmd * '400) + cur_chr;
cur_cs ← 0; goto reswitch;
end
else begin back_input; { now loc and start point to a one-item list }
p ← get_avail; info(p) ← cs_token_flag + frozen_primitive; link(p) ← loc; loc ← p; start ← p;
end;
end;
end;
```

This code is used in section 391.

395. This block deals with unexpandable `\primitive` appearing at a spot where an integer or an internal values should have been found. It fetches the next token then resets `cur_cmd`, `cur_cs`, and `cur_tok`, based on the primitive value of that token. No expansion takes place, because the next token may be all sorts of things. This could trigger further expansion creating new errors.

```
(Reset cur_tok for unexpandable primitives, goto restart 395) ≡
begin get_token;
if cur_cs < hash_base then cur_cs ← prim_lookup(cur_cs - single_base)
else cur_cs ← prim_lookup(text(cur_cs));
if cur_cs ≠ undefined_primitive then
begin cur_cmd ← prim_eq_type(cur_cs); cur_chr ← prim_equiv(cur_cs);
cur_cs ← prim_eqtb_base + cur_cs; cur_tok ← cs_token_flag + cur_cs;
end
else begin cur_cmd ← relax; cur_chr ← 0; cur_tok ← cs_token_flag + frozen_relax;
cur_cs ← frozen_relax;
end;
goto restart;
end
```

This code is used in sections 439 and 466.

396. ⟨Complain about an undefined macro 396⟩ ≡

```
begin print_err("Undefined control sequence");
help5 ("The control sequence at the end of the top line")
("of your error message was never \def`ed. If you have")
("misspelled it (e.g., `\\hbox`), type `I` and the correct")
("spelling (e.g., `I\\hbox`). Otherwise just continue,")
("and I'll forget about whatever was undefined."); error;
end
```

This code is used in section 391.

397. The *expand* procedure and some other routines that construct token lists find it convenient to use the following macros, which are valid only if the variables *p* and *q* are reserved for token-list building.

```
define store_new_token (#) ≡
begin q ← get_avail; link(p) ← q; info(q) ← #; p ← q; { link(p) is null }
end
define fast_store_new_token (#) ≡
begin fast_get_avail(q); link(p) ← q; info(q) ← #; p ← q; { link(p) is null }
end
```

398. ⟨Manufacture a control sequence name 398⟩ ≡

```
begin r ← get_avail; p ← r; { head of the list of characters }
b ← is_in_csname; is_in_csname ← true;
repeat get_x_token;
  if cur_cs = 0 then store_new_token(cur_tok);
until cur_cs ≠ 0;
if cur_cmd ≠ end_cs_name then ⟨Complain about missing \\endcsname 399⟩;
is_in_csname ← b; { Look up the characters of list r in the hash table, and set cur_cs 400 };
flush_list(r);
if eq_type(cur_cs) = undefined_cs then
  begin eq_define(cur_cs, relax, 256); { N.B.: The save_stack might change }
  end; { the control sequence will now match '\\relax' }
cur_tok ← cur_cs + cs_token_flag; back_input;
end
```

This code is used in section 391.

399. ⟨Complain about missing \\endcsname 399⟩ ≡

```
begin print_err("Missing "); print_esc("endcsname"); print(" inserted");
help2 ("The control sequence marked <to be read again> should")
("not appear between \\csname and \\endcsname."); back_error;
end
```

This code is used in sections 398 and 1767.

400. ⟨ Look up the characters of list r in the hash table, and set cur_cs 400 ⟩ ≡

```

 $j \leftarrow first; p \leftarrow link(r);$ 
while  $p \neq null$  do
  begin if  $j \geq max\_buf\_stack$  then
    begin  $max\_buf\_stack \leftarrow j + 1;$ 
    if  $max\_buf\_stack = buf\_size$  then overflow("buffer_size", buf_size);
    end;
     $buffer[j] \leftarrow info(p) \bmod 400; incr(j); p \leftarrow link(p);$ 
  end;
if  $j > first + 1$  then
  begin  $no\_new\_control\_sequence \leftarrow false; cur\_cs \leftarrow id\_lookup(first, j - first);$ 
   $no\_new\_control\_sequence \leftarrow true;$ 
  end
else if  $j = first$  then  $cur\_cs \leftarrow null\_cs$  { the list is empty }
  else  $cur\_cs \leftarrow single\_base + buffer[first]$  { the list has length one }
```

This code is used in section 398.

401. An *end_template* command is effectively changed to an *endv* command by the following code. (The reason for this is discussed below; the *frozen_end_template* at the end of the template has passed the *check_outer_validity* test, so its mission of error detection has been accomplished.)

⟨ Insert a token containing *frozen_endv* 401 ⟩ ≡

```

begin  $cur\_tok \leftarrow cs\_token\_flag + frozen\_endv; back\_input;$ 
end
```

This code is used in section 388.

402. The processing of *\input* involves the *start_input* subroutine, which will be declared later; the processing of *\endinput* is trivial.

⟨ Put each of TeX's primitives into the hash table 244 ⟩ +≡

```

primitive("input", input, 0);
primitive("endinput", input, 1);
```

403. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 245 ⟩ +≡

```

: if  $chr\_code = 0$  then print_esc("input")
  ⟨ Cases of input for print_cmd_chr 1748 ⟩
else print_esc("endinput");
```

404. ⟨ Initiate or terminate input from a file 404 ⟩ ≡

```

if  $cur\_chr = 1$  then force_eof ← true
  ⟨ Cases for input 1749 ⟩
else if name_in_progress then insert_relax
  else start_input
```

This code is used in section 391.

405. Sometimes the expansion looks too far ahead, so we want to insert a harmless *\relax* into the user's input.

⟨ Declare the procedure called *insert_relax* 405 ⟩ ≡

```

procedure insert_relax;
  begin  $cur\_tok \leftarrow cs\_token\_flag + cur\_cs; back\_input; cur\_tok \leftarrow cs\_token\_flag + frozen\_relax; back\_input;$ 
   $token\_type \leftarrow inserted;$ 
  end;
```

This code is used in section 388.

406. Here is a recursive procedure that is TeX's usual way to get the next token of input. It has been slightly optimized to take account of common cases.

```
procedure get_x_token; { sets cur_cmd, cur_chr, cur_tok, and expands macros }
  label restart, done;
  begin restart: get_next;
    if cur_cmd ≤ max_command then goto done;
    if cur_cmd ≥ call then
      if cur_cmd < end_template then macro_call
      else begin cur_cs ← frozen_endv; cur_cmd ← endv; goto done; { cur_chr = null_list }
        end
      else expand;
      goto restart;
  done: if cur_cs = 0 then cur_tok ← (cur_cmd * '400) + cur_chr
    else cur_tok ← cs_token_flag + cur_cs;
  end;
```

407. The *get_x_token* procedure is essentially equivalent to two consecutive procedure calls: *get_next*; *x_token*.

```
procedure x_token; { get_x_token without the initial get_next }
  begin while cur_cmd > max_command do
    begin expand; get_next;
    end;
    if cur_cs = 0 then cur_tok ← (cur_cmd * '400) + cur_chr
    else cur_tok ← cs_token_flag + cur_cs;
  end;
```

408. 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 }
```

409. ⟨ Set initial values of key variables 21 ⟩ +≡

```
top_mark ← null; first_mark ← null; bot_mark ← null; split_first_mark ← null; split_bot_mark ← null;
```

410. \langle Put each of TeX's primitives into the hash table 244 $\rangle + \equiv$

```
primitive("topmark", top_bot_mark, top_mark_code);
primitive("firstmark", top_bot_mark, first_mark_code);
primitive("botmark", top_bot_mark, bot_mark_code);
primitive("splitfirstmark", top_bot_mark, split_first_mark_code);
primitive("splitbotmark", top_bot_mark, split_bot_mark_code);
```

411. \langle Cases of *print_cmd_chr* for symbolic printing of primitives 245 $\rangle + \equiv$

```
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;
```

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

\langle Insert the appropriate mark text into the scanner 412 $\rangle \equiv$

```
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 1824 ⟩;
if cur_ptr ≠ null then begin_token_list(cur_ptr, mark_text);
end
```

This code is used in section 391.

413. Now let's consider *macro_call* itself, which is invoked when TeX is scanning a control sequence whose *cur_cmd* is either *call*, *long_call*, *outer_call*, or *long_outer_call*. The control sequence definition appears in the token list whose reference count is in location *cur_chr* of *mem*.

The global variable *long_state* will be set to *call* or to *long_call*, depending on whether or not the control sequence disallows *\par* in its parameters. The *get_next* routine will set *long_state* to *outer_call* and emit *\par*, if a file ends or if an *\outer* control sequence occurs in the midst of an argument.

\langle Global variables 13 $\rangle + \equiv$
long_state: *call* .. *long_outer_call*; { governs the acceptance of *\par* }

414. The parameters, if any, must be scanned before the macro is expanded. Parameters are token lists without reference counts. They are placed on an auxiliary stack called *pstack* while they are being scanned, since the *param_stack* may be losing entries during the matching process. (Note that *param_stack* can't be gaining entries, since *macro_call* is the only routine that puts anything onto *param_stack*, and it is not recursive.)

\langle Global variables 13 $\rangle + \equiv$
pstack: array [0 .. 8] of pointer; { arguments supplied to a macro }

415. 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 415 > ≡
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 427 };
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 417 };
    { Feed the macro body and its parameters to the scanner 416 };
exit: scanner_status ← save_scanner_status; warning_index ← save_warning_index;
end;
```

This code is used in section 388.

416. Before we put a new token list on the input stack, it is wise to clean off all token lists that have recently been depleted. Then a user macro that ends with a call to itself will not require unbounded stack space.

```
< Feed the macro body and its parameters to the scanner 416 > ≡
while (state = token_list) ∧ (loc = null) ∧ (token_type ≠ v_template) do end_token_list;
    { conserve stack space }
begin_token_list(ref_count, macro); name ← warning_index; loc ← link(r);
if n > 0 then
    begin if param_ptr + n > max_param_stack then
        begin max_param_stack ← param_ptr + n;
            if max_param_stack > param_size then overflow("parameter_stack_size", param_size);
            end;
        for m ← 0 to n – 1 do param_stack[param_ptr + m] ← pstack[m];
        param_ptr ← param_ptr + n;
    end
```

This code is used in section 415.

417. At this point, the reader will find it advisable to review the explanation of token list format that was presented earlier, since many aspects of that format are of importance chiefly in the *macro_call* routine.

The token list might begin with a string of compulsory tokens before the first *match* or *end_match*. In that case the macro name is supposed to be followed by those tokens; the following program will set *s* = *null* to represent this restriction. Otherwise *s* will be set to the first token of a string that will delimit the next parameter.

```
< Scan the parameters and make link(r) point to the macro body; but return if an illegal \par is
detected 417 > ≡
begin scanner_status ← matching; unbalance ← 0; long_state ← eq_type(cur_cs);
if long_state ≥ outer_call then long_state ← long_state - 2;
repeat link(temp_head) ← null;
  if (info(r) > match_token + 255) ∨ (info(r) < match_token) then s ← null
  else begin match_chr ← info(r) - match_token; s ← link(r); r ← s; p ← temp_head; m ← 0;
    end;
< Scan a parameter until its delimiter string has been found; or, if s = null, simply scan the delimiter
string 418 >;
  { now info(r) is a token whose command code is either match or end_match }
until info(r) = end_match_token;
end
```

This code is used in section 415.

418. If *info*(*r*) is a *match* or *end_match* command, it cannot be equal to any token found by *get_token*. Therefore an undelimited parameter—i.e., a *match* that is immediately followed by *match* or *end_match*—will always fail the test '*cur_tok* = *info*(*r*)' in the following algorithm.

```
< Scan a parameter until its delimiter string has been found; or, if s = null, simply scan the delimiter
string 418 > ≡
continue: get_token; { set cur_tok to the next token of input }
if cur_tok = info(r) then < Advance r; goto found if the parameter delimiter has been fully matched,
otherwise goto continue 420 >;
< Contribute the recently matched tokens to the current parameter, and goto continue if a partial match
is still in effect; but abort if s = null 423 >;
if cur_tok = par_token then
  if long_state ≠ long_call then < Report a runaway argument and abort 422 >;
  if cur_tok < right_brace_limit then
    if cur_tok < left_brace_limit then < Contribute an entire group to the current parameter 425 >
    else < Report an extra right brace and goto continue 421 >
  else < Store the current token, but goto continue if it is a blank space that would become an undelimited
parameter 419 >;
incr(m);
if info(r) > end_match_token then goto continue;
if info(r) < match_token then goto continue;
found: if s ≠ null then < Tidy up the parameter just scanned, and tuck it away 426 >
```

This code is used in section 417.

419. ⟨ Store the current token, but **goto** *continue* if it is a blank space that would become an undelimited parameter 419 ⟩ ≡

```
begin if cur_tok = space_token then
  if info(r) ≤ end_match_token then
    if info(r) ≥ match_token then goto continue;
  store_new_token(cur_tok);
end
```

This code is used in section 418.

420. A slightly subtle point arises here: When the parameter delimiter ends with ‘#{}’, the token list will have a left brace both before and after the *end_match*. Only one of these should affect the *align_state*, but both will be scanned, so we must make a correction.

⟨ Advance *r*; **goto** *found* if the parameter delimiter has been fully matched, otherwise **goto** *continue* 420 ⟩ ≡

```
begin r ← link(r);
if (info(r) ≥ match_token) ∧ (info(r) ≤ end_match_token) then
  begin if cur_tok < left_brace_limit then decr(align_state);
  goto found;
end
else goto continue;
end
```

This code is used in section 418.

421. ⟨ Report an extra right brace and **goto** *continue* 421 ⟩ ≡

```
begin back_input; print_err("Argument of "); sprint_cs(warning_index); print(" has an extra }");
help6("I've run across a `}` that doesn't seem to match anything.")
("For example, `\\def\\a#1{...}` and `\\a` would produce")
("this error. If you simply proceed now, the `\\par` that")
("I've just inserted will cause me to report a runaway")
("argument that might be the root of the problem. But if")
("your `}` was spurious, just type `2` and it will go away."); incr(align_state);
long_state ← call; cur_tok ← par_token; ins_error; goto continue;
end { a white lie; the \\par won't always trigger a runaway }
```

This code is used in section 418.

422. If *long_state* = *outer_call*, a runaway argument has already been reported.

⟨ Report a runaway argument and abort 422 ⟩ ≡

```
begin if long_state = call then
  begin runaway; print_err("Paragraph ended before "); sprint_cs(warning_index);
  print(" was complete");
  help3("I suspect you've forgotten a `}` causing me to apply this")
  ("control sequence to too much text. How can we recover?")
  ("My plan is to forget the whole thing and hope for the best."); back_error;
end;
pstack[n] ← link(temp_head); align_state ← align_state - unbalance;
for m ← 0 to n do flush_list(pstack[m]);
return;
end
```

This code is used in sections 418 and 425.

423. When the following code becomes active, we have matched tokens from s to the predecessor of r , and we have found that $\text{cur_tok} \neq \text{info}(r)$. An interesting situation now presents itself: If the parameter is to be delimited by a string such as ‘ab’, and if we have scanned ‘aa’, we want to contribute one ‘a’ to the current parameter and resume looking for a ‘b’. The program must account for such partial matches and for others that can be quite complex. But most of the time we have $s = r$ and nothing needs to be done.

Incidentally, it is possible for $\backslash\text{par}$ tokens to sneak in to certain parameters of non- $\backslash\text{long}$ macros. For example, consider a case like ‘ $\backslash\text{def}\backslash a\#1\backslash\text{par}!\{\dots\}$ ’ where the first $\backslash\text{par}$ is not followed by an exclamation point. In such situations it does not seem appropriate to prohibit the $\backslash\text{par}$, so T_EX keeps quiet about this bending of the rules.

```
<Contribute the recently matched tokens to the current parameter, and goto continue if a partial match is
still in effect; but abort if  $s = \text{null}$  423> ≡
if  $s \neq r$  then
  if  $s = \text{null}$  then <Report an improper use of the macro and abort 424>
  else begin  $t \leftarrow s$ ;
    repeat store_new_token(info( $t$ )); incr( $m$ );  $u \leftarrow \text{link}(t)$ ;  $v \leftarrow s$ ;
      loop begin if  $u = r$  then
        if  $\text{cur\_tok} \neq \text{info}(v)$  then goto done
        else begin  $r \leftarrow \text{link}(v)$ ; goto continue;
        end;
        if  $\text{info}(u) \neq \text{info}(v)$  then goto done;
         $u \leftarrow \text{link}(u)$ ;  $v \leftarrow \text{link}(v)$ ;
        end;
      done:  $t \leftarrow \text{link}(t)$ ;
      until  $t = r$ ;
       $r \leftarrow s$ ; { at this point, no tokens are recently matched }
    end
```

This code is used in section 418.

424. <Report an improper use of the macro and abort 424> ≡

```
begin print_err("Use\uof\u"); sprint_cs(warning_index); print(" \udoesn't \umatch \uits \udefinition");
help4("If \you \say, \e.g., \def\al{\dots}, \then \you \must \always")
("put \u 1 \uafter \u'a', \usince \ucontrol \usequence \unames \uare")
("made \uup \uof \uletters \uonly. \uThe \umacro \uhere \uhas \unot \ubeen")
("followed \uby \uthe \urequired \ustuff, \uso \I'm \uignoring \uit."); error; return;
end
```

This code is used in section 423.

425. <Contribute an entire group to the current parameter 425> ≡

```
begin unbalance ← 1;
loop begin fast_store_new_token(cur_tok); get_token;
  if cur_tok = par_token then
    if long_state ≠ long_call then <Report a runaway argument and abort 422>;
  if cur_tok < right_brace_limit then
    if cur_tok < left_brace_limit then incr(unbalance)
    else begin decr(unbalance);
      if unbalance = 0 then goto done1;
      end;
    end;
  end;
done1: rbrace_ptr ← p; store_new_token(cur_tok);
end
```

This code is used in section 418.

426. If the parameter consists of a single group enclosed in braces, we must strip off the enclosing braces. That's why *rbrace_ptr* was introduced.

```
< Tidy up the parameter just scanned, and tuck it away 426 > ≡
begin if (m = 1) ∧ (info(p) < right_brace_limit) then
  begin link(rbrace_ptr) ← null; free_avail(p); p ← link(temp_head); pstack[n] ← link(p); free_avail(p);
  end
else pstack[n] ← link(temp_head);
incr(n);
if tracing_macros > 0 then
  begin begin_diagnostic; print_nl(match_chr); print_int(n); print("<-");
  show_token_list(pstack[n - 1], null, 1000); end_diagnostic(false);
  end;
end
```

This code is used in section 418.

427. < Show the text of the macro being expanded 427 > ≡

```
begin begin_diagnostic; print_ln; print_cs(warning_index); token_show(ref_count);
end_diagnostic(false);
end
```

This code is used in section 415.

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

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

```
procedure scan_left_brace; { reads a mandatory left_brace }
begin <Get the next non-blank non-relax non-call token 430>;
if cur_cmd ≠ left_brace then
  begin print_err("Missing_{_ inserted");
  help4 ("A_left_brace_was_mandatory_here,_so_I've_put_one_in.")
  ("You_might_want_to_delete_and/or_insert_some_corrections")
  ("so_that_I_will_find_a_matching_right_brace_soon.")
  ("If_you're_confused_by_all_this,_try_ttyping_{_now."); back_error;
  cur_tok ← left_brace_token + "{"; cur_cmd ← left_brace; cur_chr ← "{"; incr(align_state);
  end;
end;
```

430. <Get the next non-blank non-relax non-call token 430> ≡

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

This code is used in sections 429, 1256, 1262, 1329, 1338, 1389, 1404, and 1448.

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

```
procedure scan_optional_equals;
begin <Get the next non-blank non-call token 432>;
if cur_tok ≠ other_token + "=" then back_input;
end;
```

432. <Get the next non-blank non-call token 432> ≡

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

This code is used in sections 431, 467, 481, 529, 552, 604, 1223, 1769, 1784, and 1785.

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

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

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

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

```
procedure mu_error;
begin print_err("Incompatible_glue_units");
help1("I'm going to assume that 1mu=1pt when they're mixed."); error;
end;
```

435. 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 459 }
{ Declare ε-TEx procedures for scanning 1682 }
{ Declare procedures that scan font-related stuff 604 }
```

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

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

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

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

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

437. 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 244 ⟩ +≡
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);
```

438. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 245 ⟩ +≡
register: ⟨ Cases of *register* for *print_cmd_chr* 1832 ⟩;

439. 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, restart;
var m: halfword; { chr_code part of the operand token }
n, k: integer; { accumulators }
q, r: pointer; { general purpose indices }
tx: pointer; { effective tail node }
i: four_quarters; { character info }
p: 0 .. nest_size; { index into nest }

begin restart: m ← cur_chr;
case cur_cmd of
def_code: ⟨Fetch a character code from some table 440⟩;
toks_register, assign_toks, def_family, set_font, def_font, letterspace_font, pdf_copy_font: ⟨Fetch a token list
or font identifier, provided that level = tok_val 441⟩;
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 444⟩;
set_prev_graf: ⟨Fetch the prev_graf 448⟩;
set_page_int: ⟨Fetch the dead_cycles or the insert_penalties 445⟩;
set_page_dimen: ⟨Fetch something on the page_so_far 447⟩;
set_shape: ⟨Fetch the par_shape size 449⟩;
set_box_dimen: ⟨Fetch a box dimension 446⟩;
char_given, math_given: scanned_result (cur_chr)(int_val);
assign_font_dimen: ⟨Fetch a font dimension 451⟩;
assign_font_int: ⟨Fetch a font integer 452⟩;
register: ⟨Fetch a register 453⟩;
last_item: ⟨Fetch an item in the current node, if appropriate 450⟩;
ignore_spaces: { trap unexpandable primitives }
    if cur_chr = 1 then ⟨Reset cur_tok for unexpandable primitives, goto restart 395⟩;
    othercases ⟨Complain that \the can't do this; give zero result 454⟩
endcases;
while cur_val_level > level do ⟨Convert cur_val to a lower level 455⟩;
    ⟨Fix the reference count, if any, and negate cur_val if negative 456⟩;
exit: end;

```

440. ⟨Fetch a character code from some table 440⟩ ≡
begin scan_char_num;
if m = math_code_base **then** scanned_result (ho(math_code(cur_val)))(int_val)
else if m < math_code_base **then** scanned_result (equiv(m + cur_val))(int_val)
else scanned_result (eqtb[m + cur_val].int)(int_val);
end

This code is used in section 439.

441. ⟨ Fetch a token list or font identifier, provided that $level = tok_val$ 441 ⟩ ≡

```
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 439.

442. 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`. pdfTeX adds the codes for its extensions: `pdftex_version_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 \lastnode type }
define input_line_no_code = glue_val + 2 { code for \inputlineno }
define badness_code = input_line_no_code + 1 { code for \badness }

define pdftex_first_rint_code = badness_code + 1 { base for pdfTeX's command codes }
define pdftex_version_code = pdftex_first_rint_code + 0 { code for \pdftexversion }
define pdf_last_obj_code = pdftex_first_rint_code + 1 { code for \pdflastobj }
define pdf_last_xform_code = pdftex_first_rint_code + 2 { code for \pdflastxform }
define pdf_last_ximage_code = pdftex_first_rint_code + 3 { code for \pdflastximage }
define pdf_last_ximage_pages_code = pdftex_first_rint_code + 4 { code for \pdflastximagepages }
define pdf_last_annot_code = pdftex_first_rint_code + 5 { code for \pdflastannot }
define pdf_last_x_pos_code = pdftex_first_rint_code + 6 { code for \pdflastxpos }
define pdf_last_y_pos_code = pdftex_first_rint_code + 7 { code for \pdflastypos }
define pdf_retval_code = pdftex_first_rint_code + 8 { global multi-purpose return value }
define pdf_last_ximage_colord depth_code = pdftex_first_rint_code + 9
    { code for \pdflastximagecolor depth }

define elapsed_time_code = pdftex_first_rint_code + 10 { code for \pdfelapsedtime }
define pdf_shell_escape_code = pdftex_first_rint_code + 11 { code for \pdfshellescape }
define random_seed_code = pdftex_first_rint_code + 12 { code for \pdfrandomseed }
define pdf_last_link_code = pdftex_first_rint_code + 13 { code for \pdflastlink }
define pdftex_last_item_codes = pdftex_first_rint_code + 13 { end of pdfTeX's command codes }

define eTeX_int = pdftex_last_item_codes + 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 244) +≡

```

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);
primitive("pdftexversion", last_item, pdftex_version_code);
primitive("pdflastobj", last_item, pdf_last_obj_code);
primitive("pdflastxform", last_item, pdf_last_xform_code);
primitive("pdflastximage", last_item, pdf_last_ximage_code);
primitive("pdflastximagepages", last_item, pdf_last_ximage_pages_code);
primitive("pdflastannot", last_item, pdf_last_annot_code);
primitive("pdflastxpos", last_item, pdf_last_x_pos_code);
primitive("pdflastypos", last_item, pdf_last_y_pos_code);
primitive("pdfretval", last_item, pdf_retval_code);
primitive("pdflastximagecolor depth", last_item, pdf_last_ximage_colord depth_code);
primitive("pdfelapsedtime", last_item, elapsed_time_code);
primitive("pdfshellescape", last_item, pdf_shell_escape_code);

```

```
primitive("pdfrandomseed", last_item, random_seed_code);
primitive("pdflastlink", last_item, pdf_last_link_code);
```

443. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡

```
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 1693⟩ 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 1650⟩
    pdftex_version_code: print_esc("pdftexversion");
    pdf_last_obj_code: print_esc("pdflastobj");
    pdf_last_xform_code: print_esc("pdflastxform");
    pdf_last_ximage_code: print_esc("pdflastximage");
    pdf_last_ximage_pages_code: print_esc("pdflastximagepages");
    pdf_last_annot_code: print_esc("pdflastannot");
    pdf_last_x_pos_code: print_esc("pdflastxpos");
    pdf_last_y_pos_code: print_esc("pdflastypos");
    pdf_reval_code: print_esc("pdfreval");
    pdf_last_ximage_colordepth_code: print_esc("pdflastximagecolordepth");
    elapsed_time_code: print_esc("pdfelapsedtime");
    pdf_shell_escape_code: print_esc("pdfshellescape");
    random_seed_code: print_esc("pdfrandomseed");
    pdf_last_link_code: print_esc("pdflastlink");
    othercases print_esc("badness")
  endcases;
```

444. ⟨Fetch the *space_factor* or the *prev_depth* 444⟩ ≡

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

This code is used in section 439.

445. *< Fetch the `dead_cycles` or the `insert_penalties` 445 >* ≡
begin if $m = 0$ **then** $cur_val \leftarrow dead_cycles$
< Cases for 'Fetch the `dead_cycles` or the `insert_penalties`' 1694 >
else $cur_val \leftarrow insert_penalties$; $cur_val_level \leftarrow int_val$;
end

This code is used in section 439.

446. *< Fetch a box dimension 446 >* ≡
begin `scan_register_num`; `fetch_box`(q);
if $q = null$ **then** $cur_val \leftarrow 0$ **else** $cur_val \leftarrow mem[q + m].sc$;
 $cur_val_level \leftarrow dimen_val$;
end

This code is used in section 439.

447. Inside an `\output` routine, a user may wish to look at the page totals that were present at the moment when output was triggered.

```
define max_dimen ≡ '777777777777 { $2^{30} - 1$ }  

<Fetch something on the page_so_far 447 > ≡  

begin if (page_contents = empty)  $\wedge$  ( $\neg output\_active$ ) then  

    if  $m = 0$  then  $cur\_val \leftarrow max\_dimen$  else  $cur\_val \leftarrow 0$   

    else  $cur\_val \leftarrow page\_so\_far[m]$ ;  

     $cur\_val\_level \leftarrow dimen\_val$ ;  

end
```

This code is used in section 439.

448. *< Fetch the `prev_graf` 448 >* ≡
if $mode = 0$ **then** `scanned_result`(0)(`int_val`) {`prev_graf` = 0 within `\write`}
else begin `nest[nest_ptr] ← cur.list`; $p \leftarrow nest_ptr$;
while $abs(nest[p].mode_field) \neq vmode$ **do** `decr(p)`;
`scanned_result(nest[p].pg_field)(int_val)`;
end

This code is used in section 439.

449. *< Fetch the `par_shape` size 449 >* ≡
begin if $m > par_shape_loc$ **then** *< Fetch a penalties array element 1866 >*
else if $par_shape_ptr = null$ **then** $cur_val \leftarrow 0$
else $cur_val \leftarrow info(par_shape_ptr)$;
 $cur_val_level \leftarrow int_val$;
end

This code is used in section 439.

450. 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  $\equiv$  tx  $\leftarrow$  tail;
  if  $\neg$ is_char_node(tx) then
    if (type(tx) = math_node)  $\wedge$  (subtype(tx) = end_M_code) then
      begin r  $\leftarrow$  head;
      repeat q  $\leftarrow$  r; r  $\leftarrow$  link(q);
      until r = tx;
      tx  $\leftarrow$  q;
    end
define find_effective_tail  $\equiv$  find_effective_tail_eTeX

⟨ Fetch an item in the current node, if appropriate 450 ⟩  $\equiv$ 
  if m  $\geq$  input_line_no_code then
    if m  $\geq$  eTeX_glue then ⟨ Process an expression and return 1780 ⟩
    else if m  $\geq$  eTeX_dim then
      begin case m of
        ⟨ Cases for fetching a dimension value 1671 ⟩
      end; { there are no other cases }
      cur_val_level  $\leftarrow$  dimen_val;
    end
  else begin case m of
    input_line_no_code: cur_val  $\leftarrow$  line;
    badness_code: cur_val  $\leftarrow$  last_badness;
    pdftex_version_code: cur_val  $\leftarrow$  pdftex_version;
    pdf_last_obj_code: cur_val  $\leftarrow$  pdf_last_obj;
    pdf_last_xform_code: cur_val  $\leftarrow$  pdf_last_xform;
    pdf_last_ximage_code: cur_val  $\leftarrow$  pdf_last_ximage;
    pdf_last_ximage_pages_code: cur_val  $\leftarrow$  pdf_last_ximage_pages;
    pdf_last_annot_code: cur_val  $\leftarrow$  pdf_last_annot;
    pdf_last_x_pos_code: cur_val  $\leftarrow$  pdf_last_x_pos;
    pdf_last_y_pos_code: cur_val  $\leftarrow$  pdf_last_y_pos;
    pdf_retval_code: cur_val  $\leftarrow$  pdf_retval;
    pdf_last_ximage_colorddepth_code: cur_val  $\leftarrow$  pdf_last_ximage_colorddepth;
    elapsed_time_code: cur_val  $\leftarrow$  get_microinterval;
    random_seed_code: cur_val  $\leftarrow$  random_seed;
    pdf_shell_escape_code: begin if shellenabledp then
      begin if restrictedshell then cur_val  $\leftarrow$  2
      else cur_val  $\leftarrow$  1;
      end
      else cur_val  $\leftarrow$  0;
    end;
    pdf_last_link_code: cur_val  $\leftarrow$  pdf_last_link;
    ⟨ Cases for fetching an integer value 1651 ⟩
  end; { there are no other cases }
  cur_val_level  $\leftarrow$  int_val;
end
else begin if cur_chr = glue_val then cur_val  $\leftarrow$  zero_glue else cur_val  $\leftarrow$  0;
  find_effective_tail;
  if cur_chr = last_node_type_code then
    begin cur_val_level  $\leftarrow$  int_val;

```

```

if ( $tx = head$ )  $\vee$  ( $mode = 0$ ) then  $cur\_val \leftarrow -1$ ;
end
else  $cur\_val\_level \leftarrow cur\_chr$ ;
if  $\neg is\_char\_node(tx) \wedge (mode \neq 0)$  then
  case  $cur\_chr$  of
     $int\_val$ : if  $type(tx) = penalty\_node$  then  $cur\_val \leftarrow penalty(tx)$ ;
     $dimen\_val$ : if  $type(tx) = kern\_node$  then  $cur\_val \leftarrow width(tx)$ ;
     $glue\_val$ : if  $type(tx) = glue\_node$  then
      begin  $cur\_val \leftarrow glue\_ptr(tx)$ ;
      if  $subtype(tx) = mu\_glue$  then  $cur\_val\_level \leftarrow mu\_val$ ;
      end;
     $last\_node\_type\_code$ : if  $type(tx) \leq unset\_node$  then  $cur\_val \leftarrow type(tx) + 1$ 
      else  $cur\_val \leftarrow unset\_node + 2$ ;
    end { there are no other cases }
else if ( $mode = vmode$ )  $\wedge$  ( $tx = head$ ) then
  case  $cur\_chr$  of
     $int\_val$ :  $cur\_val \leftarrow last\_penalty$ ;
     $dimen\_val$ :  $cur\_val \leftarrow last\_kern$ ;
     $glue\_val$ : if  $last\_glue \neq max\_halfword$  then  $cur\_val \leftarrow last\_glue$ ;
     $last\_node\_type\_code$ :  $cur\_val \leftarrow last\_node\_type$ ;
  end; { there are no other cases }
end

```

This code is used in section 439.

451. ⟨ Fetch a font dimension 451 ⟩ ≡

```

begin  $find\_font\_dimen(false)$ ;  $font.info[fmem\_ptr].sc \leftarrow 0$ ;
 $scanned\_result(font.info[cur\_val].sc)(dimen\_val)$ ;
end

```

This code is used in section 439.

452. ⟨ Fetch a font integer 452 ⟩ ≡

```

begin  $scan\_font\_ident$ ;
if  $m = 0$  then  $scanned\_result(hyphen\_char[cur\_val])(int\_val)$ 
else if  $m = 1$  then  $scanned\_result(skew\_char[cur\_val])(int\_val)$ 
else if  $m = no\_lig\_code$  then  $scanned\_result(test\_no\_ligatures(cur\_val))(int\_val)$ 
else begin  $n \leftarrow cur\_val$ ;  $scan\_char\_num$ ;  $k \leftarrow cur\_val$ ;
  case  $m$  of
     $lp\_code\_base$ :  $scanned\_result(get\_lp\_code(n, k))(int\_val)$ ;
     $rp\_code\_base$ :  $scanned\_result(get\_rp\_code(n, k))(int\_val)$ ;
     $ef\_code\_base$ :  $scanned\_result(get\_ef\_code(n, k))(int\_val)$ ;
     $tag\_code$ :  $scanned\_result(get\_tag\_code(n, k))(int\_val)$ ;
     $kn\_bs\_code\_base$ :  $scanned\_result(get\_kn\_bs\_code(n, k))(int\_val)$ ;
     $st\_bs\_code\_base$ :  $scanned\_result(get\_st\_bs\_code(n, k))(int\_val)$ ;
     $sh\_bs\_code\_base$ :  $scanned\_result(get\_sh\_bs\_code(n, k))(int\_val)$ ;
     $kn\_bc\_code\_base$ :  $scanned\_result(get\_kn\_bc\_code(n, k))(int\_val)$ ;
     $kn\_ac\_code\_base$ :  $scanned\_result(get\_kn\_ac\_code(n, k))(int\_val)$ ;
  end;
end;
end

```

This code is used in section 439.

453. *{ Fetch a register 453 }* \equiv

```

begin if ( $m < mem\_bot$ )  $\vee$  ( $m > lo\_mem\_stat\_max$ ) then
  begin  $cur\_val\_level \leftarrow sa\_type(m)$ ;
    if  $cur\_val\_level < glue\_val$  then  $cur\_val \leftarrow sa\_int(m)$ 
    else  $cur\_val \leftarrow sa\_ptr(m)$ ;
    end
  else begin  $scan\_register\_num$ ;  $cur\_val\_level \leftarrow m - mem\_bot$ ;
    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  $cur\_val \leftarrow 0$ 
          else  $cur\_val \leftarrow zero\_glue$ 
        else if  $cur\_val\_level < glue\_val$  then  $cur\_val \leftarrow sa\_int(cur\_ptr)$ 
          else  $cur\_val \leftarrow sa\_ptr(cur\_ptr)$ ;
        end
      else case  $cur\_val\_level$  of
        int_val:  $cur\_val \leftarrow count(cur\_val)$ ;
        dimen_val:  $cur\_val \leftarrow dimen(cur\_val)$ ;
        glue_val:  $cur\_val \leftarrow skip(cur\_val)$ ;
        mu_val:  $cur\_val \leftarrow mu\_skip(cur\_val)$ ;
      end; { there are no other cases }
    end;
  end
end

```

This code is used in section 439.

454. *{ Complain that \the can't do this; give zero result 454 }* \equiv

```

begin  $print\_err("You\can't\use\")$ ;  $print\_cmd\_chr(cur\_cmd, cur\_chr)$ ;  $print("`\after")$ ;
 $print\_esc("the")$ ;  $help1("I'm\forgetting\what\you\said\and\using\zero\instead.")$ ;  $error$ ;
if  $level \neq tok\_val$  then  $scanned\_result(0)(dimen\_val)$ 
else  $scanned\_result(0)(int\_val)$ ;
end

```

This code is used in section 439.

455. When a $glue_val$ changes to a $dimen_val$, we use the width component of the glue; there is no need to decrease the reference count, since it has not yet been increased. When a $dimen_val$ changes to an int_val , we use scaled points so that the value doesn't actually change. And when a mu_val changes to a $glue_val$, the value doesn't change either.

{ Convert cur_val to a lower level 455 } \equiv

```

begin if  $cur\_val\_level = glue\_val$  then  $cur\_val \leftarrow width(cur\_val)$ 
else if  $cur\_val\_level = mu\_val$  then  $mu\_error$ ;
decr( $cur\_val\_level$ );
end

```

This code is used in section 439.

456. If *cur_val* points to a glue specification at this point, the reference count for the glue does not yet include the reference by *cur_val*. If *negative* is true, *cur_val_level* is known to be $\leq \mu_{val}$.

```
<Fix the reference count, if any, and negate cur_val if negative 456> ≡
  if negative then
    if cur_val_level ≥ glue_val then
      begin cur_val ← new_spec(cur_val); <Negate all three glue components of cur_val 457>;
      end
    else negate(cur_val)
  else if (cur_val_level ≥ glue_val) ∧ (cur_val_level ≤ mu_val) then add_glue_ref(cur_val)
```

This code is used in section 439.

457. *<Negate all three glue components of cur_val 457>* ≡

```
begin negate(width(cur_val)); negate(stretch(cur_val)); negate(shrink(cur_val));
end
```

This code is used in sections 456 and 1780.

458. Our next goal is to write the *scan_int* procedure, which scans anything that TeX treats as an integer. But first we might as well look at some simple applications of *scan_int* that have already been made inside of *scan_something_internal*.

459. *<Declare procedures that scan restricted classes of integers 459>* ≡

```
procedure scan_eight_bit_int;
begin scan_int;
if (cur_val < 0) ∨ (cur_val > 255) then
  begin print_err("Bad_register_code");
  help2("A_register_number_must_be_between_0_and_255.");
  ("I_changed_this_one_to_zero."); int_error(cur_val); cur_val ← 0;
  end;
end;
```

See also sections 460, 461, 462, 463, and 1811.

This code is used in section 435.

460. *<Declare procedures that scan restricted classes of integers 459>* +≡

```
procedure scan_char_num;
begin scan_int;
if (cur_val < 0) ∨ (cur_val > 255) then
  begin print_err("Bad_character_code");
  help2("A_character_number_must_be_between_0_and_255.");
  ("I_changed_this_one_to_zero."); int_error(cur_val); cur_val ← 0;
  end;
end;
```

461. While we're at it, we might as well deal with similar routines that will be needed later.

\langle Declare procedures that scan restricted classes of integers 459 $\rangle +\equiv$

```
procedure scan_four_bit_int;
begin scan_int;
if (cur_val < 0) ∨ (cur_val > 15) then
begin print_err("Bad_number");
help2("Since I expected to read a number between 0 and 15, ")
("I changed this one to zero."); int_error(cur_val); cur_val ← 0;
end;
end;
```

462. \langle Declare procedures that scan restricted classes of integers 459 $\rangle +\equiv$

```
procedure scan_fifteen_bit_int;
begin scan_int;
if (cur_val < 0) ∨ (cur_val > '77777) then
begin print_err("Bad_mathchar"); help2("A mathchar number must be between 0 and 32767 .")
("I changed this one to zero."); int_error(cur_val); cur_val ← 0;
end;
end;
```

463. \langle Declare procedures that scan restricted classes of integers 459 $\rangle +\equiv$

```
procedure scan_twenty_seven_bit_int;
begin scan_int;
if (cur_val < 0) ∨ (cur_val > '777777777) then
begin print_err("Bad_delimiter_code");
help2("A numeric delimiter code must be between 0 and 2^{27}-1 .")
("I changed this one to zero."); int_error(cur_val); cur_val ← 0;
end;
end;
```

464. An integer number can be preceded by any number of spaces and '+' or '-' signs. Then comes either a decimal constant (i.e., radix 10), an octal constant (i.e., radix 8, preceded by `), a hexadecimal constant (radix 16, preceded by "), an alphabetic constant (preceded by `), or an internal variable. After scanning is complete, *cur_val* will contain the answer, which must be at most $2^{31} - 1 = 2147483647$ in absolute value. The value of *radix* is set to 10, 8, or 16 in the cases of decimal, octal, or hexadecimal constants, otherwise *radix* is set to zero. An optional space follows a constant.

```
define octal_token = other_token + ` { apostrophe, indicates an octal constant }
define hex_token = other_token + "" " { double quote, indicates a hex constant }
define alpha_token = other_token + ` { reverse apostrophe, precedes alpha constants }
define point_token = other_token + ". " { decimal point }
define continental_point_token = other_token + ", " { decimal point, Eurostyle }

⟨ Global variables 13 ⟩ +≡
radix: small_number; { scan_int sets this to 8, 10, 16, or zero }
```

465. We initialize the following global variables just in case *expand* comes into action before any of the basic scanning routines has assigned them a value.

\langle Set initial values of key variables 21 $\rangle +\equiv$

```
cur_val ← 0; cur_val_level ← int_val; radix ← 0; cur_order ← normal;
```

466. The *scan_int* routine is used also to scan the integer part of a fraction; for example, the '3' in '3.14159' will be found by *scan_int*. The *scan_dimen* routine assumes that *cur_tok* = *point_token* after the integer part of such a fraction has been scanned by *scan_int*, and that the decimal point has been backed up to be scanned again.

```

procedure scan_int; { sets cur_val to an integer }
  label done, restart;
  var negative: boolean; { should the answer be negated? }
  m: integer; { 231 div radix, the threshold of danger }
  d: small_number; { the digit just scanned }
  vacuous: boolean; { have no digits appeared? }
  OK_so_far: boolean; { has an error message been issued? }
begin radix ← 0; OK_so_far ← true;
⟨ Get the next non-blank non-sign token; set negative appropriately 467 ⟩;
restart: if cur_tok = alpha_token then ⟨ Scan an alphabetic character code into cur_val 468 ⟩
  else if cur_tok = cs_token_flag + frozen_primitive then
    ⟨ Reset cur_tok for unexpandable primitives, goto restart 395 ⟩
    else if (cur_cmd ≥ min_internal) ∧ (cur_cmd ≤ max_internal) then
      scan_something_internal(int_val, false)
    else ⟨ Scan a numeric constant 470 ⟩;
  if negative then negate(cur_val);
end;
```

467. ⟨ Get the next non-blank non-sign token; set negative appropriately 467 ⟩ ≡
 negative ← false;
 repeat ⟨ Get the next non-blank non-call token 432 ⟩;
 if cur_tok = other_token + "−" then
 begin negative ← ¬negative; cur_tok ← other_token + "+";
 end;
 until cur_tok ≠ other_token + "+"

This code is used in sections 466, 474, and 487.

468. A space is ignored after an alphabetic character constant, so that such constants behave like numeric ones.

```

⟨ Scan an alphabetic character code into cur_val 468 ⟩ ≡
begin get_token; { suppress macro expansion }
if cur_tok < cs_token_flag then
  begin cur_val ← cur_chr;
  if cur_cmd ≤ right_brace then
    if cur_cmd = right_brace then incr(align_state)
    else decr(align_state);
  end
else if cur_tok < cs_token_flag + single_base then cur_val ← cur_tok − cs_token_flag − active_base
  else cur_val ← cur_tok − cs_token_flag − single_base;
if cur_val > 255 then
  begin print_err("Improper_alphanticn constant");
  help2("Anone-characterncontrolnsequencenbelongsnafternan `nmark.n")
  ("SonIn'mnessentiallyninsertingn\0nhere.n"); cur_val ← "0"; back_error;
  end
else ⟨ Scan an optional space 469 ⟩;
end
```

This code is used in section 466.

469. \langle Scan an optional space 469 $\rangle \equiv$

```
begin get_x_token;
if cur_cmd ≠ spacer then back_input;
end
```

This code is used in sections 468, 474, 481, 705, 1378, 1544, 1556, 1556, 1558, and 1565.

470. \langle Scan a numeric constant 470 $\rangle \equiv$

```
begin radix ← 10; m ← 214748364;
if cur_tok = octal_token then
begin radix ← 8; m ← '2000000000; get_x_token;
end
else if cur_tok = hex_token then
begin radix ← 16; m ← '1000000000; get_x_token;
end;
vacuous ← true; cur_val ← 0;
⟨ Accumulate the constant until cur_tok is not a suitable digit 471 ⟩;
if vacuous then ⟨ Express astonishment that no number was here 472 ⟩
else if cur_cmd ≠ spacer then back_input;
end
```

This code is used in section 466.

471. define infinity ≡ '17777777777 { the largest positive value that TeX knows }

```
define zero_token = other_token + "0" { zero, the smallest digit }
define A_token = letter_token + "A" { the smallest special hex digit }
define other_A_token = other_token + "A" { special hex digit of type other_char }
```

⟨ Accumulate the constant until cur_tok is not a suitable digit 471 ⟩ ≡

```
loop begin if (cur_tok < zero_token + radix) ∧ (cur_tok ≥ zero_token) ∧ (cur_tok ≤ zero_token + 9)
then d ← cur_tok - zero_token
else if radix = 16 then
if (cur_tok ≤ A_token + 5) ∧ (cur_tok ≥ A_token) then d ← cur_tok - A_token + 10
else if (cur_tok ≤ other_A_token + 5) ∧ (cur_tok ≥ other_A_token) then
d ← cur_tok - other_A_token + 10
else goto done
else goto done;
vacuous ← false;
if (cur_val ≥ m) ∧ ((cur_val > m) ∨ (d > 7) ∨ (radix ≠ 10)) then
begin if OK_so_far then
begin print_err("Number_too_big");
help2("I_can_only_go_up_to_2147483647='17777777777=""7FFFFFFF,");
("so_I_m_using_that_number_instead_of_yours."); error; cur_val ← infinity;
OK_so_far ← false;
end;
end
else cur_val ← cur_val * radix + d;
get_x_token;
end;
```

done:

This code is used in section 470.

472. { Express astonishment that no number was here 472 } ≡

```
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;
end
```

This code is used in section 470.

473. The *scan_dimen* routine is similar to *scan_int*, but it sets *cur_val* to a *scaled* value, i.e., an integral number of sp. One of its main tasks is therefore to interpret the abbreviations for various kinds of units and to convert measurements to scaled points.

There are three parameters: *mu* is *true* if the finite units must be 'mu', while *mu* is *false* if 'mu' units are disallowed; *inf* is *true* if the infinite units 'fil', 'fill', 'filll' are permitted; and *shortcut* is *true* if *cur_val* already contains an integer and only the units need to be considered.

The order of infinity that was found in the case of infinite glue is returned in the global variable *cur_order*.

{ Global variables 13 } +≡
cur_order: *glue_ord*; { order of infinity found by *scan_dimen* }

474. Constructions like ‘`-`77 pt’ are legal dimensions, so *scan_dimen* may begin with *scan_int*. This explains why it is convenient to use *scan_int* also for the integer part of a decimal fraction.

Several branches of *scan_dimen* work with *cur_val* as an integer and with an auxiliary fraction *f*, so that the actual quantity of interest is $cur_val + f/2^{16}$. At the end of the routine, this “unpacked” representation is put into the single word *cur_val*, which suddenly switches significance from *integer* to *scaled*.

```

define attach_fraction = 88 { go here to pack cur_val and f into cur_val }
define attach_sign = 89 { go here when cur_val is correct except perhaps for sign }
define scan_normal_dimen ≡ scan_dimen(false, false, false)

procedure scan_dimen(mu, inf, shortcut : boolean); { sets cur_val to a dimension }
label done, done1, done2, found, not_found, attach_fraction, attach_sign;
var negative: boolean; { should the answer be negated? }
  f: integer; { numerator of a fraction whose denominator is 216 }
  { Local variables for dimension calculations 476 }

begin f ← 0; arith_error ← false; cur_order ← normal; negative ← false;
if ¬shortcut then
  begin { Get the next non-blank non-sign token; set negative appropriately 467 };
  if (cur_cmd ≥ min_internal) ∧ (cur_cmd ≤ max_internal) then
    { Fetch an internal dimension and goto attach_sign, or fetch an internal integer 475 }
  else begin back_input;
    if cur_tok = continental_point_token then cur_tok ← point_token;
    if cur_tok ≠ point_token then scan_int
    else begin radix ← 10; cur_val ← 0;
      end;
    if cur_tok = continental_point_token then cur_tok ← point_token;
    if (radix = 10) ∧ (cur_tok = point_token) then { Scan decimal fraction 478 };
    end;
  end;
  if cur_val < 0 then { in this case f = 0 }
    begin negative ← ¬negative; negate(cur_val);
  end;
  { Scan units and set cur_val to x · (cur_val + f/216), where there are x sp per unit; goto attach_sign if
    the units are internal 479 };
  { Scan an optional space 469 };
attach_sign: if arith_error ∨ (abs(cur_val) ≥ 10000000000) then
  { Report that this dimension is out of range 486 };
  if negative then negate(cur_val);
end;
```

475. { Fetch an internal dimension and goto attach_sign, or fetch an internal integer 475 } ≡

```

if mu then
  begin scan_something_internal(mu_val, false); { Coerce glue to a dimension 477 };
  if cur_val_level = mu_val then goto attach_sign;
  if cur_val_level ≠ int_val then mu_error;
  end
else begin scan_something_internal(dimen_val, false);
  if cur_val_level = dimen_val then goto attach_sign;
end
```

This code is used in section 474.

476. \langle Local variables for dimension calculations 476 $\rangle \equiv$
num, denom: 1 .. 65536; {conversion ratio for the scanned units}
k, kk: small_number; {number of digits in a decimal fraction}
p, q: pointer; {top of decimal digit stack}
v: scaled; {an internal dimension}
save_cur_val: integer; {temporary storage of cur_val}

This code is used in section 474.

477. The following code is executed when *scan_something_internal* was called asking for *mu_val*, when we really wanted a “mudimen” instead of “muglue.”

\langle Coerce glue to a dimension 477 $\rangle \equiv$
if *cur_val.level* \geq *glue_val* **then**
 begin *v* \leftarrow *width(cur_val)*; *delete_glue_ref(cur_val)*; *cur_val* \leftarrow *v*;
 end

This code is used in sections 475 and 481.

478. When the following code is executed, we have *cur_tok = point_token*, but this token has been backed up using *back_input*; we must first discard it.

It turns out that a decimal point all by itself is equivalent to ‘0.0’. Let’s hope people don’t use that fact.

\langle Scan decimal fraction 478 $\rangle \equiv$
begin *k* \leftarrow 0; *p* \leftarrow null; *get_token*; {*point_token* is being re-scanned}
loop begin *get_x_token*;
 if (*cur_tok* $>$ *zero_token* + 9) \vee (*cur_tok* $<$ *zero_token*) **then goto** *done1*;
 if *k* $<$ 17 **then** {digits for *k* \geq 17 cannot affect the result}
 begin *q* \leftarrow *get_avail*; *link(q)* \leftarrow *p*; *info(q)* \leftarrow *cur_tok - zero_token*; *p* \leftarrow *q*; *incr(k)*;
 end;
 end;
done1: for *kk* \leftarrow *k* **downto** 1 **do**
 begin *dig[kk - 1]* \leftarrow *info(p)*; *q* \leftarrow *p*; *p* \leftarrow *link(p)*; *free_avail(q)*;
 end;
 f \leftarrow *round_decimals(k)*;
 if *cur_cmd* \neq *spacer* **then** *back_input*;
 end

This code is used in section 474.

479. Now comes the harder part: At this point in the program, *cur_val* is a nonnegative integer and $f/2^{16}$ is a nonnegative fraction less than 1; we want to multiply the sum of these two quantities by the appropriate factor, based on the specified units, in order to produce a *scaled* result, and we want to do the calculation with fixed point arithmetic that does not overflow.

```
<Scan units and set cur_val to  $x \cdot (cur\_val + f/2^{16})$ , where there are  $x$  sp per unit; goto attach_sign if the
units are internal 479> ≡
if inf then <Scan for fil units; goto attach_fraction if found 480>;
<Scan for units that are internal dimensions; goto attach_sign with cur_val set if found 481>;
if mu then <Scan for mu units and goto attach_fraction 482>;
if scan_keyword("true") then <Adjust for the magnification ratio 483>;
if scan_keyword("pt") then goto attach_fraction; { the easy case }
<Scan for all other units and adjust cur_val and f accordingly; goto done in the case of scaled
points 484>;
attach_fraction: if cur_val ≥ '40000 then arith_error ← true
else cur_val ← cur_val * unity + f;
done:
```

This code is used in section 474.

480. A specification like ‘filllll’ or ‘fill L L L’ will lead to two error messages (one for each additional keyword “L”).

```
<Scan for fil units; goto attach_fraction if found 480> ≡
if scan_keyword("fil") then
begin cur_order ← fil;
while scan_keyword("L") do
begin if cur_order = fill then
begin print_err("Illegal_unit_of_measure"); print("replaced_by_fill");
help1("I_ddon't_go_any_higher_than_fill."); error;
end
else incr(cur_order);
end;
goto attach_fraction;
end
```

This code is used in section 479.

481. { Scan for units that are internal dimensions; goto attach_sign with cur_val set if found 481 } ≡
 $\text{save_cur_val} \leftarrow \text{cur_val};$ { Get the next non-blank non-call token 432 };
if ($\text{cur_cmd} < \text{min_internal}$) \vee ($\text{cur_cmd} > \text{max_internal}$) **then** back_input
else begin if mu then
 begin scan_something_internal(mu_val, false); { Coerce glue to a dimension 477 };
 if cur_val_level \neq mu_val **then** mu_error;
 end
 else scan_something_internal(dimen_val, false);
 $v \leftarrow \text{cur_val};$ **goto** found;
 end;
if mu **then** goto not_found;
if scan_keyword("em") **then** $v \leftarrow (\langle \text{The em width for cur_font 584} \rangle)$
else if scan_keyword("ex") **then** $v \leftarrow (\langle \text{The x-height for cur_font 585} \rangle)$
else if scan_keyword("px") **then** $v \leftarrow \text{pdf_px_dimen}$
else goto not_found;
{ Scan an optional space 469 };
found: $\text{cur_val} \leftarrow \text{nx_plus_y}(\text{save_cur_val}, v, \text{xn_over_d}(v, f, '200000));$ **goto** attach_sign;
not_found:

This code is used in section 479.

482. { Scan for mu units and goto attach_fraction 482 } ≡
if scan_keyword("mu") **then** goto attach_fraction
else begin print_err("Illegal unit of measure(); print("mu inserted");
 help4("The unit of measurement in math glue must be mu.");
 ("To recover gracefully from this error, it's best to")
 ("delete the erroneous units; e.g., type `2' to delete")
 ("two letters. (See Chapter 27 of The TeXbook.)"); error; **goto** attach_fraction;
end

This code is used in section 479.

483. { Adjust for the magnification ratio 483 } ≡
begin prepare_mag;
if mag \neq 1000 **then**
 begin cur_val $\leftarrow \text{xn_over_d}(\text{cur_val}, 1000, \text{mag});$ $f \leftarrow (1000 * f + '200000 * remainder) \text{ div } \text{mag};$
 $\text{cur_val} \leftarrow \text{cur_val} + (f \text{ div } '200000);$ $f \leftarrow f \text{ mod } '200000;$
 end;
end

This code is used in section 479.

484. The necessary conversion factors can all be specified exactly as fractions whose numerator and denominator sum to 32768 or less. According to the definitions here, $2660\text{dd} \approx 1000.33297\text{ mm}$; this agrees well with the value 1000.333 mm cited by Bosshard in *Technische Grundlagen zur Satzherstellung* (Bern, 1980). The Didot point has been newly standardized in 1978; it's now exactly $1\text{nd} = 0.375\text{ mm}$. Conversion uses the equation $0.375 = 21681/20320/72.27 \cdot 25.4$. The new Cicero follows the new Didot point; $1\text{nc} = 12\text{nd}$. These would lead to the ratios $21681/20320$ and $65043/5080$, respectively. The closest approximations supported by the algorithm would be $11183/10481$ and $1370/107$. In order to maintain the relation $1\text{nc} = 12\text{nd}$, we pick the ratio $685/642$ for nd , however.

```

define set_conversion_end (#) ≡ denom ← #;
end
define set_conversion (#) ≡ begin num ← #; set_conversion_end
{ Scan for all other units and adjust cur_val and f accordingly; goto done in the case of scaled points 484 } ≡
  if scan_keyword("in") then set_conversion(7227)(100)
  else if scan_keyword("pc") then set_conversion(12)(1)
  else if scan_keyword("cm") then set_conversion(7227)(254)
  else if scan_keyword("mm") then set_conversion(7227)(2540)
  else if scan_keyword("bp") then set_conversion(7227)(7200)
  else if scan_keyword("dd") then set_conversion(1238)(1157)
  else if scan_keyword("cc") then set_conversion(14856)(1157)
  else if scan_keyword("nd") then set_conversion(685)(642)
  else if scan_keyword("nc") then set_conversion(1370)(107)
  else if scan_keyword("sp") then goto done
  else { Complain about unknown unit and goto done2 485 };
  cur_val ← xn_over_d(cur_val, num, denom); f ← (num * f + `200000 * remainder) div denom;
  cur_val ← cur_val + (f div `200000); f ← f mod `200000;
done2:

```

This code is used in section 479.

485. { Complain about unknown unit and **goto** done2 485 } ≡

```

begin print_err("Illegal_unit_of_measure(); print("pt_inserted");
help6("Dimensions can be in units of em, ex, pt, pc, ")
("cm, mm, dd, cc, nd, nc, bp, or sp; but yours is a new one!")
("I'll assume that you meant to say pt, for printer's points.")
("To recover gracefully from this error, hit 's' best to")
("delete the erroneous units; e.g., type `2' to delete")
("two letters. (See Chapter 27 of The TeXbook.)"); error; goto done2;
end

```

This code is used in section 484.

486. { Report that this dimension is out of range 486 } ≡

```

begin print_err("Dimension_too_large");
help2("I can't work with sizes bigger than about 19 feet.")
("Continue and I'll use the largest value I can.");
error; cur_val ← max_dimen; arith_error ← false;
end

```

This code is used in section 474.

487. 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 467 };
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 488 };
exit: end;
{ Declare procedures needed for expressions 1782 }
```

488. { Create a new glue specification whose width is *cur_val*; scan for its stretch and shrink components 488 } ≡

```
q ← new_spec(zero_glue); width(q) ← cur_val;
if scan_keyword("plus") then
begin scan_dimen(mu, true, false); stretch(q) ← cur_val; stretch_order(q) ← cur_order;
end;
if scan_keyword("minus") then
begin scan_dimen(mu, true, false); shrink(q) ← cur_val; shrink_order(q) ← cur_order;
end;
cur_val ← q
```

This code is used in section 487.

489. Here's a similar procedure that returns a pointer to a rule node. This routine is called just after TeX has seen `\hrule` or `\vrule`; therefore `cur_cmd` will be either `hrule` or `vrule`. The idea is to store the default rule dimensions in the node, then to override them if 'height' or 'width' or 'depth' specifications are found (in any order).

```
define default_rule = 26214 { 0.4pt }
function scan_rule_spec: pointer;
label reswitch;
var q: pointer; { the rule node being created }
begin q ← new_rule; { width, depth, and height all equal null_flag now }
if cur_cmd = vrule then width(q) ← default_rule
else begin height(q) ← default_rule; depth(q) ← 0;
end;
reswitch: if scan_keyword("width") then
begin scan_normal_dimen; width(q) ← cur_val; goto reswitch;
end;
if scan_keyword("height") then
begin scan_normal_dimen; height(q) ← cur_val; goto reswitch;
end;
if scan_keyword("depth") then
begin scan_normal_dimen; depth(q) ← cur_val; goto reswitch;
end;
scan_rule_spec ← q;
end;
```

490. 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 1683 >
function str_toks(b: pool_pointer): pointer; { converts str_pool[b .. pool_ptr - 1] 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;
```

491. 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 1688};
  get_x_token; scan_something_internal(tok_val, false);
  if cur_val.level ≥ ident_val then {Copy the token list 492}
  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;
```

492. $\langle \text{Copy the token list } 492 \rangle \equiv$

```
begin  $p \leftarrow \text{temp\_head}$ ;  $\text{link}(p) \leftarrow \text{null}$ ;
if  $\text{cur\_val.level} = \text{ident\_val}$  then  $\text{store\_new\_token}(\text{cs\_token\_flag} + \text{cur\_val})$ 
else if  $\text{cur\_val} \neq \text{null}$  then
  begin  $r \leftarrow \text{link}(\text{cur\_val})$ ; { do not copy the reference count }
    while  $r \neq \text{null}$  do
      begin  $\text{fast\_store\_new\_token}(\text{info}(r))$ ;  $r \leftarrow \text{link}(r)$ ;
        end;
    end;
  end;
 $\text{the\_toks} \leftarrow p$ ;
end
```

This code is used in section 491.

493. Here's part of the *expand* subroutine that we are now ready to complete:

```
procedure  $\text{ins\_the\_toks}$ ;
begin  $\text{link}(\text{garbage}) \leftarrow \text{the\_toks}$ ;  $\text{ins\_list}(\text{link}(\text{temp\_head}))$ ;
end;
```

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

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

pdftEX adds `\pdftexrevision`, `\pdftexbanner`, `\pdffontname`, `\pdffontobjnum`, `\pdffontsize`, and `\pdfpageref` 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  $\varepsilon$ -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  $\varepsilon$ -TeX's command codes }
define expanded_code = etex_convert_codes { command code for \expanded }
define pdftex_first_expand_code = expanded_code + 1 { base for pdftEX's command codes }
define pdftex_revision_code = pdftex_first_expand_code + 0 { command code for \pdftexrevision }
define pdftex_banner_code = pdftex_first_expand_code + 1 { command code for \pdftexbanner }
define pdf_font_name_code = pdftex_first_expand_code + 2 { command code for \pdffontname }
define pdf_font_objnum_code = pdftex_first_expand_code + 3 { command code for \pdffontobjnum }
define pdf_font_size_code = pdftex_first_expand_code + 4 { command code for \pdffontsize }
define pdf_page_ref_code = pdftex_first_expand_code + 5 { command code for \pdfpageref }
define pdf_xform_name_code = pdftex_first_expand_code + 6 { command code for \pdfxformname }
define pdf_escape_string_code = pdftex_first_expand_code + 7 { command code for \pdfescapestring }
define pdf_escape_name_code = pdftex_first_expand_code + 8 { command code for \pdfescapename }
define left_margin_kern_code = pdftex_first_expand_code + 9 { command code for \leftmarginkern }
define right_margin_kern_code = pdftex_first_expand_code + 10 { command code for \rightmarginkern }
define pdf_strcmp_code = pdftex_first_expand_code + 11 { command code for \pdfstrcmp }
define pdf_colorstack_init_code = pdftex_first_expand_code + 12
    { command code for \pdfcolorstackinit }
define pdf_escape_hex_code = pdftex_first_expand_code + 13 { command code for \pdfescapehex }
define pdf_unescape_hex_code = pdftex_first_expand_code + 14 { command code for \pdfunescapehex }
define pdf_creation_date_code = pdftex_first_expand_code + 15 { command code for \pdfcreationdate }
define pdf_file_mod_date_code = pdftex_first_expand_code + 16 { command code for \pdffilemoddate }
define pdf_file_size_code = pdftex_first_expand_code + 17 { command code for \pdffilesize }
define pdf_md5sum_code = pdftex_first_expand_code + 18 { command code for \pdfmd5sum }
define pdf_file_dump_code = pdftex_first_expand_code + 19 { command code for \pdffiledump }
define pdf_match_code = pdftex_first_expand_code + 20 { command code for \pdfmatch }
define pdf_last_match_code = pdftex_first_expand_code + 21 { command code for \pdflastmatch }
define uniform_deviate_code = pdftex_first_expand_code + 22 { end of pdftEX's command codes }
define normal_deviate_code = pdftex_first_expand_code + 23 { end of pdftEX's command codes }
define pdf_insert_ht_code = pdftex_first_expand_code + 24 { command code for \pdfinsertht }
define pdf_ximage_bbox_code = pdftex_first_expand_code + 25 { command code for \pdfximagebbox }
define pdftex_convert_codes = pdftex_first_expand_code + 26 { end of pdftEX's command codes }
define job_name_code = pdftex_convert_codes { command code for \jobname }
```

(Put each of TeX's primitives into the hash table 244) \equiv

```
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("expanded", convert, expanded_code);
```

```
primitive("pdftexrevision", convert, pdf_tex_revision_code);
primitive("pdftexbanner", convert, pdf_tex_banner_code);
primitive("pdffontname", convert, pdf_font_name_code);
primitive("pdffontobjnum", convert, pdf_font_objnum_code);
primitive("pdffontsize", convert, pdf_font_size_code);
primitive("pdfpageref", convert, pdf_page_ref_code);
primitive("leftmarginkern", convert, left_margin_kern_code);
primitive("rightmarginkern", convert, right_margin_kern_code);
primitive("pdfxformname", convert, pdf_xform_name_code);
primitive("pdfescapestring", convert, pdf_escape_string_code);
primitive("pdfescapename", convert, pdf_escape_name_code);
primitive("pdfescapehex", convert, pdf_escape_hex_code);
primitive("pdfunescapehex", convert, pdf_unescape_hex_code);
primitive("pdfcreationdate", convert, pdf_creation_date_code);
primitive("pdffilemoddate", convert, pdf_file_mod_date_code);
primitive("pdffilesize", convert, pdf_file_size_code);
primitive("pdfmdfivesum", convert, pdf_mdfive_sum_code);
primitive("pdffiledump", convert, pdf_file_dump_code);
primitive("pdfmatch", convert, pdf_match_code);
primitive("pdflastmatch", convert, pdf_last_match_code);
primitive("pdfstrcmp", convert, pdf_strcmp_code);
primitive("pdfcolorstackinit", convert, pdf_colorstack_init_code);
primitive("pdfuniformdeviate", convert, uniform_deviate_code);
primitive("pdfnormaldeviate", convert, normal_deviate_code);
primitive("jobname", convert, job_name_code);
primitive("pdfinsertht", convert, pdf_insert_ht_code);
primitive("pdfximagebbox", convert, pdf_ximage_bbox_code);
```

495. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡
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");
expanded_code: print_esc("expanded");
pdftex_revision_code: print_esc("pdftexrevision");
pdftex_banner_code: print_esc("pdftexbanner");
pdf_font_name_code: print_esc("pdffontname");
pdf_font_objnum_code: print_esc("pdffontobjnum");
pdf_font_size_code: print_esc("pdffontsize");
pdf_page_ref_code: print_esc("pdfpageref");
left_margin_kern_code: print_esc("leftmarginkern");
right_margin_kern_code: print_esc("rightmarginkern");
pdf_xform_name_code: print_esc("pdfxformname");
pdf_escape_string_code: print_esc("pdfescapestring");
pdf_escape_name_code: print_esc("pdfescapename");
pdf_escape_hex_code: print_esc("pdfescapehex");
pdf_unescape_hex_code: print_esc("pdfunescapehex");
pdf_creation_date_code: print_esc("pdfcreationdate");
pdf_file_mod_date_code: print_esc("pdffilemoddate");
pdf_file_size_code: print_esc("pdffilesize");
pdf_mdfive_sum_code: print_esc("pdfmdfivesum");
pdf_file_dump_code: print_esc("pdffiledump");
pdf_match_code: print_esc("pdfmatch");
pdf_last_match_code: print_esc("pdflastmatch");
pdf_strcmp_code: print_esc("pdfstrcmp");
pdf_colorstack_init_code: print_esc("pdfcolorstackinit");
uniform_deviate_code: print_esc("pdfuniformdeviate");
normal_deviate_code: print_esc("pdfnormaldeviate");
pdf_insert_ht_code: print_esc("pdfinsertht");
pdf_ximage_bbox_code: print_esc("pdfximagebbox");
othercases print_esc("jobname")
endcases;

```

496. The procedure *conv_toks* uses *str_toks* to insert the token list for *convert* functions into the scanner; ‘\outer’ control sequences are allowed to follow ‘\string’ and ‘\meaning’.

The extra temp string *u* is needed because *pdf_scan_ext_toks* incorporates any pending string in its output. In order to save such a pending string, we have to create a temporary string that is destroyed immediately after.

```

define save_cur_string ≡
  if str_start[str_ptr] < pool_ptr then u ← make_string
define restore_cur_string ≡
  if u ≠ 0 then
    begin decr(str_ptr); u ← 0;
  end

procedure conv_toks;
  label exit;
  var old_setting: 0 .. max_selector; { holds selector setting }
  p, q: pointer; c: number_code .. job_name_code; { desired type of conversion }
  save_scanner_status: small_number; { scanner_status upon entry }
  save_def_ref: pointer; { def_ref upon entry, important if inside '\message' }
  save_warning_index: pointer; bool: boolean; { temp boolean }
  i: integer; { first temp integer }
  j: integer; { second temp integer }
  b: pool_pointer; { base of temporary string }
  s: str_number; { first temp string }
  t: str_number; { second temp string }
  u: str_number; { saved current string string }

begin c ← cur_chr; u ← 0; { will become non-nil if a string is already being built }
  { Scan the argument for command c 497 };
  old_setting ← selector; selector ← new_string; b ← pool_ptr; { Print the result of command c 498 };
  selector ← old_setting; link(garbage) ← str_toks(b); ins_list(link(temp_head));
  exit: end;

```

497. { Scan the argument for command *c* 497 } ≡

```

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;
  expanded_code: begin save_scanner_status ← scanner_status; save_warning_index ← warning_index;
    save_def_ref ← def_ref; save_cur_string; scan_pdf_ext_toks; warning_index ← save_warning_index;
    scanner_status ← save_scanner_status; ins_list(link(def_ref)); free_avail(def_ref);
    def_ref ← save_def_ref; restore_cur_string; return;
  end;
  pdftex_revision_code: do_nothing;
  pdftex_banner_code: do_nothing;
  pdf_font_name_code, pdf_font_objnum_code, pdf_font_size_code: begin scan_font_ident;
    if cur_val = null_font then pdf_error("font", "invalid_font_identifier");
    if c ≠ pdf_font_size_code then
      begin pdf_check_vf_cur_val;
      if ¬font_used[cur_val] then pdf_init_font_cur_val;
      end;
    end;
    pdf_page_ref_code: begin scan_int;
      if cur_val ≤ 0 then pdf_error("pageref", "invalid_page_number");
    end;
    left_margin_kern_code, right_margin_kern_code: begin scan_register_num; fetch_box(p);
      if (p = null) ∨ (type(p) ≠ hlist_node) then pdf_error("marginkern", "a non-empty hbox expected")
    end;
    pdf_xform_name_code: begin scan_int; pdf_check_obj(obj_type_xform, cur_val);
    end;
    pdf_escape_string_code: begin save_scanner_status ← scanner_status;
      save_warning_index ← warning_index; save_def_ref ← def_ref; save_cur_string; scan_pdf_ext_toks;
      s ← tokens_to_string(def_ref); delete_token_ref(def_ref); def_ref ← save_def_ref;
      warning_index ← save_warning_index; scanner_status ← save_scanner_status; b ← pool_ptr;
      escapestring(str_start[s]); link(garbage) ← str_toks(b); flush_str(s); ins_list(link(temp_head));
      restore_cur_string; return;
    end;
    pdf_escape_name_code: begin save_scanner_status ← scanner_status;
      save_warning_index ← warning_index; save_def_ref ← def_ref; save_cur_string; scan_pdf_ext_toks;
      s ← tokens_to_string(def_ref); delete_token_ref(def_ref); def_ref ← save_def_ref;
      warning_index ← save_warning_index; scanner_status ← save_scanner_status; b ← pool_ptr;
      escapename(str_start[s]); link(garbage) ← str_toks(b); flush_str(s); ins_list(link(temp_head));
      restore_cur_string; return;
    end;
    pdf_escape_hex_code: begin save_scanner_status ← scanner_status; save_warning_index ← warning_index;
      save_def_ref ← def_ref; save_cur_string; scan_pdf_ext_toks; s ← tokens_to_string(def_ref);
      delete_token_ref(def_ref); def_ref ← save_def_ref; warning_index ← save_warning_index;
      scanner_status ← save_scanner_status; b ← pool_ptr; escapehex(str_start[s]);
      link(garbage) ← str_toks(b); flush_str(s); ins_list(link(temp_head)); restore_cur_string; return;
    end;
    pdf_unescape_hex_code: begin save_scanner_status ← scanner_status;
      save_warning_index ← warning_index; save_def_ref ← def_ref; save_cur_string; scan_pdf_ext_toks;

```

```

 $s \leftarrow tokens\_to\_string(def\_ref); delete\_token\_ref(def\_ref); def\_ref \leftarrow save\_def\_ref;$ 
 $warning\_index \leftarrow save\_warning\_index; scanner\_status \leftarrow save\_scanner\_status; b \leftarrow pool\_ptr;$ 
 $unescapehex(str\_start[s]); link(garbage) \leftarrow str\_toks(b); flush\_str(s); ins\_list(link(temp\_head));$ 
 $restore\_cur\_string; return;$ 
end;
pdf_creation_date_code: begin  $b \leftarrow pool\_ptr; getcreationdate; link(garbage) \leftarrow str\_toks(b);$ 
 $ins\_list(link(temp\_head)); return;$ 
end;
pdf_file_mod_date_code: begin save_scanner_status  $\leftarrow$  scanner_status;
 $save\_warning\_index \leftarrow warning\_index; save\_def\_ref \leftarrow def\_ref; save\_cur\_string; scan\_pdf\_ext\_toks;$ 
 $s \leftarrow tokens\_to\_string(def\_ref); delete\_token\_ref(def\_ref); def\_ref \leftarrow save\_def\_ref;$ 
 $warning\_index \leftarrow save\_warning\_index; scanner\_status \leftarrow save\_scanner\_status; b \leftarrow pool\_ptr;$ 
 $getfilemoddate(s); link(garbage) \leftarrow str\_toks(b); flush\_str(s); ins\_list(link(temp\_head));$ 
 $restore\_cur\_string; return;$ 
end;
pdf_file_size_code: begin save_scanner_status  $\leftarrow$  scanner_status; save_warning_index  $\leftarrow$  warning_index;
 $save\_def\_ref \leftarrow def\_ref; save\_cur\_string; scan\_pdf\_ext\_toks; s \leftarrow tokens\_to\_string(def\_ref);$ 
 $delete\_token\_ref(def\_ref); def\_ref \leftarrow save\_def\_ref; warning\_index \leftarrow save\_warning\_index;$ 
 $scanner\_status \leftarrow save\_scanner\_status; b \leftarrow pool\_ptr; getfilesize(s); link(garbage) \leftarrow str\_toks(b);$ 
 $flush\_str(s); ins\_list(link(temp\_head)); restore\_cur\_string; return;$ 
end;
pdf_mdfive_sum_code: begin save_scanner_status  $\leftarrow$  scanner_status;
 $save\_warning\_index \leftarrow warning\_index; save\_def\_ref \leftarrow def\_ref; save\_cur\_string;$ 
 $bool \leftarrow scan\_keyword("file"); scan\_pdf\_ext\_toks; s \leftarrow tokens\_to\_string(def\_ref);$ 
 $delete\_token\_ref(def\_ref); def\_ref \leftarrow save\_def\_ref; warning\_index \leftarrow save\_warning\_index;$ 
 $scanner\_status \leftarrow save\_scanner\_status; b \leftarrow pool\_ptr; getmd5sum(s, bool); link(garbage) \leftarrow str\_toks(b);$ 
 $flush\_str(s); ins\_list(link(temp\_head)); restore\_cur\_string; return;$ 
end;
pdf_file_dump_code: begin save_scanner_status  $\leftarrow$  scanner_status; save_warning_index  $\leftarrow$  warning_index;
 $save\_def\_ref \leftarrow def\_ref; save\_cur\_string; \{ scan offset \}$ 
 $cur\_val \leftarrow 0;$ 
if (scan_keyword("offset")) then
 $\begin{aligned} &\text{begin } scan\_int; \\ &\text{if } (cur\_val < 0) \text{ then} \\ &\quad \begin{aligned} &\text{begin } print\_err("Bad\_file\_offset"); \\ &\quad help2("A\_file\_offset\_must\_be\_between\_0\_and\_2^{31}-1,") \\ &\quad ("I\_changed\_this\_one\_to\_zero."); int\_error(cur\_val); cur\_val \leftarrow 0; \\ &\quad \text{end}; \end{aligned} \\ &\text{end}; \end{aligned}$ 
 $i \leftarrow cur\_val; \{ \text{scan length} \}$ 
 $cur\_val \leftarrow 0;$ 
if (scan_keyword("length")) then
 $\begin{aligned} &\text{begin } scan\_int; \\ &\text{if } (cur\_val < 0) \text{ then} \\ &\quad \begin{aligned} &\text{begin } print\_err("Bad\_dump\_length"); \\ &\quad help2("A\_dump\_length\_must\_be\_between\_0\_and\_2^{31}-1,") \\ &\quad ("I\_changed\_this\_one\_to\_zero."); int\_error(cur\_val); cur\_val \leftarrow 0; \\ &\quad \text{end}; \end{aligned} \\ &\text{end}; \end{aligned}$ 
 $j \leftarrow cur\_val; \{ \text{scan file name} \}$ 
 $scan\_pdf\_ext\_toks; s \leftarrow tokens\_to\_string(def\_ref); delete\_token\_ref(def\_ref); def\_ref \leftarrow save\_def\_ref;$ 
 $warning\_index \leftarrow save\_warning\_index; scanner\_status \leftarrow save\_scanner\_status; b \leftarrow pool\_ptr;$ 

```

```

getfiledump(s, i, j); link(garbage)  $\leftarrow$  str_toks(b); flush_str(s); ins_list(link(temp_head));
restore_cur_string; return;
end;

pdf_match_code: begin save_scanner_status  $\leftarrow$  scanner_status; save_warning_index  $\leftarrow$  warning_index;
save_def_ref  $\leftarrow$  def_ref; save_cur_string; { scan for icase }
bool  $\leftarrow$  scan_keyword("icase"); { scan for subcount }
i  $\leftarrow$  -1; { default for subcount }
if scan_keyword("subcount") then
  begin scan_int; i  $\leftarrow$  cur_val;
  end;
scan_pdf_ext_toks; s  $\leftarrow$  tokens_to_string(def_ref); delete_token_ref(def_ref); scan_pdf_ext_toks;
t  $\leftarrow$  tokens_to_string(def_ref); delete_token_ref(def_ref); def_ref  $\leftarrow$  save_def_ref;
warning_index  $\leftarrow$  save_warning_index; scanner_status  $\leftarrow$  save_scanner_status; b  $\leftarrow$  pool_ptr;
matchstrings(s, t, i, bool); link(garbage)  $\leftarrow$  str_toks(b); flush_str(t); flush_str(s);
ins_list(link(temp_head)); restore_cur_string; return;
end;

pdf_last_match_code: begin scan_int;
if cur_val < 0 then
  begin print_err("Bad_match_number");
  help2("Since I expected zero or a positive number,")
  ("I changed this one to zero."); int_error(cur_val); cur_val  $\leftarrow$  0;
  end;
b  $\leftarrow$  pool_ptr; getmatch(cur_val); link(garbage)  $\leftarrow$  str_toks(b); ins_list(link(temp_head)); return;
end;

pdf_strcmp_code: begin save_scanner_status  $\leftarrow$  scanner_status; save_warning_index  $\leftarrow$  warning_index;
save_def_ref  $\leftarrow$  def_ref; save_cur_string; compare_strings; def_ref  $\leftarrow$  save_def_ref;
warning_index  $\leftarrow$  save_warning_index; scanner_status  $\leftarrow$  save_scanner_status; restore_cur_string;
end;

pdf_colorstack_init_code: begin bool  $\leftarrow$  scan_keyword("page");
if scan_keyword("direct") then cur_val  $\leftarrow$  direct_always
else if scan_keyword("page") then cur_val  $\leftarrow$  direct_page
  else cur_val  $\leftarrow$  set_origin;
save_scanner_status  $\leftarrow$  scanner_status; save_warning_index  $\leftarrow$  warning_index; save_def_ref  $\leftarrow$  def_ref;
save_cur_string; scan_pdf_ext_toks; s  $\leftarrow$  tokens_to_string(def_ref); delete_token_ref(def_ref);
def_ref  $\leftarrow$  save_def_ref; warning_index  $\leftarrow$  save_warning_index; scanner_status  $\leftarrow$  save_scanner_status;
cur_val  $\leftarrow$  newcolorstack(s, cur_val, bool); flush_str(s); cur_val_level  $\leftarrow$  int_val;
if cur_val < 0 then
  begin print_err("Too many color stacks");
  help2("The number of color stacks is unlimited up to 32768.")
  ("I'll use the default color stack 0 here."); error; cur_val  $\leftarrow$  0; restore_cur_string;
  end;
end;

job_name_code: if job_name = 0 then open_log_file;
uniform_deviate_code: scan_int;
normal_deviate_code: do_nothing;
pdf_insert_ht_code: scan_register_num;
pdf_ximage_bbox_code: begin scan_int; pdf_check_obj(obj_type_ximage, cur_val);
i  $\leftarrow$  obj_ximage_data(cur_val); scan_int; j  $\leftarrow$  cur_val;
if (j < 1)  $\vee$  (j > 4) then pdf_error("pdfximagebbox", "invalid parameter");
end;
end { there are no other cases }

```

This code is used in section 496.

498. *< Print the result of command c 498 >* ≡

```

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);
  pdftex_revision_code: print(pdftex_revision);
  pdftex_banner_code: print(pdftex_banner);
  pdf_font_name_code, pdf_font_objnum_code: begin set_ff(cur_val);
    if c = pdf_font_name_code then print_int(obj_info(pdf_font_num[ff]))
    else print_int(pdf_font_num[ff]);
    end;
  pdf_font_size_code: begin print_scaled(font_size[cur_val]); print("pt");
  end;
  pdf_page_ref_code: print_int(get_obj(obj_type_page, cur_val, false));
  left_margin_kern_code: begin p ← list_ptr(p);
    while (p ≠ null) ∧ (cp_skipable(p) ∨ ((¬is_char_node(p)) ∧ (type(p) = glue_node) ∧ (subtype(p) =
      left_skip_code + 1))) do p ← link(p);
    if (p ≠ null) ∧ (¬is_char_node(p)) ∧ (type(p) = margin_kern_node) ∧ (subtype(p) = left_side) then
      print_scaled(width(p))
    else print("0");
    print("pt");
  end;
  right_margin_kern_code: begin q ← list_ptr(p); p ← prev_rightmost(q, null);
    while (p ≠ null) ∧ (cp_skipable(p) ∨ ((¬is_char_node(p)) ∧ (type(p) = glue_node) ∧ (subtype(p) =
      right_skip_code + 1))) do p ← prev_rightmost(q, p);
    if (p ≠ null) ∧ (¬is_char_node(p)) ∧ (type(p) = margin_kern_node) ∧ (subtype(p) = right_side) then
      print_scaled(width(p))
    else print("0");
    print("pt");
  end;
  pdf_xform_name_code: print_int(obj_info(cur_val));
  pdf_strcmp_code: print_int(cur_val);
  pdf_colorstack_init_code: print_int(cur_val);
  uniform_deviate_code: print_int(unif_rand(cur_val));
  normal_deviate_code: print_int(norm_rand);
  pdf_insert_ht_code: begin i ← qi(cur_val); p ← page_ins_head;
    while i ≥ subtype(link(p)) do p ← link(p);
    if subtype(p) = i then print_scaled(height(p))
    else print("0");
    print("pt");
  end;
  pdf_ximage_bbox_code: begin if is_pdf_image(i) then
    begin case j of
      1: print_scaled(epdf_orig_x(i));

```

```

2: print_scaled(epdf_orig_y(i));
3: print_scaled(epdf_orig_x(i) + image_width(i));
4: print_scaled(epdf_orig_y(i) + image_height(i));
endcases;
end
else print_scaled(0);
print("pt");
end;
job_name_code: print(job_name);
end { there are no other cases }

```

This code is used in section 496.

499. Now we can't postpone the difficulties any longer; we must bravely tackle *scan_toks*. This function returns a pointer to the tail of a new token list, and it also makes *def_ref* point to the reference count at the head of that list.

There are two boolean parameters, *macro_def* and *xpand*. If *macro_def* is true, the goal is to create the token list for a macro definition; otherwise the goal is to create the token list for some other TeX primitive: *\mark*, *\output*, *\everypar*, *\lowercase*, *\uppercase*, *\message*, *\errmessage*, *\write*, or *\special*. In the latter cases a left brace must be scanned next; this left brace will not be part of the token list, nor will the matching right brace that comes at the end. If *xpand* is false, the token list will simply be copied from the input using *get_token*. Otherwise all expandable tokens will be expanded until unexpandable tokens are left, except that the results of expanding '*\the*' are not expanded further. If both *macro_def* and *xpand* are true, the expansion applies only to the macro body (i.e., to the material following the first *left_brace* character).

The value of *cur_cs* when *scan_toks* begins should be the *eqtb* address of the control sequence to display in "runaway" error messages.

```

function scan_toks(macro_def, xpand : boolean): pointer;
label found, continue, done, done1, done2;
var t: halfword; { token representing the highest parameter number }
    s: halfword; { saved token }
    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 }
    hash_brace: halfword; { possible '#{' token }

begin if macro_def then scanner_status ← defining else scanner_status ← absorbing;
warning_index ← cur_cs; def_ref ← get_avail; token_ref_count(def_ref) ← null; p ← def_ref;
hash_brace ← 0; t ← zero_token;
if macro_def then { Scan and build the parameter part of the macro definition 500 }
else scan_left_brace; { remove the compulsory left brace }
{ Scan and build the body of the token list; goto found when finished 503 };
found: scanner_status ← normal;
if hash_brace ≠ 0 then store_new_token(hash_brace);
scan_toks ← p;
end;

```

500. ⟨ Scan and build the parameter part of the macro definition 500 ⟩ ≡

```

begin loop
  begin continue: get_token; { set cur_cmd, cur_chr, cur_tok }
  if cur_tok < right_brace_limit then goto done1;
  if cur_cmd = mac_param then ⟨ If the next character is a parameter number, make cur_tok a match
      token; but if it is a left brace, store 'left_brace, end_match', set hash_brace, and goto done 502 ⟩;
    store_new_token(cur_tok);
  end;
done1: store_new_token(end_match_token);
  if cur_cmd = right_brace then ⟨ Express shock at the missing left brace; goto found 501 ⟩;
done: end

```

This code is used in section 499.

501. ⟨ Express shock at the missing left brace; goto found 501 ⟩ ≡

```

begin print_err("Missing\{ inserted"); incr(align_state);
help2("Where\was\the\leftbrace?\You\said\something\like`\def\{`,")
("which\I'm\going\to\interpret\as`\def\{`."); error; goto found;
end

```

This code is used in section 500.

502. ⟨ If the next character is a parameter number, make cur_tok a match token; but if it is a left brace,
 store 'left_brace, end_match', set hash_brace, and goto done 502 ⟩ ≡

```

begin s ← match_token + cur_chr; get_token;
if cur_tok < left_brace_limit then
  begin hash_brace ← cur_tok; store_new_token(cur_tok); store_new_token(end_match_token);
  goto done;
  end;
if t = zero_token + 9 then
  begin print_err("You\already\have\nine\parameters");
  help2("I'm\going\to\ignore\the#\sign\you\just\used,")
  ("as\well\as\the\token\that\followed\it."); error; goto continue;
  end
else begin incr(t);
  if cur_tok ≠ t then
    begin print_err("Parameters\must\be\numbered\consecutively");
    help2("I've\inserted\the\digit\you\should\have\used\after\the\#.")
    ("Type\`1\ToDelete\what\you\did\use."); back_error;
    end;
  cur_tok ← s;
  end;
end

```

This code is used in section 500.

503. *< Scan and build the body of the token list; goto found when finished 503 > ≡
 unbalance ← 1;
 loop begin if xpand then < Expand the next part of the input 504 >
 else get_token;
 if cur_tok < right_brace_limit then
 if cur_cmd < right_brace then incr(unbalance)
 else begin decrec(unbalance);
 if unbalance = 0 then goto found;
 end
 else if cur_cmd = mac_param then
 if macro_def then < Look for parameter number or ## 505 >;
 store_new_token(cur_tok);
 end*

This code is used in section 499.

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

*< Expand the next part of the input 504 > ≡
 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 503.

505. *< Look for parameter number or ## 505 > ≡*

```
begin s ← cur_tok;  

if xpand then get_x_token  

else get_token;  

if cur_cmd ≠ mac_param then  

if (cur Tok ≤ zero_token) ∨ (cur Tok > t) then  

begin print_err("Illegal_parameter_number_in_definition_of_"); sprint_cs(warning_index);  

  help3("You_meant_to_type##_instead_of_#,_right?")  

  ("Or_maybe_a_}_was_forgotten_somewhere_earlier,_and_things")  

  ("are_all_screwed_up?_I_m_going_to_assume_that_you_meant_##."); back_error; cur Tok ← s;  

end  

else cur Tok ← out_param_token - "0" + cur Chr;  

end
```

This code is used in section 503.

506. Another way to create a token list is via the `\read` command. The sixteen files potentially usable for reading appear in the following global variables. The value of `read_open[n]` will be *closed* if stream number *n* has not been opened or if it has been fully read; *just_open* if an `\openin` but not a `\read` has been done; and *normal* if it is open and ready to read the next line.

```
define closed = 2 { not open, or at end of file }
define just_open = 1 { newly opened, first line not yet read }
⟨ Global variables 13 ⟩ +≡
read_file: array [0 .. 15] of alpha_file; { used for \read }
read_open: array [0 .. 16] of normal .. closed; { state of read_file[n] }
```

507. ⟨ Set initial values of key variables 21 ⟩ +≡
`for k ← 0 to 16 do read_open[k] ← closed;`

508. 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) ∨ (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 509 ⟩;
until align_state = 1000000;
cur_val ← def_ref; scanner_status ← normal; align_state ← s;
end;
```

509. *< Input and store tokens from the next line of the file 509 >* ≡

```

begin_file_reading; name ← m + 1;
if read_open[m] = closed then < Input for \read from the terminal 510 >
else if read_open[m] = just_open then < Input the first line of read_file[m] 511 >
else < Input the next line of read_file[m] 512 >;
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 1761 >;
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 508.

510. Here we input on-line into the *buffer* array, prompting the user explicitly if $n \geq 0$. The value of n is set negative so that additional prompts will not be given in the case of multi-line input.

< Input for \read from the terminal 510 > ≡

```

if interaction > nonstop_mode then
  if n < 0 then prompt_input("")
  else begin wake_up_terminal; print_ln; sprint_cs(r); prompt_input("=");
    n ← -1;
  end
else fatal_error("*** (cannot \read from terminal in nonstop modes)")

```

This code is used in section 509.

511. The first line of a file must be treated specially, since *input_ln* must be told not to start with *get*.

< Input the first line of read_file[m] 511 > ≡

```

if input_ln(read_file[m], false) then read_open[m] ← normal
else begin a_close(read_file[m]); read_open[m] ← closed;
end

```

This code is used in section 509.

512. An empty line is appended at the end of a *read_file*.

< Input the next line of read_file[m] 512 > ≡

```

begin if ¬input_ln(read_file[m], true) then
  begin a_close(read_file[m]); read_open[m] ← closed;
  if align_state ≠ 1000000 then
    begin runaway; print_err("File ended within"); print_esc("read");
    help1("This \read has unbalanced braces."); align_state ← 1000000; limit ← 0; error;
    end;
  end;
end

```

This code is used in section 509.

513. 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' }
define if_pdfprimitive_code = 21 { '\ifpdfprimitive' }

⟨ Put each of TeX's primitives into the hash table 244 ⟩ +≡
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); primitive("ifpdfprimitive", if_test, if_pdfprimitive_code);
```

```

514. { Cases of print_cmd_chr for symbolic printing of primitives 245 } +≡
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");
    if_pdfprimitive_code: print_esc("ifpdfprimitive");
    { Cases of if_test for print_cmd_chr 1764 }
  othercases print_esc("if")
  endcases;
end;

```

515. Conditions can be inside conditions, and this nesting has a stack that is independent of the *save_stack*.

Four global variables represent the top of the condition stack: *cond_ptr* points to pushed-down entries, if any; *if_limit* specifies the largest code of a *fi_or_else* command that is syntactically legal; *cur_if* is the name of the current type of conditional; and *if_line* is the line number at which it began.

If no conditions are currently in progress, the condition stack has the special state *cond_ptr* = *null*, *if_limit* = *normal*, *cur_if* = 0, *if_line* = 0. Otherwise *cond_ptr* points to a two-word node; the *type*, *subtype*, and *link* fields of the first word contain *if_limit*, *cur_if*, and *cond_ptr* at the next level, and the second word contains the corresponding *if_line*.

```

define if_node_size = 2 { number of words in stack entry for conditionals }
define if_line_field(#) ≡ mem[# + 1].int
define if_code = 1 { code for \if... being evaluated }
define fi_code = 2 { code for \fi }
define else_code = 3 { code for \else }
define or_code = 4 { code for \or }

{ Global variables 13 } +≡
cond_ptr: pointer; { top of the condition stack }
if_limit: normal .. or_code; { upper bound on fi_or_else codes }
cur_if: small_number; { type of conditional being worked on }
if_line: integer; { line where that conditional began }

```

516. { Set initial values of key variables 21 } +≡
cond_ptr ← *null*; *if_limit* ← *normal*; *cur_if* ← 0; *if_line* ← 0;

517. { Put each of TeX's primitives into the hash table 244 } +≡
primitive("fi", *fi_or_else*, *fi_code*); *text*(*frozen_fi*) ← "fi"; *eqtb*[*frozen_fi*] ← *eqtb*[*cur_val*];
primitive("or", *fi_or_else*, *or_code*); *primitive*("else", *fi_or_else*, *else_code*);

518. *(Cases of print_cmd_chr for symbolic printing of primitives 245) +≡*
`fi_or_else: if chr_code = fi_code then print_esc("fi")
 else if chr_code = or_code then print_esc("or")
 else print_esc("else");`

519. When we skip conditional text, we keep track of the line number where skipping began, for use in error messages.

(Global variables 13) +≡
`skip_line: integer; { skipping began here }`

520. Here is a procedure that ignores text until coming to an \or, \else, or \fi at the current level 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;
```

521. When we begin to process a new \if, we set if_limit ← if_code; then if \or or \else or \fi occurs before the current \if condition has been evaluated, \relax will be inserted. For example, a sequence of commands like '\ifvoid1\else...\fi' would otherwise require something after the '1'.

(Push the condition stack 521) ≡
`begin p ← get_node(if_node_size); link(p) ← cond_ptr; type(p) ← if_limit; subtype(p) ← cur_if;
 if_line_field(p) ← if_line; cond_ptr ← p; cur_if ← cur_chr; if_limit ← if_code; if_line ← line;
 end`

This code is used in section 524.

522. *(Pop the condition stack 522) ≡*
`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 524, 526, 535, and 536.

523. Here's a procedure that changes the *if_limit* code corresponding to a given value of *cond_ptr*.

```
procedure change_if_limit(l : small_number; p : pointer);
label exit;
var q: pointer;
begin if p = cond_ptr then if_limit ← l { that's the easy case }
else begin q ← cond_ptr;
loop begin if q = null then confusion("if");
      if link(q) = p then
        begin type(q) ← l; return;
        end;
      q ← link(q);
      end;
end;
exit: end;
```

524. 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? }
e: boolean; { keep track of nested csnames }
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 ≤ 1 then show_cur_cmd_chr;
      { Push the condition stack 521 }; save_cond_ptr ← cond_ptr; is_unless ← (cur_chr ≥ unless_code);
      this_if ← cur_chr mod unless_code;
      { Either process \ifcase or set b to the value of a boolean condition 527 };
      if is_unless then b ← ¬b;
      if tracing_commands > 1 then { Display the value of b 528 };
      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 526 };
common_ending: if cur_chr = fi_code then { Pop the condition stack 522 }
      else if_limit ← fi_code; { wait for \fi }
exit: end;
```

525. In a construction like '\if\iftrue abc\else d\fi', the first \else that we come to after learning that the \if is false is not the \else we're looking for. Hence the following curious logic is needed.

526. ⟨ Skip to `\else` or `\fi`, then `goto common-ending` 526 ⟩ ≡

```

loop begin pass_text;
  if cond_ptr = save_cond_ptr then
    begin if cur_chr ≠ or_code then goto common-ending;
    print_err("Extra\|"); print_esc("or");
    help1("I'm ignoring this; it doesn't match any\|if.\|"); error;
    end
  else if cur_chr = fi_code then ⟨ Pop the condition stack 522 ⟩;
  end

```

This code is used in section 524.

527. ⟨ Either process `\ifcase` or set `b` to the value of a boolean condition 527 ⟩ ≡

```

case this_if of
  if_char_code, if_cat_code: ⟨ Test if two characters match 532 ⟩;
  if_int_code, if_dim_code: ⟨ Test relation between integers or dimensions 529 ⟩;
  if_odd_code: ⟨ Test if an integer is odd 530 ⟩;
  if_vmode_code: b ← (abs(mode) = vmode);
  if_hmode_code: b ← (abs(mode) = hmode);
  if_mmode_code: b ← (abs(mode) = mmode);
  if_inner_code: b ← (mode < 0);
  if_void_code, if_hbox_code, if_vbox_code: ⟨ Test box register status 531 ⟩;
  ifx_code: ⟨ Test if two tokens match 533 ⟩;
  if_eof_code: begin scan_four_bit_int; b ← (read_open[cur_val] = closed);
  end;
  if_true_code: b ← true;
  if_false_code: b ← false;
  ⟨ Cases for conditional 1766 ⟩
  if_case_code: ⟨ Select the appropriate case and return or goto common-ending 535 ⟩;
  if_pdfprimitive_code: begin save_scanner_status ← scanner_status; scanner_status ← normal; get_next;
  scanner_status ← save_scanner_status;
  if cur_cs < hash_base then m ← prim_lookup(cur_cs - single_base)
  else m ← prim_lookup(text(cur_cs));
  b ← ((cur_cmd ≠ undefined_cs) ∧ (m ≠ undefined_primitive) ∧ (cur_cmd = prim_eq_type(m)) ∧ (cur_chr =
  prim_equiv(m)));
  end;
end { there are no other cases }

```

This code is used in section 524.

528. ⟨ Display the value of `b` 528 ⟩ ≡

```

begin begin_diagnostic;
  if b then print("{true}") else print("{false}");
  end_diagnostic(false);
end

```

This code is used in section 524.

529. Here we use the fact that "<", "=", and ">" are consecutive ASCII codes.

```
( Test relation between integers or dimensions 529 ) ≡
begin if this_if = if_int_code then scan_int else scan_normal_dimen;
n ← cur_val; (Get the next non-blank non-call token 432);
if (cur_tok ≥ other_token + "<") ∧ (cur_tok ≤ other_token + ">") then r ← cur_tok − other_token
else begin print_err("Missing=insertedfor"); print_cmd_chr(if_test, this_if);
help1("I was expecting to see `<`, `=`, or `>'. Didn't.");
back_error; r ← "=";
end;
if this_if = if_int_code then scan_int else scan_normal_dimen;
case r of
"<": b ← (n < cur_val);
"=": b ← (n = cur_val);
">": b ← (n > cur_val);
end;
end
```

This code is used in section 527.

530. (Test if an integer is odd 530) ≡

```
begin scan_int; b ← odd(cur_val);
end
```

This code is used in section 527.

531. (Test box register status 531) ≡

```
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 527.

532. An active character will be treated as category 13 following `\if\noexpand` or following `\ifcat\noexpand`.
We use the fact that active characters have the smallest tokens, among all control sequences.

```
define get_x_token_or_active_char ≡
  begin get_x_token;
  if cur_cmd = relax then
    if cur_chr = no_expand_flag then
      begin cur_cmd ← active_char; cur_chr ← cur_tok - cs_token_flag - active_base;
      end;
    end
  end

⟨ Test if two characters match 532 ⟩ ≡
begin get_x_token_or_active_char;
if (cur_cmd > active_char) ∨ (cur_chr > 255) then { not a character }
  begin m ← relax; n ← 256;
  end
else begin m ← cur_cmd; n ← cur_chr;
  end;
get_x_token_or_active_char;
if (cur_cmd > active_char) ∨ (cur_chr > 255) then
  begin cur_cmd ← relax; cur_chr ← 256;
  end;
if this_if = if_char_code then b ← (n = cur_chr) else b ← (m = cur_cmd);
end
```

This code is used in section 527.

533. Note that ‘`\ifx`’ will declare two macros different if one is *long* or *outer* and the other isn’t, even though the texts of the macros are the same.

We need to reset `scanner_status`, since `\outer` control sequences are allowed, but we might be scanning a macro definition or preamble.

```
⟨ Test if two tokens match 533 ⟩ ≡
begin save_scanner_status ← scanner_status; scanner_status ← normal; get_next; n ← cur_cs;
p ← cur_cmd; q ← cur_chr; get_next;
if cur_cmd ≠ p then b ← false
else if cur_cmd < call then b ← (cur_chr = q)
  else ⟨ Test if two macro texts match 534 ⟩;
scanner_status ← save_scanner_status;
end
```

This code is used in section 527.

534. Note also that ‘\ifx’ decides that macros \a and \b are different in examples like this:

```
\def\aa{\c}\def\cc{}\def\bb{\d}\def\dd{}
```

```
< Test if two macro texts match 534 > ≡
begin p ← link(cur_chr); q ← link(equiv(n)); { omit reference counts }
if p = q then b ← true
else begin while (p ≠ null) ∧ (q ≠ null) do
  if info(p) ≠ info(q) then p ← null
  else begin p ← link(p); q ← link(q);
  end;
b ← ((p = null) ∧ (q = null));
end;
end
```

This code is used in section 533.

```
< Select the appropriate case and return or goto common-ending 535 > ≡
begin scan_int; n ← cur_val; { n is the number of cases to pass }
if tracing_commands > 1 then
begin begin_diagnostic; print("{case}"); print_int(n); print_char("}"); end_diagnostic(false);
end;
while n ≠ 0 do
begin pass_text;
if cond_ptr = save_cond_ptr then
  if cur_chr = or_code then decr(n)
  else goto common-ending
else if cur_chr = fi_code then < Pop the condition stack 522 >;
end;
change_if_limit(or_code, save_cond_ptr); return; { wait for \or, \else, or \fi }
end
```

This code is used in section 527.

536. 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 536 > ≡
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 522 >;
  end;
end
```

This code is used in section 391.

537. File names. It's time now to fret about file names. Besides the fact that different operating systems treat files in different ways, we must cope with the fact that completely different naming conventions are used by different groups of people. The following programs show what is required for one particular operating system; similar routines for other systems are not difficult to devise.

TeX assumes that a file name has three parts: the name proper; its "extension"; and a "file area" where it is found in an external file system. The extension of an input file or a write file is assumed to be '.tex' unless otherwise specified; it is '.log' on the transcript file that records each run of TeX; it is '.tfm' on the font metric files that describe characters in the fonts TeX uses; it is '.dvi' on the output files that specify typesetting information; and it is '.fmt' on the format files written by INITEX to initialize TeX. The file area can be arbitrary on input files, but files are usually output to the user's current area. If an input file cannot be found on the specified area, TeX will look for it on a special system area; this special area is intended for commonly used input files like `webmac.tex`.

Simple uses of TeX refer only to file names that have no explicit extension or area. For example, a person usually says '`\input paper`' or '`\font\tenrm = helvetica`' instead of '`\input paper.new`' or '`\font\tenrm = <csd.knuth>test`'. Simple file names are best, because they make the TeX source files portable; whenever a file name consists entirely of letters and digits, it should be treated in the same way by all implementations of TeX. However, users need the ability to refer to other files in their environment, especially when responding to error messages concerning unopenable files; therefore we want to let them use the syntax that appears in their favorite operating system.

The following procedures don't allow spaces to be part of file names; but some users seem to like names that are spaced-out. System-dependent changes to allow such things should probably be made with reluctance, and only when an entire file name that includes spaces is "quoted" somehow.

538. In order to isolate the system-dependent aspects of file names, the system-independent parts of TeX are expressed in terms of three system-dependent procedures called *begin_name*, *more_name*, and *end_name*. In essence, if the user-specified characters of the file name are $c_1 \dots c_n$, the system-independent driver program does the operations

$$\text{begin_name}; \text{more_name}(c_1); \dots; \text{more_name}(c_n); \text{end_name}.$$

These three procedures communicate with each other via global variables. Afterwards the file name will appear in the string pool as three strings called *cur_name*, *cur_area*, and *cur_ext*; the latter two are null (i.e., ""), unless they were explicitly specified by the user.

Actually the situation is slightly more complicated, because TeX needs to know when the file name ends. The *more_name* routine is a function (with side effects) that returns *true* on the calls *more_name*(c_1), ..., *more_name*(c_{n-1}). The final call *more_name*(c_n) returns *false*; or, it returns *true* and the token following c_n is something like '`\hbox`' (i.e., not a character). In other words, *more_name* is supposed to return *true* unless it is sure that the file name has been completely scanned; and *end_name* is supposed to be able to finish the assembly of *cur_name*, *cur_area*, and *cur_ext* regardless of whether *more_name*(c_n) returned *true* or *false*.

```
< Global variables 13 > +≡
cur_name: str_number; { name of file just scanned }
cur_area: str_number; { file area just scanned, or "" }
cur_ext: str_number; { file extension just scanned, or "" }
```

539. The file names we shall deal with for illustrative purposes have the following structure: If the name contains ‘>’ or ‘:’, the file area consists of all characters up to and including the final such character; otherwise the file area is null. If the remaining file name contains ‘.’, the file extension consists of all such characters from the first remaining ‘.’ to the end, otherwise the file extension is null.

We can scan such file names easily by using two global variables that keep track of the occurrences of area and extension delimiters:

```
<Global variables 13> +≡
area_delimiter: pool_pointer; { the most recent ‘>’ or ‘:’, if any }
ext_delimiter: pool_pointer; { the relevant ‘.’, if any }
```

540. Input files that can’t be found in the user’s area may appear in a standard system area called *TEX_area*. Font metric files whose areas are not given explicitly are assumed to appear in a standard system area called *TEX_font_area*. These system area names will, of course, vary from place to place.

```
define TEX_area ≡ "TeXinputs:"
define TEX_font_area ≡ "TeXfonts:"
```

541. Here now is the first of the system-dependent routines for file name scanning.

```
procedure begin_name;
begin area_delimiter ← 0; ext_delimiter ← 0;
end;
```

542. And here’s the second. The string pool might change as the file name is being scanned, since a new \csname might be entered; therefore we keep *area_delimiter* and *ext_delimiter* relative to the beginning of the current string, instead of assigning an absolute address like *pool_ptr* to them.

```
function more_name(c : ASCII_code): boolean;
begin if c = "_" then more_name ← false
else begin str_room(1); append_char(c); { contribute c to the current string }
if (c = ">") ∨ (c = ":") then
begin area_delimiter ← cur_length; ext_delimiter ← 0;
end
else if (c = ".") ∧ (ext_delimiter = 0) then ext_delimiter ← cur_length;
more_name ← true;
end;
end;
```

543. The third.

```
procedure end_name;
begin if str_ptr + 3 > max_strings then overflow("number_of_strings", max_strings - init_str_ptr);
if area_delimiter = 0 then cur_area ← ""
else begin cur_area ← str_ptr; str_start[str_ptr + 1] ← str_start[str_ptr] + area_delimiter; incr(str_ptr);
end;
if ext_delimiter = 0 then
begin cur_ext ← ""; cur_name ← make_string;
end
else begin cur_name ← str_ptr;
str_start[str_ptr + 1] ← str_start[str_ptr] + ext_delimiter - area_delimiter - 1; incr(str_ptr);
cur_ext ← make_string;
end;
end;
```

544. Conversely, here is a routine that takes three strings and prints a file name that might have produced them. (The routine is system dependent, because some operating systems put the file area last instead of first.)

```
< Basic printing procedures 57 > +≡
procedure print_file_name(n, a, e : integer);
begin slow_print(a); slow_print(n); slow_print(e);
end;
```

545. Another system-dependent routine is needed to convert three internal TeX strings into the *name_of_file* value that is used to open files. The present code allows both lowercase and uppercase letters in the file name.

```
define append_to_name(#) ≡
begin c ← #; incr(k);
if k ≤ file_name_size then name_of_file[k] ← xchr[c];
end

procedure pack_file_name(n, a, e : str_number);
var k: integer; { number of positions filled in name_of_file }
c: ASCII_code; { character being packed }
j: pool_pointer; { index into str_pool }
begin k ← 0;
for j ← str_start[a] to str_start[a + 1] - 1 do append_to_name(so(str_pool[j]));
for j ← str_start[n] to str_start[n + 1] - 1 do append_to_name(so(str_pool[j]));
for j ← str_start[e] to str_start[e + 1] - 1 do append_to_name(so(str_pool[j]));
if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
for k ← name_length + 1 to file_name_size do name_of_file[k] ← ' ';
end;
```

546. A messier routine is also needed, since format file names must be scanned before TeX's string mechanism has been initialized. We shall use the global variable *TEX_format_default* to supply the text for default system areas and extensions related to format files.

```
define format_default_length = 20 { length of the TEX_format_default string }
define format_area_length = 11 { length of its area part }
define format_ext_length = 4 { length of its '.fmt' part }
define format_extension = ".fmt" { the extension, as a WEB constant }

< Global variables 13 > +≡
TEX_format_default: packed array [1 .. format_default_length] of char;
```

547. < Set initial values of key variables 21 > +≡
TEX_format_default ← 'TeXformats:plain.fmt';

548. < Check the "constant" values for consistency 14 > +≡
if format_default_length > file_name_size then bad ← 31;

549. Here is the messy routine that was just mentioned. It sets *name_of_file* from the first *n* characters of *TEX_format_default*, followed by *buffer[a .. b]*, followed by the last *format_ext_length* characters of *TEX_format_default*.

We dare not give error messages here, since TeX calls this routine before the *error* routine is ready to roll. Instead, we simply drop excess characters, since the error will be detected in another way when a strange file name isn't found.

```
procedure pack_buffered_name(n : small_number; a, b : integer);
  var k: integer; { number of positions filled in name_of_file }
    c: ASCII_code; { character being packed }
    j: integer; { index into buffer or TEX_format_default }
  begin if n + b - a + 1 + format_ext_length > file_name_size then
    b ← a + file_name_size - n - 1 - format_ext_length;
    k ← 0;
    for j ← 1 to n do append_to_name(xord[TEX_format_default[j]]);
    for j ← a to b do append_to_name(buffer[j]);
    for j ← format_default_length - format_ext_length + 1 to format_default_length do
      append_to_name(xord[TEX_format_default[j]]);
    if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
    for k ← name_length + 1 to file_name_size do name_of_file[k] ← ' ';
  end;
```

550. Here is the only place we use *pack_buffered_name*. This part of the program becomes active when a "virgin" TeX is trying to get going, just after the preliminary initialization, or when the user is substituting another format file by typing '&' after the initial '**' prompt. The buffer contains the first line of input in *buffer*[*loc* .. (*last* - 1)], where *loc* < *last* and *buffer*[*loc*] ≠ ' '.

```
< Declare the function called open_fmt_file 550 > ≡
function open_fmt_file: boolean;
  label found, exit;
  var j: 0 .. buf_size; { the first space after the format file name }
  begin j ← loc;
  if buffer[loc] = "&" then
    begin incr(loc); j ← loc; buffer[last] ← ' ';
    while buffer[j] ≠ ' ' do incr(j);
    pack_buffered_name(0, loc, j - 1); { try first without the system file area }
    if w_open_in(fmt_file) then goto found;
    pack_buffered_name(format_area_length, loc, j - 1); { now try the system format file area }
    if w_open_in(fmt_file) then goto found;
    wake_up_terminal; wterm_ln(`Sorry, I can't find that format; I will try PLAIN.');
    update_terminal;
  end; { now pull out all the stops: try for the system plain file }
  pack_buffered_name(format_default_length - format_ext_length, 1, 0);
  if ¬w_open_in(fmt_file) then
    begin wake_up_terminal; wterm_ln(`I can't find the PLAIN format file!');
    open_fmt_file ← false; return;
  end;
  found: loc ← j; open_fmt_file ← true;
  exit: end;
```

This code is used in section 1481.

551. Operating systems often make it possible to determine the exact name (and possible version number) of a file that has been opened. The following routine, which simply makes a TeX string from the value of *name_of_file*, should ideally be changed to deduce the full name of file *f*, which is the file most recently opened, if it is possible to do this in a Pascal program.

This routine might be called after string memory has overflowed, hence we dare not use '*str_room*'.

```
function make_name_string: str_number;
  var k: 1 .. file_name_size; {index into name_of_file}
begin if (pool_ptr + name_length > pool_size) ∨ (str_ptr = max_strings) ∨ (cur_length > 0) then
  make_name_string ← "?"
else begin for k ← 1 to name_length do append_char(xord[name_of_file[k]]);
  make_name_string ← make_string;
end;
end;

function a_make_name_string(var f : alpha_file): str_number;
begin a_make_name_string ← make_name_string;
end;

function b_make_name_string(var f : byte_file): str_number;
begin b_make_name_string ← make_name_string;
end;

function w_make_name_string(var f : word_file): str_number;
begin w_make_name_string ← make_name_string;
end;
```

552. Now let's consider the "driver" routines by which TeX deals with file names in a system-independent manner. First comes a procedure that looks for a file name in the input by calling *get_x_token* for the information.

```
procedure scan_file_name;
label done;
begin name_in_progress ← true; begin_name; {Get the next non-blank non-call token 432};
loop begin if (cur_cmd > other_char) ∨ (cur_chr > 255) then {not a character}
  begin back_input; goto done;
end;
if ¬more_name(cur_chr) then goto done;
get_x_token;
end;
done: end_name; name_in_progress ← false;
end;
```

553. The global variable *name_in_progress* is used to prevent recursive use of *scan_file_name*, since the *begin_name* and other procedures communicate via global variables. Recursion would arise only by devious tricks like '\input\input f'; such attempts at sabotage must be thwarted. Furthermore, *name_in_progress* prevents \input from being initiated when a font size specification is being scanned.

Another global variable, *job_name*, contains the file name that was first \input by the user. This name is extended by '.log' and '.dvi' and '.fmt' in the names of TeX's output files.

```
< Global variables 13 > +≡
name_in_progress: boolean; {is a file name being scanned?}
job_name: str_number; {principal file name}
log_opened: boolean; {has the transcript file been opened?}
```

554. Initially *job_name* = 0; it becomes nonzero as soon as the true name is known. We have *job_name* = 0 if and only if the ‘log’ file has not been opened, except of course for a short time just after *job_name* has become nonzero.

```
< Initialize the output routines 55 > +≡
  job_name ← 0; name_in_progress ← false; log_opened ← false;
```

555. Here is a routine that manufactures the output file names, assuming that *job_name* ≠ 0. It ignores and changes the current settings of *cur_area* and *cur_ext*.

```
define pack_cur_name ≡ pack_file_name(cur_name, cur_area, cur_ext)
procedure pack_job_name(s : str_number); { s = ".log", ".dvi", or format_extension }
  begin cur_area ← ""; cur_ext ← s; cur_name ← job_name; pack_cur_name;
  end;
```

556. If some trouble arises when TeX tries to open a file, the following routine calls upon the user to supply another file name. Parameter *s* is used in the error message to identify the type of file; parameter *e* is the default extension if none is given. Upon exit from the routine, variables *cur_name*, *cur_area*, *cur_ext*, and *name_of_file* are ready for another attempt at file opening.

```
procedure prompt_file_name(s, e : str_number);
  label done;
  var k: 0 .. buf_size; { index into buffer }
  begin if interaction = scroll_mode then wake_up_terminal;
  if s = "input_file_name" then print_err("I can't find file `")
  else print_err("I can't write on file `");
  print_file_name(cur_name, cur_area, cur_ext); print(`.');
  if e = ".tex" then show_context;
  print_nl("Please type another"); print(s);
  if interaction < scroll_mode then fatal_error("*** (job_aborted, file_error_in_nonstop_mode)");
  clear_terminal; prompt_input(":"); { Scan file name in the buffer 557 };
  if cur_ext = "" then cur_ext ← e;
  pack_cur_name;
  end;
```

557. { Scan file name in the buffer 557 } ≡

```
begin begin_name; k ← first;
while (buffer[k] = " ") ∧ (k < last) do incr(k);
loop begin if k = last then goto done;
  if ¬more_name(buffer[k]) then goto done;
  incr(k);
end;
done: end_name;
end
```

This code is used in section 556.

558. Here's an example of how these conventions are used. Whenever it is time to ship out a box of stuff, we shall use the macro *ensure_dvi_open*.

```
define ensure_dvi_open ==
  if output_file_name = 0 then
    begin if job_name = 0 then open_log_file;
    pack_job_name(".dvi");
    while ¬b_open_out(dvi_file) do prompt_file_name("file_name_for_output", ".dvi");
    output_file_name ← b_make_name_string(dvi_file);
  end
⟨ Global variables 13 ⟩ +≡
dvi_file: byte_file; { the device-independent output goes here }
output_file_name: str_number; { full name of the output file }
log_name: str_number; { full name of the log file }
```

559. ⟨ Initialize the output routines 55 ⟩ +≡
output_file_name ← 0;

560. The *open_log_file* routine is used to open the transcript file and to help it catch up to what has previously been printed on the terminal.

```
procedure open_log_file;
  var old_setting: 0 .. max_selector; { previous selector setting }
  k: 0 .. buf_size; { index into months and buffer }
  l: 0 .. buf_size; { end of first input line }
  months: packed array [1 .. 36] of char; { abbreviations of month names }
begin old_setting ← selector;
if job_name = 0 then job_name ← "texput";
pack_job_name(".log");
while ¬a_open_out(log_file) do ⟨ Try to get a different log file name 561 ⟩;
log_name ← a_make_name_string(log_file); selector ← log_only; log_opened ← true;
⟨ Print the banner line, including the date and time 562 ⟩;
input_stack[input_ptr] ← cur_input; { make sure bottom level is in memory }
print_nl("##"); l ← input_stack[0].limit_field; { last position of first line }
if buffer[l] = end_line_char then decr(l);
for k ← 1 to l do print(buffer[k]);
print_ln; { now the transcript file contains the first line of input }
selector ← old_setting + 2; { log_only or term_and_log }
end;
```

561. Sometimes *open_log_file* is called at awkward moments when TeX is unable to print error messages or even to *show_context*. The *prompt_file_name* routine can result in a *fatal_error*, but the *error* routine will not be invoked because *log_opened* will be false.

The normal idea of *batch_mode* is that nothing at all should be written on the terminal. However, in the unusual case that no log file could be opened, we make an exception and allow an explanatory message to be seen.

Incidentally, the program always refers to the log file as a 'transcript file', because some systems cannot use the extension '.log' for this file.

```
⟨ Try to get a different log file name 561 ⟩ ≡
begin selector ← term_only; prompt_file_name("transcript_file_name", ".log");
end
```

This code is used in section 560.

562. { Print the banner line, including the date and time 562 } \equiv

```

begin wlog(banner); slow_print(format_ident); print("„"); print_int(sys_day); print_char("„");
months ← `JANFEBMARAPR MAY JUN JUL AUG SEPOCT NOV DEC`;
for k ← 3 * sys_month - 2 to 3 * sys_month do wlog(months[k]);
print_char("„"); print_int(sys_year); print_char("„"); print_two(sys_time div 60); print_char(":");
print_two(sys_time mod 60);
if eTeX_ex then
  begin ; wlog_cr; wlog(`entering„extended„mode`);
  end;
end

```

This code is used in section 560.

563. Let's turn now to the procedure that is used to initiate file reading when an '\input' command is being processed. Beware: For historic reasons, this code foolishly conserves a tiny bit of string pool space; but that can confuse the interactive 'E' option.

```

procedure start_input; { TEX will \input something }
label done;
begin scan_file_name; { set cur_name to desired file name }
if cur_ext = "" then cur_ext ← ".tex";
pack_cur_name;
loop begin begin_file_reading; { set up cur_file and new level of input }
if a_open_in(cur_file) then goto done;
if cur_area = "" then
  begin pack_file_name(cur_name, TEX_area, cur_ext);
  if a_open_in(cur_file) then goto done;
  end;
end_file_reading; { remove the level that didn't work }
prompt_file_name("input„file„name", ".tex");
end;
done: name ← a_make_name_string(cur_file);
if job_name = 0 then
  begin job_name ← cur_name; open_log_file;
  end; { open_log_file doesn't show_context, so limit and loc needn't be set to meaningful values yet }
if term_offset + length(name) > max_print_line - 2 then print_ln
else if (term_offset > 0) ∨ (file_offset > 0) then print_char("„");
print_char("("); incr(open_parens); slow_print(name); update_terminal; state ← new_line;
if name = str_ptr - 1 then { conserve string pool space (but see note above) }
  begin flush_string; name ← cur_name;
  end;
{ Read the first line of the new file 564 };
end;

```

564. Here we have to remember to tell the *input_ln* routine not to start with a *get*. If the file is empty, it is considered to contain a single blank line.

⟨ Read the first line of the new file 564 ⟩ ≡

```
begin line ← 1;
if input_ln(cur_file, false) then do_nothing;
firm_up_the_line;
if end_line_char_inactive then decr(limit)
else buffer[limit] ← end_line_char;
first ← limit + 1; loc ← start;
end
```

This code is used in section 563.

565. Font metric data. TeX gets its knowledge about fonts from font metric files, also called TFM files; the ‘T’ in ‘TFM’ stands for TeX, but other programs know about them too.

The information in a TFM file appears in a sequence of 8-bit bytes. Since the number of bytes is always a multiple of 4, we could also regard the file as a sequence of 32-bit words, but TeX uses the byte interpretation. The format of TFM files was designed by Lyle Ramshaw in 1980. The intent is to convey a lot of different kinds of information in a compact but useful form.

`<Global variables 13> +≡
tfm-file: byte-file;`

566. The first 24 bytes (6 words) of a TFM file contain twelve 16-bit integers that give the lengths of the various subsequent portions of the file. These twelve integers are, in order:

lf = length of the entire file, in words;
lh = length of the header data, in words;
bc = smallest character code in the font;
ec = largest character code in the font;
nw = number of words in the width table;
nh = number of words in the height table;
nd = number of words in the depth table;
ni = number of words in the italic correction table;
nl = number of words in the lig/kern table;
nk = number of words in the kern table;
ne = number of words in the extensible character table;
np = number of font parameter words.

They are all nonnegative and less than 2^{15} . We must have $bc - 1 \leq ec \leq 255$, and

$$lf = 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.$$

Note that a font may contain as many as 256 characters (if $bc = 0$ and $ec = 255$), and as few as 0 characters (if $bc = ec + 1$).

Incidentally, when two or more 8-bit bytes are combined to form an integer of 16 or more bits, the most significant bytes appear first in the file. This is called BigEndian order.

567. The rest of the TFM file may be regarded as a sequence of ten data arrays having the informal specification

```
header : array [0 .. lh - 1] of stuff
char_info : array [bc .. ec] of char_info_word
width : array [0 .. nw - 1] of fix_word
height : array [0 .. nh - 1] of fix_word
depth : array [0 .. nd - 1] of fix_word
italic : array [0 .. ni - 1] of fix_word
lig_kern : array [0 .. nl - 1] of lig_kern_command
kern : array [0 .. nk - 1] of fix_word
exten : array [0 .. ne - 1] of extensible_recipe
param : array [1 .. np] of fix_word
```

The most important data type used here is a *fix_word*, which is a 32-bit representation of a binary fraction. A *fix_word* is a signed quantity, with the two’s complement of the entire word used to represent negation. Of the 32 bits in a *fix_word*, exactly 12 are to the left of the binary point; thus, the largest *fix_word* value is $2048 - 2^{-20}$, and the smallest is -2048 . We will see below, however, that all but two of the *fix_word* values must lie between -16 and $+16$.

568. The first data array is a block of header information, which contains general facts about the font. The header must contain at least two words, *header*[0] and *header*[1], whose meaning is explained below. Additional header information of use to other software routines might also be included, but TeX82 does not need to know about such details. For example, 16 more words of header information are in use at the Xerox Palo Alto Research Center; the first ten specify the character coding scheme used (e.g., ‘XEROX text’ or ‘TeX math symbols’), the next five give the font identifier (e.g., ‘HELVETICA’ or ‘CMSY’), and the last gives the “face byte.” The program that converts DVI files to Xerox printing format gets this information by looking at the TFM file, which it needs to read anyway because of other information that is not explicitly repeated in DVI format.

header[0] is a 32-bit check sum that TeX will copy into the DVI output file. Later on when the DVI file is printed, possibly on another computer, the actual font that gets used is supposed to have a check sum that agrees with the one in the TFM file used by TeX. In this way, users will be warned about potential incompatibilities. (However, if the check sum is zero in either the font file or the TFM file, no check is made.) The actual relation between this check sum and the rest of the TFM file is not important; the check sum is simply an identification number with the property that incompatible fonts almost always have distinct check sums.

header[1] is a *fix-word* containing the design size of the font, in units of TeX points. This number must be at least 1.0; it is fairly arbitrary, but usually the design size is 10.0 for a “10 point” font, i.e., a font that was designed to look best at a 10-point size, whatever that really means. When a TeX user asks for a font ‘at δ pt’, the effect is to override the design size and replace it by δ , and to multiply the *x* and *y* coordinates of the points in the font image by a factor of δ divided by the design size. All other dimensions in the TFM file are *fix-word* numbers in design-size units, with the exception of *param*[1] (which denotes the slant ratio). Thus, for example, the value of *param*[6], which defines the *em* unit, is often the *fix-word* value $2^{20} = 1.0$, since many fonts have a design size equal to one em. The other dimensions must be less than 16 design-size units in absolute value; thus, *header*[1] and *param*[1] are the only *fix-word* entries in the whole TFM file whose first byte might be something besides 0 or 255.

569. Next comes the *char_info* array, which contains one *char_info_word* per character. Each word in this part of the file contains six fields packed into four bytes as follows.

first byte: *width_index* (8 bits)

second byte: *height_index* (4 bits) times 16, plus *depth_index* (4 bits)

third byte: *italic_index* (6 bits) times 4, plus *tag* (2 bits)

fourth byte: *remainder* (8 bits)

The actual width of a character is *width*[*width_index*], in design-size units; this is a device for compressing information, since many characters have the same width. Since it is quite common for many characters to have the same height, depth, or italic correction, the TFM format imposes a limit of 16 different heights, 16 different depths, and 64 different italic corrections.

The italic correction of a character has two different uses. (a) In ordinary text, the italic correction is added to the width only if the TeX user specifies ‘\v’ after the character. (b) In math formulas, the italic correction is always added to the width, except with respect to the positioning of subscripts.

Incidentally, the relation *width*[0] = *height*[0] = *depth*[0] = *italic*[0] = 0 should always hold, so that an index of zero implies a value of zero. The *width_index* should never be zero unless the character does not exist in the font, since a character is valid if and only if it lies between *bc* and *ec* and has a nonzero *width_index*.

- 570.** The *tag* field in a *char_info_word* has four values that explain how to interpret the *remainder* field.
tag = 0 (*no_tag*) means that *remainder* is unused.
tag = 1 (*lig_tag*) means that this character has a ligature/kerning program starting at position *remainder* in the *lig_kern* array.
tag = 2 (*list_tag*) means that this character is part of a chain of characters of ascending sizes, and not the largest in the chain. The *remainder* field gives the character code of the next larger character.
tag = 3 (*ext_tag*) means that this character code represents an extensible character, i.e., a character that is built up of smaller pieces so that it can be made arbitrarily large. The pieces are specified in *exten*[*remainder*].

Characters with *tag* = 2 and *tag* = 3 are treated as characters with *tag* = 0 unless they are used in special circumstances in math formulas. For example, the *\sum* operation looks for a *list_tag*, and the *\left* operation looks for both *list_tag* and *ext_tag*.

```
define no_tag = 0 { vanilla character }
define lig_tag = 1 { character has a ligature/kerning program }
define list_tag = 2 { character has a successor in a charlist }
define ext_tag = 3 { character is extensible }
```

571. The *lig_kern* array contains instructions in a simple programming language that explains what to do for special letter pairs. Each word in this array is a *lig_kern_command* of four bytes.

first byte: *skip_byte*, indicates that this is the final program step if the byte is 128 or more, otherwise the next step is obtained by skipping this number of intervening steps.

second byte: *next_char*, “if *next_char* follows the current character, then perform the operation and stop, otherwise continue.”

third byte: *op_byte*, indicates a ligature step if less than 128, a kern step otherwise.

fourth byte: *remainder*.

In a kern step, an additional space equal to $kern[256 * (op_byte - 128) + remainder]$ is inserted between the current character and *next_char*. This amount is often negative, so that the characters are brought closer together by kerning; but it might be positive.

There are eight kinds of ligature steps, having *op_byte* codes $4a + 2b + c$ where $0 \leq a \leq b + c$ and $0 \leq b, c \leq 1$. The character whose code is *remainder* is inserted between the current character and *next_char*; then the current character is deleted if $b = 0$, and *next_char* is deleted if $c = 0$; then we pass over a characters to reach the next current character (which may have a ligature/kerning program of its own).

If the very first instruction of the *lig_kern* array has *skip_byte* = 255, the *next_char* byte is the so-called boundary character of this font; the value of *next_char* need not lie between *bc* and *ec*. If the very last instruction of the *lig_kern* array has *skip_byte* = 255, there is a special ligature/kerning program for a boundary character at the left, beginning at location $256 * op_byte + remainder$. The interpretation is that TeX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character’s *lig_kern* program has *skip_byte* > 128, the program actually begins in location $256 * op_byte + remainder$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255 .

Any instruction with *skip_byte* > 128 in the *lig_kern* array must satisfy the condition

$$256 * op_byte + remainder < nl.$$

If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature or kerning command is performed.

```
define stop_flag ≡ qi(128) { value indicating 'STOP' in a lig/kern program }
define kern_flag ≡ qi(128) { op code for a kern step }
define skip_byte(#) ≡ #.b0
define next_char(#) ≡ #.b1
define op_byte(#) ≡ #.b2
define rem_byte(#) ≡ #.b3
```

572. Extensible characters are specified by an *extensible_recipe*, which consists of four bytes called *top*, *mid*, *bot*, and *rep* (in this order). These bytes are the character codes of individual pieces used to build up a large symbol. If *top*, *mid*, or *bot* are zero, they are not present in the built-up result. For example, an extensible vertical line is like an extensible bracket, except that the top and bottom pieces are missing.

Let *T*, *M*, *B*, and *R* denote the respective pieces, or an empty box if the piece isn’t present. Then the extensible characters have the form $TR^k MR^k B$ from top to bottom, for some $k \geq 0$, unless *M* is absent; in the latter case we can have $TR^k B$ for both even and odd values of *k*. The width of the extensible character is the width of *R*; and the height-plus-depth is the sum of the individual height-plus-depths of the components used, since the pieces are butted together in a vertical list.

```
define ext_top(#) ≡ #.b0 { top piece in a recipe }
define ext_mid(#) ≡ #.b1 { mid piece in a recipe }
define ext_bot(#) ≡ #.b2 { bot piece in a recipe }
define ext_rep(#) ≡ #.b3 { rep piece in a recipe }
```

573. The final portion of a TFM file is the *param* array, which is another sequence of *fix_word* values.

param[1] = *slant* is the amount of italic slant, which is used to help position accents. For example, *slant* = .25 means that when you go up one unit, you also go .25 units to the right. The *slant* is a pure number; it's the only *fix_word* other than the design size itself that is not scaled by the design size.

param[2] = *space* is the normal spacing between words in text. Note that character " \sqcup " in the font need not have anything to do with blank spaces.

param[3] = *space_stretch* is the amount of glue stretching between words.

param[4] = *space_shrink* is the amount of glue shrinking between words.

param[5] = *x_height* is the size of one ex in the font; it is also the height of letters for which accents don't have to be raised or lowered.

param[6] = *quad* is the size of one em in the font.

param[7] = *extra_space* is the amount added to *param*[2] at the ends of sentences.

If fewer than seven parameters are present, TeX sets the missing parameters to zero. Fonts used for math symbols are required to have additional parameter information, which is explained later.

```
define slant_code = 1
define space_code = 2
define space_stretch_code = 3
define space_shrink_code = 4
define x_height_code = 5
define quad_code = 6
define extra_space_code = 7
```

574. So that is what TFM files hold. Since TeX has to absorb such information about lots of fonts, it stores most of the data in a large array called *font_info*. Each item of *font_info* is a *memory_word*; the *fix_word* data gets converted into *scaled* entries, while everything else goes into words of type *four_quarters*.

When the user defines \font\f, say, TeX assigns an internal number to the user's font \f. Adding this number to *font_id_base* gives the *eqtb* location of a "frozen" control sequence that will always select the font.

<Types in the outer block 18> +≡

```
internal_font_number = font_base .. font_max; { font in a char_node }
font_index = 0 .. font_mem_size; { index into font_info }
```

575. Here now is the (rather formidable) array of font arrays.

```

define non_char ≡ qi(256) { a halfword code that can't match a real character }
define non_address = 0 { a spurious bchar_label }

{ Global variables 13 } +≡
font_info: array [font_index] of memory_word; { the big collection of font data }
fmem_ptr: font_index; { first unused word of font_info }
font_ptr: internal_font_number; { largest internal font number in use }
font_check: array [internal_font_number] of four_quarters; { check sum }
font_size: array [internal_font_number] of scaled; { "at" size }
font_dsize: array [internal_font_number] of scaled; { "design" size }
font_params: array [internal_font_number] of font_index; { how many font parameters are present }
font_name: array [internal_font_number] of str_number; { name of the font }
font_area: array [internal_font_number] of str_number; { area of the font }
font_bc: array [internal_font_number] of eight_bits; { beginning (smallest) character code }
font_ec: array [internal_font_number] of eight_bits; { ending (largest) character code }
font_glue: array [internal_font_number] of pointer;
    { glue specification for interword space, null if not allocated }
font_used: array [internal_font_number] of boolean;
    { has a character from this font actually appeared in the output? }
hyphen_char: array [internal_font_number] of integer; { current \hyphenchar values }
skew_char: array [internal_font_number] of integer; { current \skewchar values }
bchar_label: array [internal_font_number] of font_index;
    { start of lig_kern program for left boundary character, non_address if there is none }
font_bchar: array [internal_font_number] of min_quarterword .. non_char;
    { boundary character, non_char if there is none }
font_false_bchar: array [internal_font_number] of min_quarterword .. non_char;
    { font_bchar if it doesn't exist in the font, otherwise non_char }

```

576. Besides the arrays just enumerated, we have directory arrays that make it easy to get at the individual entries in *font_info*. For example, the *char_info* data for character *c* in font *f* will be in *font_info*[*char_base*[*f*] + *c*].*qqqq*; and if *w* is the *width_index* part of this word (the *b0* field), the width of the character is *font_info*[*width_base*[*f*] + *w*].*sc*. (These formulas assume that *min_quarterword* has already been added to *c* and to *w*, since TeX stores its quarterwords that way.)

```

{ Global variables 13 } +≡
char_base: array [internal_font_number] of integer; { base addresses for char_info }
width_base: array [internal_font_number] of integer; { base addresses for widths }
height_base: array [internal_font_number] of integer; { base addresses for heights }
depth_base: array [internal_font_number] of integer; { base addresses for depths }
italic_base: array [internal_font_number] of integer; { base addresses for italic corrections }
lig_kern_base: array [internal_font_number] of integer; { base addresses for ligature/kerning programs }
kern_base: array [internal_font_number] of integer; { base addresses for kerns }
exten_base: array [internal_font_number] of integer; { base addresses for extensible recipes }
param_base: array [internal_font_number] of integer; { base addresses for font parameters }

```

577. { Set initial values of key variables 21 } +≡

```

for k ← font_base to font_max do font_used[k] ← false;

```

578. TeX always knows at least one font, namely the null font. It has no characters, and its seven parameters are all equal to zero.

$\langle \text{Initialize table entries (done by INITEX only) } 182 \rangle +\equiv$

```
font_ptr  $\leftarrow$  null_font; fmem_ptr  $\leftarrow$  7; font_name=null_font; font_area=null_font  $\leftarrow$  "";
hyphen_char=null_font  $\leftarrow$  "-"; skew_char=null_font  $\leftarrow$  -1; bchar_label=null_font  $\leftarrow$  non_address;
font_bchar=null_font  $\leftarrow$  non_char; font_false_bchar=null_font  $\leftarrow$  non_char; font_bc=null_font  $\leftarrow$  1;
font_ec=null_font  $\leftarrow$  0; font_size=null_font  $\leftarrow$  0; font_dsize=null_font  $\leftarrow$  0; char_base=null_font  $\leftarrow$  0;
width_base=null_font  $\leftarrow$  0; height_base=null_font  $\leftarrow$  0; depth_base=null_font  $\leftarrow$  0;
italic_base=null_font  $\leftarrow$  0; lig_kern_base=null_font  $\leftarrow$  0; kern_base=null_font  $\leftarrow$  0;
exten_base=null_font  $\leftarrow$  0; font_glue=null_font  $\leftarrow$  null; font_params=null_font  $\leftarrow$  7;
param_base=null_font  $\leftarrow$  -1;
for k  $\leftarrow$  0 to 6 do font_info[k].sc  $\leftarrow$  0;
```

579. $\langle \text{Put each of TeX's primitives into the hash table } 244 \rangle +\equiv$

```
primitive("nullfont", set_font, null_font); text(frozen_null_font)  $\leftarrow$  "nullfont";
eqtb[frozen_null_font]  $\leftarrow$  eqtb[cur_val];
```

580. Of course we want to define macros that suppress the detail of how font information is actually packed, so that we don't have to write things like

$$\text{font_info}[\text{width_base}[f] + \text{font_info}[\text{char_base}[f] + c].\text{qqqq}.b0].sc$$

too often. The WEB definitions here make $\text{char_info}(f)(c)$ the *four-quarters* word of font information corresponding to character c of font f . If q is such a word, $\text{char_width}(f)(q)$ will be the character's width; hence the long formula above is at least abbreviated to

$$\text{char_width}(f)(\text{char_info}(f)(c)).$$

Usually, of course, we will fetch q first and look at several of its fields at the same time.

The italic correction of a character will be denoted by $\text{char_italic}(f)(q)$, so it is analogous to char_width . But we will get at the height and depth in a slightly different way, since we usually want to compute both height and depth if we want either one. The value of $\text{height_depth}(q)$ will be the 8-bit quantity

$$b = \text{height_index} \times 16 + \text{depth_index},$$

and if b is such a byte we will write $\text{char_height}(f)(b)$ and $\text{char_depth}(f)(b)$ for the height and depth of the character c for which $q = \text{char_info}(f)(c)$. Got that?

The tag field will be called $\text{char_tag}(q)$; the remainder byte will be called $\text{rem_byte}(q)$, using a macro that we have already defined above.

Access to a character's *width*, *height*, *depth*, and *tag* fields is part of TeX's inner loop, so we want these macros to produce code that is as fast as possible under the circumstances.

```
define char_info_end(#) ≡ # ].qqqq
define char_info(#) ≡ font_info [ char_base[#] + char_info_end
define char_width_end(#) ≡ #.b0 ].sc
define char_width(#) ≡ font_info [ width_base[#] + char_width_end
define char_exists(#) ≡ (#.b0 > min_quarterword)
define char_italic_end(#) ≡ (qo(#.b2)) div 4 ].sc
define char_italic(#) ≡ font_info [ italic_base[#] + char_italic_end
define height_depth(#) ≡ qo(#.b1)
define char_height_end(#) ≡ (#) div 16 ].sc
define char_height(#) ≡ font_info [ height_base[#] + char_height_end
define char_depth_end(#) ≡ (#) mod 16 ].sc
define char_depth(#) ≡ font_info [ depth_base[#] + char_depth_end
define char_tag(#) ≡ ((qo(#.b2)) mod 4)
```

581. The global variable *null_character* is set up to be a word of *char_info* for a character that doesn't exist. Such a word provides a convenient way to deal with erroneous situations.

<Global variables 13> +≡
null_character: four-quarters; { nonexistent character information }

582. *<Set initial values of key variables 21>* +≡
null_character.b0 ← min_quarterword; null_character.b1 ← min_quarterword;
null_character.b2 ← min_quarterword; null_character.b3 ← min_quarterword;

583. Here are some macros that help process ligatures and kerns. We write $\text{char_kern}(f)(j)$ to find the amount of kerning specified by kerning command j in font f . If j is the char_info for a character with a ligature/kern program, the first instruction of that program is either $i = \text{font.info}[\text{lig_kern_start}(f)(j)]$ or $\text{font.info}[\text{lig_kern_restart}(f)(i)]$, depending on whether or not $\text{skip_byte}(i) \leq \text{stop_flag}$.

The constant kern_base_offset should be simplified, for Pascal compilers that do not do local optimization.

```
define char_kern_end(#) ≡ 256 * op_byte(#) + rem_byte(#) ].sc
define char_kern(#) ≡ font.info [ kern_base[#] + char_kern_end
define kern_base_offset ≡ 256 * (128 + min_quarterword)
define lig_kern_start(#) ≡ lig_kern_base[#] + rem_byte { beginning of lig/kern program }
define lig_kern_restart_end(#) ≡ 256 * op_byte(#) + rem_byte(#) + 32768 - kern_base_offset
define lig_kern_restart(#) ≡ lig_kern_base[#] + lig_kern_restart_end
```

584. Font parameters are referred to as $\text{slant}(f)$, $\text{space}(f)$, etc.

```
define param_end(#) ≡ param_base[#] ].sc
define param(#) ≡ font.info [ # + param_end
define slant ≡ param(slant_code) { slant to the right, per unit distance upward }
define space ≡ param(space_code) { normal space between words }
define space_stretch ≡ param(space_stretch_code) { stretch between words }
define space_shrink ≡ param(space_shrink_code) { shrink between words }
define x_height ≡ param(x_height_code) { one ex }
define quad ≡ param(quad_code) { one em }
define extra_space ≡ param(extra_space_code) { additional space at end of sentence }
```

$\langle \text{The em width for } \text{cur_font 584} \rangle \equiv$
 $\quad \quad \quad \text{quad}(\text{cur_font})$

This code is used in section 481.

585. $\langle \text{The x-height for } \text{cur_font 585} \rangle \equiv$
 $\quad \quad \quad \text{x_height}(\text{cur_font})$

This code is used in section 481.

586. TeX checks the information of a TFM file for validity as the file is being read in, so that no further checks will be needed when typesetting is going on. The somewhat tedious subroutine that does this is called *read_font_info*. It has four parameters: the user font identifier *u*, the file name and area strings *nom* and *aire*, and the “at” size *s*. If *s* is negative, it’s the negative of a scale factor to be applied to the design size; *s* = −1000 is the normal case. Otherwise *s* will be substituted for the design size; in this case, *s* must be positive and less than 2048 pt (i.e., it must be less than 2²⁷ when considered as an integer).

The subroutine opens and closes a global file variable called *tfm_file*. It returns the value of the internal font number that was just loaded. If an error is detected, an error message is issued and no font information is stored; *null_font* is returned in this case.

```

define bad_tfm = 11 { label for read_font_info }
define abort ≡ goto bad_tfm { do this when the TFM data is wrong }

function read_font_info(u : pointer; nom, aire : str_number; s : scaled) : internal_font_number;
    { input a TFM file }

label done, bad_tfm, not_found;

var k: font_index; { index into font_info }

file_opened: boolean; { was tfm_file successfully opened? }
lf, lh, bc, ec, nw, nh, nd, ni, nl, nk, ne, np: halfword; { sizes of subfiles }
f: internal_font_number; { the new font’s number }
g: internal_font_number; { the number to return }
a, b, c, d: eight_bits; { byte variables }
qw: four_quarters; sw: scaled; { accumulators }
bch_label: integer; { left boundary start location, or infinity }
bchar: 0 .. 256; { boundary character, or 256 }
z: scaled; { the design size or the “at” size }
alpha: integer; beta: 1 .. 16; { auxiliary quantities used in fixed-point multiplication }

begin g ← null_font;
    { Read and check the font data; abort if the TFM file is malformed; if there’s no room for this font, say so
        and goto done; otherwise incr(font_ptr) and goto done 588 };
bad_tfm: { Report that the font won’t be loaded 587 };
done: if file_opened then b_close(tfm_file);
    read_font_info ← g;
end;

```

587. There are programs called `TFtoPL` and `PLtoTF` that convert between the TFM format and a symbolic property-list format that can be easily edited. These programs contain extensive diagnostic information, so `TEX` does not have to bother giving precise details about why it rejects a particular TFM file.

```
define start_font_error_message ≡ print_err("Font "); sprint_cs(u); print_char("=");
  print_file_name(nom, aire, "");
  if s ≥ 0 then
    begin print(" at "); print_scaled(s); print("pt");
    end
  else if s ≠ -1000 then
    begin print(" scaled"); print_int(-s);
    end
```

⟨ Report that the font won't be loaded 587 ⟩ ≡

```
start_font_error_message;
if file_opened then print("not loadable: Bad metric (TFM) file")
else print("not loadable: Metric (TFM) file not found");
help5("I wasn't able to read the size data for this font,"
("so I will ignore the font specification.")
(["Wizards can fix TFM files using TFtoPL/PLtoTF."])
("You might try inserting a different font spec;")
("e.g., type `I\font<same font id>=<substitute font name>'.");
error
```

This code is used in section 586.

588. ⟨ Read and check the font data; `abort` if the TFM file is malformed; if there's no room for this font, say so and `goto done`; otherwise `incr(font_ptr)` and `goto done` 588 ⟩ ≡

```
⟨ Open tfm_file for input 589 ⟩;
⟨ Read the TFM size fields 591 ⟩;
⟨ Use size fields to allocate font information 592 ⟩;
⟨ Read the TFM header 594 ⟩;
⟨ Read character data 595 ⟩;
⟨ Read box dimensions 598 ⟩;
⟨ Read ligature/kern program 600 ⟩;
⟨ Read extensible character recipes 601 ⟩;
⟨ Read font parameters 602 ⟩;
⟨ Make final adjustments and goto done 603 ⟩
```

This code is used in section 586.

589. ⟨ Open `tfm_file` for input 589 ⟩ ≡

```
file_opened ← false;
if aire = "" then pack_file_name(nom, TEX_font_area, ".tfm")
else pack_file_name(nom, aire, ".tfm");
if ¬b_open_in(tfm_file) then abort;
file_opened ← true
```

This code is used in section 588.

590. Note: A malformed TFM file might be shorter than it claims to be; thus `eof(tfm_file)` might be true when `read_font_info` refers to `tfm_file↑` or when it says `get(tfm_file)`. If such circumstances cause system error messages, you will have to defeat them somehow, for example by defining `fget` to be ‘`begin get(tfm_file); if eof(tfm_file) then abort; end`’.

```
define fget ≡ get(tfm_file)
define fbyte ≡ tfm_file↑
define read_sixteen(#+) ≡
  begin # ← fbyte;
  if # > 127 then abort;
  fget; # ← # * 400 + fbyte;
  end
define store_four_quarters(#+) ≡
  begin fget; a ← fbyte; qw.b0 ← qi(a); fget; b ← fbyte; qw.b1 ← qi(b); fget; c ← fbyte;
  qw.b2 ← qi(c); fget; d ← fbyte; qw.b3 ← qi(d); # ← qw;
  end
```

591. ⟨ Read the TFM size fields 591 ⟩ ≡

```
begin read_sixteen(lf); fget; read_sixteen(lh); fget; read_sixteen(bc); fget; read_sixteen(ec);
if (bc > ec + 1) ∨ (ec > 255) then abort;
if bc > 255 then { bc = 256 and ec = 255 }
begin bc ← 1; ec ← 0;
end;
fget; read_sixteen(nw); fget; read_sixteen(nh); fget; read_sixteen(nd); fget; read_sixteen(ni); fget;
read_sixteen(nl); fget; read_sixteen(nk); fget; read_sixteen(ne); fget; read_sixteen(np);
if lf ≠ 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np then abort;
if (nw = 0) ∨ (nh = 0) ∨ (nd = 0) ∨ (ni = 0) then abort;
end
```

This code is used in section 588.

592. The preliminary settings of the index-offset variables `char_base`, `width_base`, `lig_kern_base`, `kern_base`, and `exten_base` will be corrected later by subtracting `min_quarterword` from them; and we will subtract 1 from `param_base` too. It’s best to forget about such anomalies until later.

⟨ Use size fields to allocate font information 592 ⟩ ≡

```
lf ← lf - lh; { lf words should be loaded into font_info }
if np < 7 then lf ← lf + 7 - np; { at least seven parameters will appear }
if (font_ptr = font_max) ∨ (fmem_ptr + lf > font_mem_size) then
  ⟨ Apologize for not loading the font, goto done 593 ⟩;
  f ← font_ptr + 1; char_base[f] ← fmem_ptr - bc; width_base[f] ← char_base[f] + ec + 1;
  height_base[f] ← width_base[f] + nw; depth_base[f] ← height_base[f] + nh;
  italic_base[f] ← depth_base[f] + nd; lig_kern_base[f] ← italic_base[f] + ni;
  kern_base[f] ← lig_kern_base[f] + nl - kern_base_offset;
  exten_base[f] ← kern_base[f] + kern_base_offset + nk; param_base[f] ← exten_base[f] + ne
```

This code is used in section 588.

593. ⟨ Apologize for not loading the font, goto done 593 ⟩ ≡

```
begin start_font_error_message; print("not loaded: Not enough room left");
help4("I'm afraid I won't be able to make use of this font,");
("because my memory for character-size data is too small.")
("If you're really stuck, ask a wizard to enlarge me.")
("Or maybe try `\\font<same font id>=<name of loaded font>'"); error; goto done;
end
```

This code is used in section 592.

594. Only the first two words of the header are needed by T_EX82.

```

⟨ Read the TFM header 594 ⟩ ≡
begin if lh < 2 then abort;
store_four_quarters(font_check[f]); fget; read_sixteen(z); { this rejects a negative design size }
fget; z ← z * '400 + fbyte; fget; z ← (z * '20) + (fbyte div '20);
if z < unity then abort;
while lh > 2 do
  begin fget; fget; fget; decr(lh); { ignore the rest of the header }
  end;
font_dsize[f] ← z;
if s ≠ -1000 then
  if s ≥ 0 then z ← s
  else z ← xn_over_d(z, -s, 1000);
font_size[f] ← z;
end

```

This code is used in section 588.

595. ⟨ Read character data 595 ⟩ ≡

```

for k ← fmem_ptr to width_base[f] - 1 do
  begin store_four_quarters(font_info[k].qqq);
  if (a ≥ nw) ∨ (b div '20 ≥ nh) ∨ (b mod '20 ≥ nd) ∨ (c div 4 ≥ ni) then abort;
  case c mod 4 of
    lig_tag: if d ≥ nl then abort;
    ext_tag: if d ≥ ne then abort;
    list_tag: ⟨ Check for charlist cycle 596 ⟩;
    othercases do_nothing { no_tag }
  endcases;
end

```

This code is used in section 588.

596. We want to make sure that there is no cycle of characters linked together by *list_tag* entries, since such a cycle would get T_EX into an endless loop. If such a cycle exists, the routine here detects it when processing the largest character code in the cycle.

```

define check_byte_range(#) ≡
  begin if (# < bc) ∨ (# > ec) then abort
  end
define current_character_being_worked_on ≡ k + bc - fmem_ptr
⟨ Check for charlist cycle 596 ⟩ ≡
begin check_byte_range(d);
while d < current_character_being_worked_on do
  begin qw ← char_info(f)(d); { N.B.: not qi(d), since char_base[f] hasn't been adjusted yet }
  if char_tag(qw) ≠ list_tag then goto not_found;
  d ← qo(rem_byte(qw)); { next character on the list }
  end;
  if d = current_character_being_worked_on then abort; { yes, there's a cycle }
not_found: end

```

This code is used in section 595.

597. A *fix_word* whose four bytes are (a, b, c, d) from left to right represents the number

$$x = \begin{cases} b \cdot 2^{-4} + c \cdot 2^{-12} + d \cdot 2^{-20}, & \text{if } a = 0; \\ -16 + b \cdot 2^{-4} + c \cdot 2^{-12} + d \cdot 2^{-20}, & \text{if } a = 255. \end{cases}$$

(No other choices of a are allowed, since the magnitude of a number in design-size units must be less than 16.) We want to multiply this quantity by the integer z , which is known to be less than 2^{27} . If $z < 2^{23}$, the individual multiplications $b \cdot z$, $c \cdot z$, $d \cdot z$ cannot overflow; otherwise we will divide z by 2, 4, 8, or 16, to obtain a multiplier less than 2^{23} , and we can compensate for this later. If z has thereby been replaced by $z' = z/2^e$, let $\beta = 2^{4-e}$; we shall compute

$$\lfloor (b + c \cdot 2^{-8} + d \cdot 2^{-16}) z' / \beta \rfloor$$

if $a = 0$, or the same quantity minus $\alpha = 2^{4+e} z'$ if $a = 255$. This calculation must be done exactly, in order to guarantee portability of TeX between computers.

```
define store_scaled (#) ≡
  begin fget; a ← fbyte; fget; b ← fbyte; fget; c ← fbyte; fget; d ← fbyte;
  sw ← (((((d * z) div '400) + (c * z)) div '400) + (b * z)) div beta;
  if a = 0 then # ← sw else if a = 255 then # ← sw - alpha else abort;
  end

function store_scaled_f (sq, z : scaled): scaled;
  var a, b, c, d: eight_bits; sw: scaled; alpha: integer; beta: 1 .. 16;
  begin alpha ← 16;
  if z ≥ '1000000000 then pdf_error("font", "size is too large");
  while z ≥ '40000000 do
    begin z ← z div 2; alpha ← alpha + alpha;
    end;
  beta ← 256 div alpha; alpha ← alpha * z;
  if sq ≥ 0 then
    begin d ← sq mod 256; sq ← sq div 256;
      { any "mod 256" not really needed, would typecast alone be safe? }
      c ← sq mod 256; sq ← sq div 256; b ← sq mod 256; sq ← sq div 256; a ← sq mod 256;
    end
  else begin sq ← (sq + 1073741824) + 1073741824; { braces for optimizing compiler }
    d ← sq mod 256; sq ← sq div 256; c ← sq mod 256; sq ← sq div 256; b ← sq mod 256;
    sq ← sq div 256; a ← (sq + 128) mod 256;
  end;
  sw ← (((((d * z) div '400) + (c * z)) div '400) + (b * z)) div beta;
  if a = 0 then store_scaled_f ← sw else if a = 255 then
    store_scaled_f ← sw - alpha else pdf_error("store_scaled_f", "vf scaling");
  end;
```

598. ⟨ Read box dimensions 598 ⟩ ≡

```
begin ⟨ Replace z by z' and compute α, β 599 ⟩;
for k ← width_base[f] to lig_kern_base[f] - 1 do store_scaled(font_info[k].sc);
if font_info[width_base[f]].sc ≠ 0 then abort; { width[0] must be zero }
if font_info[height_base[f]].sc ≠ 0 then abort; { height[0] must be zero }
if font_info[depth_base[f]].sc ≠ 0 then abort; { depth[0] must be zero }
if font_info[italic_base[f]].sc ≠ 0 then abort; { italic[0] must be zero }
end
```

This code is used in section 588.

599. \langle Replace z by z' and compute α, β 599 $\rangle \equiv$

```

begin alpha ← 16;
if  $z \geq 1000000000$  then pdf_error("font", "size_is_too_large");
while  $z \geq 40000000$  do
  begin  $z \leftarrow z \text{ div } 2$ ;  $alpha \leftarrow alpha + alpha$ ;
  end;
 $beta \leftarrow 256 \text{ div } alpha$ ;  $alpha \leftarrow alpha * z$ ;
end

```

This code is used in section 598.

600. define $check_existence(\#) \equiv$

```

begin check_byte_range(#);  $qw \leftarrow char\_info(f)(\#)$ ; { N.B.: not  $qi(\#)$  }
if  $\neg char\_exists(qw)$  then abort;
end

⟨ Read ligature/kern program 600 ⟩ ≡
 $bch\_label \leftarrow 77777$ ;  $bchar \leftarrow 256$ ;
if  $nl > 0$  then
  begin for  $k \leftarrow lig\_kern\_base[f]$  to  $kern\_base[f] + kern\_base\_offset - 1$  do
    begin store_four_quarters(font_info[k].qqqq);
    if  $a > 128$  then
      begin if  $256 * c + d \geq nl$  then abort;
      if  $a = 255$  then
        if  $k = lig\_kern\_base[f]$  then  $bchar \leftarrow b$ ;
        end
      else begin if  $b \neq bchar$  then  $check\_existence(b)$ ;
        if  $c < 128$  then  $check\_existence(d)$  { check ligature }
        else if  $256 * (c - 128) + d \geq nk$  then abort; { check kern }
        if  $a < 128$  then
          if  $k - lig\_kern\_base[f] + a + 1 \geq nl$  then abort;
          end;
      end;
      if  $a = 255$  then  $bch\_label \leftarrow 256 * c + d$ ;
    end;
    for  $k \leftarrow kern\_base[f] + kern\_base\_offset$  to  $exten\_base[f] - 1$  do  $store\_scaled(font\_info[k].sc)$ ;

```

This code is used in section 588.

601. \langle Read extensible character recipes 601 $\rangle \equiv$

```

for  $k \leftarrow exten\_base[f]$  to  $param\_base[f] - 1$  do
  begin store_four_quarters(font_info[k].qqqq);
  if  $a \neq 0$  then  $check\_existence(a)$ ;
  if  $b \neq 0$  then  $check\_existence(b)$ ;
  if  $c \neq 0$  then  $check\_existence(c)$ ;
   $check\_existence(d)$ ;
end

```

This code is used in section 588.

602. We check to see that the TFM file doesn't end prematurely; but no error message is given for files having more than *lf* words.

```
(Read font parameters 602) ≡
begin for k ← 1 to np do
  if k = 1 then {the slant parameter is a pure number}
    begin fget; sw ← fbyte;
    if sw > 127 then sw ← sw - 256;
    fget; sw ← sw * '400 + fbyte; fget; sw ← sw * '400 + fbyte; fget;
    font_info[param_base[f]].sc ← (sw * '20) + (fbyte div '20);
    end
  else store_scaled(font_info[param_base[f] + k - 1].sc);
  if eof(tfm_file) then abort;
  for k ← np + 1 to 7 do font_info[param_base[f] + k - 1].sc ← 0;
  end
```

This code is used in section 588.

603. Now to wrap it up, we have checked all the necessary things about the TFM file, and all we need to do is put the finishing touches on the data for the new font.

```
define adjust(#) ≡ #[f] ← qo(#[f]) {correct for the excess min_quarterword that was added}
(Make final adjustments and goto done 603) ≡
  if np ≥ 7 then font_params[f] ← np else font_params[f] ← 7;
  hyphen_char[f] ← default_hyphen_char; skew_char[f] ← default_skew_char;
  if bch_label < nl then bchar_label[f] ← bch_label + lig_kern_base[f]
  else bchar_label[f] ← non_address;
  font_bchar[f] ← qi(bchar); font_false_bchar[f] ← qi(bchar);
  if bchar ≤ ec then
    if bchar ≥ bc then
      begin qw ← char_info(f)(bchar); {N.B.: not qi(bchar)}
        if char_exists(qw) then font_false_bchar[f] ← non_char;
      end;
    font_name[f] ← nom; font_area[f] ← aire; font_bc[f] ← bc; font_ec[f] ← ec; font_glue[f] ← null;
    adjust(char_base); adjust(width_base); adjust(lig_kern_base); adjust(kern_base); adjust(exten_base);
    decr(param_base[f]); fmem_ptr ← fmem_ptr + lf; font_ptr ← f; g ← f; goto done
```

This code is used in section 588.

604. Before we forget about the format of these tables, let's deal with two of T_EX's basic scanning routines related to font information.

```

⟨ Declare procedures that scan font-related stuff 604 ⟩ ≡
function test_no_ligatures(f : internal_font_number): integer;
  label exit;
  var c: integer;
  begin test_no_ligatures ← 1;
  for c ← font_bc[f] to font_ec[f] do
    if char_exists(orig_char_info(f)(c)) then
      if odd(char_tag(orig_char_info(f)(c))) then
        begin test_no_ligatures ← 0; return;
        end;
  exit: end;
function get_tag_code(f : internal_font_number; c : eight_bits): integer;
  var i: small_number;
  begin if is_valid_char(c) then
    begin i ← char_tag(char_info(f)(c));
    if i = lig_tag then get_tag_code ← 1
    else if i = list_tag then get_tag_code ← 2
    else if i = ext_tag then get_tag_code ← 4
    else get_tag_code ← 0;
    end
  else get_tag_code ← -1;
  end;
procedure scan_font_ident;
  var f: internal_font_number; m: halfword;
  begin ⟨ Get the next non-blank non-call token 432 ⟩;
  if (cur_cmd = def_font) ∨ (cur_cmd = letterspace_font) ∨ (cur_cmd = pdf_copy_font) then f ← cur_font
  else if cur_cmd = set_font then f ← cur_chr
  else if cur_cmd = def_family then
    begin m ← cur_chr; scan_four_bit_int; f ← equiv(m + cur_val);
    end
  else begin print_err("Missing font identifier");
    help2("I was looking for a control sequence whose"
      ("current meaning has been defined by \font."); back_error; f ← null_font;
    end;
  cur_val ← f;
  end;

```

See also section 605.

This code is used in section 435.

605. The following routine is used to implement '`\fontdimen n f'`. The boolean parameter *writing* is set *true* if the calling program intends to change the parameter value.

```
< Declare procedures that scan font-related stuff 604 > +≡
procedure find_font_dimen(writing : boolean); { sets cur_val to font_info location }
  var f: internal_font_number; n: integer; { the parameter number }
  begin scan_int; n ← cur_val; scan_font_ident; f ← cur_val;
  if n ≤ 0 then cur_val ← fmem_ptr
  else begin if writing ∧ (n ≤ space_shrink_code) ∧ (n ≥ space_code) ∧ (font_glue[f] ≠ null) then
    begin delete_glue_ref(font_glue[f]); font_glue[f] ← null;
    end;
    if n > font_params[f] then
      if f < font_ptr then cur_val ← fmem_ptr
      else < Increase the number of parameters in the last font 607 >
    else cur_val ← n + param_base[f];
    end;
  < Issue an error message if cur_val = fmem_ptr 606 >;
end;
```

606. < Issue an error message if *cur_val* = *fmem_ptr* 606 > ≡

```
if cur_val = fmem_ptr then
begin print_err("Font "); print_esc(font_id_text(f)); print(" has only ");
print_int(font_params[f]); print(" fontdimen parameters");
help2("To increase the number of font parameters, you must "
("use \fontdimen immediately after the \font is loaded."); error;
end
```

This code is used in section 605.

607. < Increase the number of parameters in the last font 607 > ≡

```
begin repeat if fmem_ptr = font_mem_size then overflow("font memory", font_mem_size);
  font_info[fmem_ptr].sc ← 0; incr(fmem_ptr); incr(font_params[f]);
until n = font_params[f];
cur_val ← fmem_ptr - 1; { this equals param_base[f] + font_params[f]}
end
```

This code is used in section 605.

608. 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;
```

609. Here is a function that returns a pointer to a character node for a given character in a given font. If that character doesn't exist, *null* is returned instead.

```
function new_character(f : internal_font_number; c : eight_bits): pointer;
label exit;
var p: pointer; { newly allocated node }
begin if font_bc[f] ≤ c then
  if font_ec[f] ≥ c then
    if char_exists(char_info(f)(qi(c))) then
      begin p ← get_avail; font(p) ← f; character(p) ← qi(c); new_character ← p; return;
    end;
  char_warning(f, c); new_character ← null;
exit: end;
```

610. Device-independent file format. The most important output produced by a run of TeX is the “device independent” (DVI) file that specifies where characters and rules are to appear on printed pages. The form of these files was designed by David R. Fuchs in 1979. Almost any reasonable typesetting device can be driven by a program that takes DVI files as input, and dozens of such DVI-to-whatever programs have been written. Thus, it is possible to print the output of TeX on many different kinds of equipment, using TeX as a device-independent “front end.”

A DVI file is a stream of 8-bit bytes, which may be regarded as a series of commands in a machine-like language. The first byte of each command is the operation code, and this code is followed by zero or more bytes that provide parameters to the command. The parameters themselves may consist of several consecutive bytes; for example, the ‘*set_rule*’ command has two parameters, each of which is four bytes long. Parameters are usually regarded as nonnegative integers; but four-byte-long parameters, and shorter parameters that denote distances, can be either positive or negative. Such parameters are given in two’s complement notation. For example, a two-byte-long distance parameter has a value between -2^{15} and $2^{15} - 1$. As in TFM files, numbers that occupy more than one byte position appear in BigEndian order.

A DVI file consists of a “preamble,” followed by a sequence of one or more “pages,” followed by a “postamble.” The preamble is simply a *pre* command, with its parameters that define the dimensions used in the file; this must come first. Each “page” consists of a *bop* command, followed by any number of other commands that tell where characters are to be placed on a physical page, followed by an *eop* command. The pages appear in the order that TeX generated them. If we ignore *nop* commands and *fnt_def* commands (which are allowed between any two commands in the file), each *eop* command is immediately followed by a *bop* command, or by a *post* command; in the latter case, there are no more pages in the file, and the remaining bytes form the postamble. Further details about the postamble will be explained later.

Some parameters in DVI commands are “pointers.” These are four-byte quantities that give the location number of some other byte in the file; the first byte is number 0, then comes number 1, and so on. For example, one of the parameters of a *bop* command points to the previous *bop*; this makes it feasible to read the pages in backwards order, in case the results are being directed to a device that stacks its output face up. Suppose the preamble of a DVI file occupies bytes 0 to 99. Now if the first page occupies bytes 100 to 999, say, and if the second page occupies bytes 1000 to 1999, then the *bop* that starts in byte 1000 points to 100 and the *bop* that starts in byte 2000 points to 1000. (The very first *bop*, i.e., the one starting in byte 100, has a pointer of -1.)

611. The DVI format is intended to be both compact and easily interpreted by a machine. Compactness is achieved by making most of the information implicit instead of explicit. When a DVI-reading program reads the commands for a page, it keeps track of several quantities: (a) The current font *f* is an integer; this value is changed only by *fnt* and *fnt_num* commands. (b) The current position on the page is given by two numbers called the horizontal and vertical coordinates, *h* and *v*. Both coordinates are zero at the upper left corner of the page; moving to the right corresponds to increasing the horizontal coordinate, and moving down corresponds to increasing the vertical coordinate. Thus, the coordinates are essentially Cartesian, except that vertical directions are flipped; the Cartesian version of (h, v) would be $(h, -v)$. (c) The current spacing amounts are given by four numbers *w*, *x*, *y*, and *z*, where *w* and *x* are used for horizontal spacing and where *y* and *z* are used for vertical spacing. (d) There is a stack containing (h, v, w, x, y, z) values; the DVI commands *push* and *pop* are used to change the current level of operation. Note that the current font *f* is not pushed and popped; the stack contains only information about positioning.

The values of *h*, *v*, *w*, *x*, *y*, and *z* are signed integers having up to 32 bits, including the sign. Since they represent physical distances, there is a small unit of measurement such that increasing *h* by 1 means moving a certain tiny distance to the right. The actual unit of measurement is variable, as explained below; TeX sets things up so that its DVI output is in sp units, i.e., scaled points, in agreement with all the *scaled* dimensions in TeX’s data structures.

612. Here is a list of all the commands that may appear in a DVI file. Each command is specified by its symbolic name (e.g., *bop*), its opcode byte (e.g., 139), and its parameters (if any). The parameters are followed by a bracketed number telling how many bytes they occupy; for example, '*p*[4]' means that parameter *p* is four bytes long.

set_char_0 0. Typeset character number 0 from font *f* such that the reference point of the character is at (h, v) . Then increase *h* by the width of that character. Note that a character may have zero or negative width, so one cannot be sure that *h* will advance after this command; but *h* usually does increase.

set_char_1 through *set_char_127* (opcodes 1 to 127). Do the operations of *set_char_0*; but use the character whose number matches the opcode, instead of character 0.

set1 128 *c*[1]. Same as *set_char_0*, except that character number *c* is typeset. T_EX82 uses this command for characters in the range $128 \leq c < 256$.

set2 129 *c*[2]. Same as *set1*, except that *c* is two bytes long, so it is in the range $0 \leq c < 65536$. T_EX82 never uses this command, but it should come in handy for extensions of T_EX that deal with oriental languages.

set3 130 *c*[3]. Same as *set1*, except that *c* is three bytes long, so it can be as large as $2^{24} - 1$. Not even the Chinese language has this many characters, but this command might prove useful in some yet unforeseen extension.

set4 131 *c*[4]. Same as *set1*, except that *c* is four bytes long. Imagine that.

set_rule 132 *a*[4] *b*[4]. Typeset a solid black rectangle of height *a* and width *b*, with its bottom left corner at (h, v) . Then set $h \leftarrow h + b$. If either $a \leq 0$ or $b \leq 0$, nothing should be typeset. Note that if $b < 0$, the value of *h* will decrease even though nothing else happens. See below for details about how to typeset rules so that consistency with METAFONT is guaranteed.

put1 133 *c*[1]. Typeset character number *c* from font *f* such that the reference point of the character is at (h, v) . (The ‘put’ commands are exactly like the ‘set’ commands, except that they simply put out a character or a rule without moving the reference point afterwards.)

put2 134 *c*[2]. Same as *set2*, except that *h* is not changed.

put3 135 *c*[3]. Same as *set3*, except that *h* is not changed.

put4 136 *c*[4]. Same as *set4*, except that *h* is not changed.

put_rule 137 *a*[4] *b*[4]. Same as *set_rule*, except that *h* is not changed.

nop 138. No operation, do nothing. Any number of *nop*’s may occur between DVI commands, but a *nop* cannot be inserted between a command and its parameters or between two parameters.

bop 139 *c*₀[4] *c*₁[4] ... *c*₉[4] *p*[4]. Beginning of a page: Set $(h, v, w, x, y, z) \leftarrow (0, 0, 0, 0, 0, 0)$ and set the stack empty. Set the current font *f* to an undefined value. The ten *c_i* parameters hold the values of \count0 ... \count9 in T_EX at the time \shipout was invoked for this page; they can be used to identify pages, if a user wants to print only part of a DVI file. The parameter *p* points to the previous *bop* in the file; the first *bop* has *p* = -1.

eop 140. End of page: Print what you have read since the previous *bop*. At this point the stack should be empty. (The DVI-reading programs that drive most output devices will have kept a buffer of the material that appears on the page that has just ended. This material is largely, but not entirely, in order by *v* coordinate and (for fixed *v*) by *h* coordinate; so it usually needs to be sorted into some order that is appropriate for the device in question.)

push 141. Push the current values of (h, v, w, x, y, z) onto the top of the stack; do not change any of these values. Note that *f* is not pushed.

pop 142. Pop the top six values off of the stack and assign them respectively to (h, v, w, x, y, z) . The number of pops should never exceed the number of pushes, since it would be highly embarrassing if the stack were empty at the time of a *pop* command.

right1 143 *b*[1]. Set $h \leftarrow h + b$, i.e., move right *b* units. The parameter is a signed number in two’s complement notation, $-128 \leq b < 128$; if *b* < 0, the reference point moves left.

right2 144 *b*[2]. Same as *right1*, except that *b* is a two-byte quantity in the range $-32768 \leq b < 32768$.

right3 145 *b*[3]. Same as *right1*, except that *b* is a three-byte quantity in the range $-2^{23} \leq b < 2^{23}$.

right4 146 *b*[4]. Same as *right1*, except that *b* is a four-byte quantity in the range $-2^{31} \leq b < 2^{31}$.

w0 147. Set $h \leftarrow h + w$; i.e., move right *w* units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how *w* gets particular values.

w1 148 *b*[1]. Set $w \leftarrow b$ and $h \leftarrow h + b$. The value of *b* is a signed quantity in two's complement notation, $-128 \leq b < 128$. This command changes the current *w* spacing and moves right by *b*.

w2 149 *b*[2]. Same as *w1*, but *b* is two bytes long, $-32768 \leq b < 32768$.

w3 150 *b*[3]. Same as *w1*, but *b* is three bytes long, $-2^{23} \leq b < 2^{23}$.

w4 151 *b*[4]. Same as *w1*, but *b* is four bytes long, $-2^{31} \leq b < 2^{31}$.

x0 152. Set $h \leftarrow h + x$; i.e., move right *x* units. The '*x*' commands are like the '*w*' commands except that they involve *x* instead of *w*.

x1 153 *b*[1]. Set $x \leftarrow b$ and $h \leftarrow h + b$. The value of *b* is a signed quantity in two's complement notation, $-128 \leq b < 128$. This command changes the current *x* spacing and moves right by *b*.

x2 154 *b*[2]. Same as *x1*, but *b* is two bytes long, $-32768 \leq b < 32768$.

x3 155 *b*[3]. Same as *x1*, but *b* is three bytes long, $-2^{23} \leq b < 2^{23}$.

x4 156 *b*[4]. Same as *x1*, but *b* is four bytes long, $-2^{31} \leq b < 2^{31}$.

down1 157 *a*[1]. Set $v \leftarrow v + a$, i.e., move down *a* units. The parameter is a signed number in two's complement notation, $-128 \leq a < 128$; if *a* < 0, the reference point moves up.

down2 158 *a*[2]. Same as *down1*, except that *a* is a two-byte quantity in the range $-32768 \leq a < 32768$.

down3 159 *a*[3]. Same as *down1*, except that *a* is a three-byte quantity in the range $-2^{23} \leq a < 2^{23}$.

down4 160 *a*[4]. Same as *down1*, except that *a* is a four-byte quantity in the range $-2^{31} \leq a < 2^{31}$.

y0 161. Set $v \leftarrow v + y$; i.e., move down *y* units. With luck, this parameterless command will usually suffice, because the same kind of motion will occur several times in succession; the following commands explain how *y* gets particular values.

y1 162 *a*[1]. Set $y \leftarrow a$ and $v \leftarrow v + a$. The value of *a* is a signed quantity in two's complement notation, $-128 \leq a < 128$. This command changes the current *y* spacing and moves down by *a*.

y2 163 *a*[2]. Same as *y1*, but *a* is two bytes long, $-32768 \leq a < 32768$.

y3 164 *a*[3]. Same as *y1*, but *a* is three bytes long, $-2^{23} \leq a < 2^{23}$.

y4 165 *a*[4]. Same as *y1*, but *a* is four bytes long, $-2^{31} \leq a < 2^{31}$.

z0 166. Set $v \leftarrow v + z$; i.e., move down *z* units. The '*z*' commands are like the '*y*' commands except that they involve *z* instead of *y*.

z1 167 *a*[1]. Set $z \leftarrow a$ and $v \leftarrow v + a$. The value of *a* is a signed quantity in two's complement notation, $-128 \leq a < 128$. This command changes the current *z* spacing and moves down by *a*.

z2 168 *a*[2]. Same as *z1*, but *a* is two bytes long, $-32768 \leq a < 32768$.

z3 169 *a*[3]. Same as *z1*, but *a* is three bytes long, $-2^{23} \leq a < 2^{23}$.

z4 170 *a*[4]. Same as *z1*, but *a* is four bytes long, $-2^{31} \leq a < 2^{31}$.

fnt-num_0 171. Set $f \leftarrow 0$. Font 0 must previously have been defined by a *fnt-def* instruction, as explained below.

fnt-num_1 through *fnt-num_63* (opcodes 172 to 234). Set $f \leftarrow 1, \dots, f \leftarrow 63$, respectively.

fnt1 235 *k*[1]. Set $f \leftarrow k$. TeX82 uses this command for font numbers in the range $64 \leq k < 256$.

fnt2 236 *k*[2]. Same as *fnt1*, except that *k* is two bytes long, so it is in the range $0 \leq k < 65536$. TeX82 never generates this command, but large font numbers may prove useful for specifications of color or texture, or they may be used for special fonts that have fixed numbers in some external coding scheme.

fnt3 237 $k[3]$. Same as *fnt1*, except that k is three bytes long, so it can be as large as $2^{24} - 1$.

fnt4 238 $k[4]$. Same as *fnt1*, except that k is four bytes long; this is for the really big font numbers (and for the negative ones).

xxx1 239 $k[1] x[k]$. This command is undefined in general; it functions as a $(k + 2)$ -byte *nop* unless special DVI-reading programs are being used. TeX82 generates *xxx1* when a short enough \special appears, setting k to the number of bytes being sent. It is recommended that x be a string having the form of a keyword followed by possible parameters relevant to that keyword.

xxx2 240 $k[2] x[k]$. Like *xxx1*, but $0 \leq k < 65536$.

xxx3 241 $k[3] x[k]$. Like *xxx1*, but $0 \leq k < 2^{24}$.

xxx4 242 $k[4] x[k]$. Like *xxx1*, but k can be ridiculously large. TeX82 uses *xxx4* when sending a string of length 256 or more.

fnt_def1 243 $k[1] c[4] s[4] d[4] a[1] l[1] n[a + l]$. Define font k , where $0 \leq k < 256$; font definitions will be explained shortly.

fnt_def2 244 $k[2] c[4] s[4] d[4] a[1] l[1] n[a + l]$. Define font k , where $0 \leq k < 65536$.

fnt_def3 245 $k[3] c[4] s[4] d[4] a[1] l[1] n[a + l]$. Define font k , where $0 \leq k < 2^{24}$.

fnt_def4 246 $k[4] c[4] s[4] d[4] a[1] l[1] n[a + l]$. Define font k , where $-2^{31} \leq k < 2^{31}$.

pre 247 $i[1] num[4] den[4] mag[4] k[1] x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameters i , num , den , mag , k , and x are explained below.

post 248. Beginning of the postamble, see below.

post_post 249. Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

```
613. define set_char_0 = 0 { typeset character 0 and move right }
define set1 = 128 { typeset a character and move right }
define set_rule = 132 { typeset a rule and move right }
define put_rule = 137 { typeset a rule }
define nop = 138 { no operation }
define bop = 139 { beginning of page }
define eop = 140 { ending of page }
define push = 141 { save the current positions }
define pop = 142 { restore previous positions }
define right1 = 143 { move right }
define w0 = 147 { move right by w }
define w1 = 148 { move right and set w }
define x0 = 152 { move right by x }
define x1 = 153 { move right and set x }
define down1 = 157 { move down }
define y0 = 161 { move down by y }
define y1 = 162 { move down and set y }
define z0 = 166 { move down by z }
define z1 = 167 { move down and set z }
define fnt_num_0 = 171 { set current font to 0 }
define fnt1 = 235 { set current font }
define xxx1 = 239 { extension to DVI primitives }
define xxx4 = 242 { potentially long extension to DVI primitives }
define fnt_def1 = 243 { define the meaning of a font number }
define pre = 247 { preamble }
define post = 248 { postamble beginning }
define post_post = 249 { postamble ending }
```

614. The preamble contains basic information about the file as a whole. As stated above, there are six parameters:

i[1] *num*[4] *den*[4] *mag*[4] *k*[1] *x*[*k*]

The *i* byte identifies DVI format; currently this byte is always set to 2. (The value *i* = 3 is currently used for an extended format that allows a mixture of right-to-left and left-to-right typesetting. Some day we will set *i* = 4, when DVI format makes another incompatible change—perhaps in the year 2048.)

The next two parameters, *num* and *den*, are positive integers that define the units of measurement; they are the numerator and denominator of a fraction by which all dimensions in the DVI file could be multiplied in order to get lengths in units of 10^{-7} meters. Since $7227\text{pt} = 254\text{cm}$, and since TeX works with scaled points where there are 2^{16} sp in a point, TeX sets $\text{num}/\text{den} = (254 \cdot 10^5)/(7227 \cdot 2^{16}) = 25400000/473628672$.

The *mag* parameter is what TeX calls `\mag`, i.e., 1000 times the desired magnification. The actual fraction by which dimensions are multiplied is therefore $\text{mag} \cdot \text{num}/1000\text{den}$. Note that if a TeX source document does not call for any ‘true’ dimensions, and if you change it only by specifying a different `\mag` setting, the DVI file that TeX creates will be completely unchanged except for the value of *mag* in the preamble and postamble. (Fancy DVI-reading programs allow users to override the *mag* setting when a DVI file is being printed.)

Finally, *k* and *x* allow the DVI writer to include a comment, which is not interpreted further. The length of comment *x* is *k*, where $0 \leq k < 256$.

```
define id_byte = 2 { identifies the kind of DVI files described here }
```

615. Font definitions for a given font number *k* contain further parameters

c[4] *s*[4] *d*[4] *a*[1] *l*[1] *n*[*a + l*].

The four-byte value *c* is the check sum that TeX found in the TFM file for this font; *c* should match the check sum of the font found by programs that read this DVI file.

Parameter *s* contains a fixed-point scale factor that is applied to the character widths in font *k*; font dimensions in TFM files and other font files are relative to this quantity, which is called the “at size” elsewhere in this documentation. The value of *s* is always positive and less than 2^{27} . It is given in the same units as the other DVI dimensions, i.e., in sp when TeX82 has made the file. Parameter *d* is similar to *s*; it is the “design size,” and (like *s*) it is given in DVI units. Thus, font *k* is to be used at $\text{mag} \cdot \text{s}/1000\text{d}$ times its normal size.

The remaining part of a font definition gives the external name of the font, which is an ASCII string of length *a + l*. The number *a* is the length of the “area” or directory, and *l* is the length of the font name itself; the standard local system font area is supposed to be used when *a* = 0. The *n* field contains the area in its first *a* bytes.

Font definitions must appear before the first use of a particular font number. Once font *k* is defined, it must not be defined again; however, we shall see below that font definitions appear in the postamble as well as in the pages, so in this sense each font number is defined exactly twice, if at all. Like *nop* commands, font definitions can appear before the first *bop*, or between an *eop* and a *bop*.

616. Sometimes it is desirable to make horizontal or vertical rules line up precisely with certain features in characters of a font. It is possible to guarantee the correct matching between DVI output and the characters generated by METAFONT by adhering to the following principles: (1) The METAFONT characters should be positioned so that a bottom edge or left edge that is supposed to line up with the bottom or left edge of a rule appears at the reference point, i.e., in row 0 and column 0 of the METAFONT raster. This ensures that the position of the rule will not be rounded differently when the pixel size is not a perfect multiple of the units of measurement in the DVI file. (2) A typeset rule of height $a > 0$ and width $b > 0$ should be equivalent to a METAFONT-generated character having black pixels in precisely those raster positions whose METAFONT coordinates satisfy $0 \leq x < \alpha b$ and $0 \leq y < \alpha a$, where α is the number of pixels per DVI unit.

617. The last page in a DVI file is followed by '*post*'; this command introduces the postamble, which summarizes important facts that TeX has accumulated about the file, making it possible to print subsets of the data with reasonable efficiency. The postamble has the form

```
post p[4] num[4] den[4] mag[4] l[4] u[4] s[2] t[2]
  ⟨ font definitions ⟩
post_post q[4] i[1] 223's[≥4]
```

Here *p* is a pointer to the final *bop* in the file. The next three parameters, *num*, *den*, and *mag*, are duplicates of the quantities that appeared in the preamble.

Parameters *l* and *u* give respectively the height-plus-depth of the tallest page and the width of the widest page, in the same units as other dimensions of the file. These numbers might be used by a DVI-reading program to position individual "pages" on large sheets of film or paper; however, the standard convention for output on normal size paper is to position each page so that the upper left-hand corner is exactly one inch from the left and the top. Experience has shown that it is unwise to design DVI-to-printer software that attempts cleverly to center the output; a fixed position of the upper left corner is easiest for users to understand and to work with. Therefore *l* and *u* are often ignored.

Parameter *s* is the maximum stack depth (i.e., the largest excess of *push* commands over *pop* commands) needed to process this file. Then comes *t*, the total number of pages (*bop* commands) present.

The postamble continues with font definitions, which are any number of *fnt_def* commands as described above, possibly interspersed with *nop* commands. Each font number that is used in the DVI file must be defined exactly twice: Once before it is first selected by a *fnt* command, and once in the postamble.

618. The last part of the postamble, following the *post_post* byte that signifies the end of the font definitions, contains *q*, a pointer to the *post* command that started the postamble. An identification byte, *i*, comes next; this currently equals 2, as in the preamble.

The *i* byte is followed by four or more bytes that are all equal to the decimal number 223 (i.e., '337 in octal). TeX puts out four to seven of these trailing bytes, until the total length of the file is a multiple of four bytes, since this works out best on machines that pack four bytes per word; but any number of 223's is allowed, as long as there are at least four of them. In effect, 223 is a sort of signature that is added at the very end.

This curious way to finish off a DVI file makes it feasible for DVI-reading programs to find the postamble first, on most computers, even though TeX wants to write the postamble last. Most operating systems permit random access to individual words or bytes of a file, so the DVI reader can start at the end and skip backwards over the 223's until finding the identification byte. Then it can back up four bytes, read *q*, and move to byte *q* of the file. This byte should, of course, contain the value 248 (*post*); now the postamble can be read, so the DVI reader can discover all the information needed for typesetting the pages. Note that it is also possible to skip through the DVI file at reasonably high speed to locate a particular page, if that proves desirable. This saves a lot of time, since DVI files used in production jobs tend to be large.

Unfortunately, however, standard Pascal does not include the ability to access a random position in a file, or even to determine the length of a file. Almost all systems nowadays provide the necessary capabilities, so DVI format has been designed to work most efficiently with modern operating systems. But if DVI files have to be processed under the restrictions of standard Pascal, one can simply read them from front to back, since the necessary header information is present in the preamble and in the font definitions. (The *l* and *u* and *s* and *t* parameters, which appear only in the postamble, are "frills" that are handy but not absolutely necessary.)

619. Shipping pages out. After considering TeX's eyes and stomach, we come now to the bowels.

The *ship_out* procedure is given a pointer to a box; its mission is to describe that box in DVI form, outputting a "page" to *dvi_file*. The DVI coordinates $(h, v) = (0, 0)$ should correspond to the upper left corner of the box being shipped.

Since boxes can be inside of boxes inside of boxes, the main work of *ship_out* is done by two mutually recursive routines, *hlist_out* and *vlist_out*, which traverse the hlists and vlists inside of horizontal and vertical boxes.

As individual pages are being processed, we need to accumulate information about the entire set of pages, since such statistics must be reported in the postamble. The global variables *total_pages*, *max_v*, *max_h*, *max_push*, and *last_bop* are used to record this information.

The variable *doing_leaders* is *true* while leaders are being output. The variable *dead_cycles* contains the number of times an output routine has been initiated since the last *ship_out*.

A few additional global variables are also defined here for use in *vlist_out* and *hlist_out*. They could have been local variables, but that would waste stack space when boxes are deeply nested, since the values of these variables are not needed during recursive calls.

```
<Global variables 13> +≡
total_pages: integer; { the number of pages that have been shipped out }
max_v: scaled; { maximum height-plus-depth of pages shipped so far }
max_h: scaled; { maximum width of pages shipped so far }
max_push: integer; { deepest nesting of push commands encountered so far }
last_bop: integer; { location of previous bop in the DVI output }
dead_cycles: integer; { recent outputs that didn't ship anything out }
doing_leaders: boolean; { are we inside a leader box? }
c, f: quarterword; { character and font in current char_node }
rule_ht, rule_dp, rule_wd: scaled; { size of current rule being output }
g: pointer; { current glue specification }
lq, lr: integer; { quantities used in calculations for leaders }
```

620. < Set initial values of key variables 21 > +≡

```
total_pages ← 0; max_v ← 0; max_h ← 0; max_push ← 0; last_bop ← -1; doing_leaders ← false;
dead_cycles ← 0; cur_s ← -1;
```

621. The DVI bytes are output to a buffer instead of being written directly to the output file. This makes it possible to reduce the overhead of subroutine calls, thereby measurably speeding up the computation, since output of DVI bytes is part of TeX's inner loop. And it has another advantage as well, since we can change instructions in the buffer in order to make the output more compact. For example, a '*down2*' command can be changed to a '*y2*', thereby making a subsequent '*y0*' command possible, saving two bytes.

The output buffer is divided into two parts of equal size; the bytes found in *dvi_buf*[0 .. *half_buf* - 1] constitute the first half, and those in *dvi_buf*[*half_buf* .. *dvi_buf_size* - 1] constitute the second. The global variable *dvi_ptr* points to the position that will receive the next output byte. When *dvi_ptr* reaches *dvi_limit*, which is always equal to one of the two values *half_buf* or *dvi_buf_size*, the half buffer that is about to be invaded next is sent to the output and *dvi_limit* is changed to its other value. Thus, there is always at least a half buffer's worth of information present, except at the very beginning of the job.

Bytes of the DVI file are numbered sequentially starting with 0; the next byte to be generated will be number *dvi_offset* + *dvi_ptr*. A byte is present in the buffer only if its number is $\geq dvi_gone$.

< Types in the outer block 18 > +≡

```
dvi_index = 0 .. dvi_buf_size; { an index into the output buffer }
```

622. Some systems may find it more efficient to make *dvi_buf* a packed array, since output of four bytes at once may be facilitated.

```
(Global variables 13) +≡
dvi_buf: array [dvi_index] of eight_bits; { buffer for DVI output }
half_buf: dvi_index; { half of dvi_buf_size }
dvi_limit: dvi_index; { end of the current half buffer }
dvi_ptr: dvi_index; { the next available buffer address }
dvi_offset: integer; { dvi_buf_size times the number of times the output buffer has been fully emptied }
dvi_gone: integer; { the number of bytes already output to dvi_file }
```

623. Initially the buffer is all in one piece; we will output half of it only after it first fills up.

(Set initial values of key variables 21) +≡

```
half_buf ← dvi_buf_size div 2; dvi_limit ← dvi_buf_size; dvi_ptr ← 0; dvi_offset ← 0; dvi_gone ← 0;
```

624. The actual output of *dvi_buf*[*a* .. *b*] to *dvi_file* is performed by calling *write_dvi(a, b)*. For best results, this procedure should be optimized to run as fast as possible on each particular system, since it is part of TeX's inner loop. It is safe to assume that *a* and *b* + 1 will both be multiples of 4 when *write_dvi(a, b)* is called; therefore it is possible on many machines to use efficient methods to pack four bytes per word and to output an array of words with one system call.

```
procedure write_dvi(a, b : dvi_index);
  var k: dvi_index;
  begin for k ← a to b do write(dvi_file, dvi_buf[k]);
  end;
```

625. To put a byte in the buffer without paying the cost of invoking a procedure each time, we use the macro *dvi_out*.

```
define dvi_out(#) ≡ begin dvi_buf[dvi_ptr] ← #; incr(dvi_ptr);
  if dvi_ptr = dvi_limit then dvi_swap;
  end

procedure dvi_swap; { outputs half of the buffer }
begin if dvi_limit = dvi_buf_size then
  begin write_dvi(0, half_buf - 1); dvi_limit ← half_buf; dvi_offset ← dvi_offset + dvi_buf_size;
  dvi_ptr ← 0;
  end
else begin write_dvi(half_buf, dvi_buf_size - 1); dvi_limit ← dvi_buf_size;
  end;
dvi_gone ← dvi_gone + half_buf;
end;
```

626. Here is how we clean out the buffer when TeX is all through; *dvi_ptr* will be a multiple of 4.

(Empty the last bytes out of *dvi_buf* 626) ≡

```
if dvi_limit = half_buf then write_dvi(half_buf, dvi_buf_size - 1);
if dvi_ptr > 0 then write_dvi(0, dvi_ptr - 1)
```

This code is used in section 670.

627. The *dvi-four* procedure outputs four bytes in two's complement notation, without risking arithmetic overflow.

```
procedure dvi-four(x : integer);
begin if x ≥ 0 then dvi-out(x div '100000000)
else begin x ← x + '1000000000; x ← x + '1000000000; dvi-out((x div '100000000) + 128);
end;
x ← x mod '100000000; dvi-out(x div '200000); x ← x mod '200000; dvi-out(x div '400);
dvi-out(x mod '400);
end;
```

628. A mild optimization of the output is performed by the *dvi-pop* routine, which issues a *pop* unless it is possible to cancel a '*push pop*' pair. The parameter to *dvi-pop* is the byte address following the old *push* that matches the new *pop*.

```
procedure dvi-pop(l : integer);
begin if (l = dvi_offset + dvi_ptr) ∧ (dvi_ptr > 0) then decr(dvi_ptr)
else dvi-out(pop);
end;
```

629. Here's a procedure that outputs a font definition. Since TeX82 uses at most 256 different fonts per job, *fnt_def1* is always used as the command code.

```
procedure dvi-font-def(f : internal-font-number);
var k: pool_pointer; { index into str_pool }
begin dvi_out(fnt_def1); dvi_out(f - font_base - 1);
dvi_out(qo(font_check[f].b0)); dvi_out(qo(font_check[f].b1)); dvi_out(qo(font_check[f].b2));
dvi_out(qo(font_check[f].b3));
dvi-four(font_size[f]); dvi-four(font_dsize[f]);
dvi_out(length(font_area[f])); dvi_out(length(font_name[f]));
⟨ Output the font name whose internal number is f 630 ⟩;
end;
```

630. ⟨ Output the font name whose internal number is f 630 ⟩ ≡
for k ← str_start[font_area[f]] to str_start[font_area[f] + 1] – 1 do dvi_out(so(str_pool[k]));
for k ← str_start[font_name[f]] to str_start[font_name[f] + 1] – 1 do dvi_out(so(str_pool[k]))

This code is used in section 629.

631. Versions of TeX intended for small computers might well choose to omit the ideas in the next few parts of this program, since it is not really necessary to optimize the DVI code by making use of the $w0$, $x0$, $y0$, and $z0$ commands. Furthermore, the algorithm that we are about to describe does not pretend to give an optimum reduction in the length of the DVI code; after all, speed is more important than compactness. But the method is surprisingly effective, and it takes comparatively little time.

We can best understand the basic idea by first considering a simpler problem that has the same essential characteristics. Given a sequence of digits, say 3141592653589, we want to assign subscripts d , y , or z to each digit so as to maximize the number of “ y -hits” and “ z -hits”; a y -hit is an instance of two appearances of the same digit with the subscript y , where no y 's intervene between the two appearances, and a z -hit is defined similarly. For example, the sequence above could be decorated with subscripts as follows:

$$3_z \ 1_y \ 4_d \ 1_y \ 5_y \ 9_d \ 2_d \ 6_d \ 5_y \ 3_z \ 5_y \ 8_d \ 9_d.$$

There are three y -hits ($1_y \dots 1_y$ and $5_y \dots 5_y$) and one z -hit ($3_z \dots 3_z$); there are no d -hits, since the two appearances of 9_d have d 's between them, but we don't count d -hits so it doesn't matter how many there are. These subscripts are analogous to the DVI commands called *down*, *y*, and *z*, and the digits are analogous to different amounts of vertical motion; a y -hit or z -hit corresponds to the opportunity to use the one-byte commands *y0* or *z0* in a DVI file.

TeX's method of assigning subscripts works like this: Append a new digit, say δ , to the right of the sequence. Now look back through the sequence until one of the following things happens: (a) You see δ_y or δ_z , and this was the first time you encountered a y or z subscript, respectively. Then assign y or z to the new δ ; you have scored a hit. (b) You see δ_d , and no y subscripts have been encountered so far during this search. Then change the previous δ_d to δ_y (this corresponds to changing a command in the output buffer), and assign y to the new δ ; it's another hit. (c) You see δ_d , and a y subscript has been seen but not a z . Change the previous δ_d to δ_z and assign z to the new δ . (d) You encounter both y and z subscripts before encountering a suitable δ , or you scan all the way to the front of the sequence. Assign d to the new δ ; this assignment may be changed later.

The subscripts $3_z \ 1_y \ 4_d \dots$ in the example above were, in fact, produced by this procedure, as the reader can verify. (Go ahead and try it.)

632. In order to implement such an idea, TeX maintains a stack of pointers to the *down*, *y*, and *z* commands that have been generated for the current page. And there is a similar stack for *right*, *w*, and *x* commands. These stacks are called the *down* stack and *right* stack, and their top elements are maintained in the variables *down_ptr* and *right_ptr*.

Each entry in these stacks contains four fields: The *width* field is the amount of motion down or to the right; the *location* field is the byte number of the DVI command in question (including the appropriate *dvi_offset*); the *link* field points to the next item below this one on the stack; and the *info* field encodes the options for possible change in the DVI command.

```
define movement_node_size = 3 { number of words per entry in the down and right stacks }
define location(#) ≡ mem[# + 2].int { DVI byte number for a movement command }
⟨ Global variables 13 ⟩ +≡
down_ptr, right_ptr: pointer; { heads of the down and right stacks }
```

633. ⟨ Set initial values of key variables 21 ⟩ +≡
 $down_ptr \leftarrow null; right_ptr \leftarrow null;$

634. Here is a subroutine that produces a DVI command for some specified downward or rightward motion. It has two parameters: w is the amount of motion, and o is either *down1* or *right1*. We use the fact that the command codes have convenient arithmetic properties: $y1 - \text{down1} = w1 - \text{right1}$ and $z1 - \text{down1} = x1 - \text{right1}$.

```

procedure movement( $w$  : scaled;  $o$  : eight_bits);
label exit, found, not_found, 2, 1;
var mstate: small_number; { have we seen a  $y$  or  $z$ ? }
 $p, q$ : pointer; { current and top nodes on the stack }
 $k$ : integer; { index into dvi_buf, modulo dvi_buf_size }
begin  $q \leftarrow \text{get.node}(\text{movement.node\_size})$ ; { new node for the top of the stack }
width( $q$ )  $\leftarrow w$ ; location( $q$ )  $\leftarrow \text{dvi\_offset} + \text{dvi\_ptr}$ ;
if  $o = \text{down1}$  then
  begin link( $q$ )  $\leftarrow \text{down\_ptr}$ ; down_ptr  $\leftarrow q$ ;
  end
else begin link( $q$ )  $\leftarrow \text{right\_ptr}$ ; right_ptr  $\leftarrow q$ ;
end;
{ Look at the other stack entries until deciding what sort of DVI command to generate; goto found if
node  $p$  is a "hit" 638 };
{ Generate a down or right command for  $w$  and return 637 };
found: { Generate a  $y0$  or  $z0$  command in order to reuse a previous appearance of  $w$  636 };
exit: end;

```

635. The *info* fields in the entries of the down stack or the right stack have six possible settings: *y_here* or *z_here* mean that the DVI command refers to y or z , respectively (or to w or x , in the case of horizontal motion); *yz_OK* means that the DVI command is *down* (or *right*) but can be changed to either y or z (or to either w or x); *y_OK* means that it is *down* and can be changed to y but not z ; *z_OK* is similar; and *d_fixed* means it must stay *down*.

The four settings *yz_OK*, *y_OK*, *z_OK*, *d_fixed* would not need to be distinguished from each other if we were simply solving the digit-subscripting problem mentioned above. But in TeX's case there is a complication because of the nested structure of *push* and *pop* commands. Suppose we add parentheses to the digit-subscripting problem, redefining hits so that $\delta_y \dots \delta_y$ is a hit if all y 's between the δ 's are enclosed in properly nested parentheses, and if the parenthesis level of the right-hand δ_y is deeper than or equal to that of the left-hand one. Thus, '(' and ')' correspond to '*push*' and '*pop*'. Now if we want to assign a subscript to the final 1 in the sequence

$$2_y 7_d 1_d (8_z 2_y 8_z)1$$

we cannot change the previous 1_d to 1_y , since that would invalidate the $2_y \dots 2_y$ hit. But we can change it to 1_z , scoring a hit since the intervening 8_z 's are enclosed in parentheses.

The program below removes movement nodes that are introduced after a *push*, before it outputs the corresponding *pop*.

```

define y_here = 1 { info when the movement entry points to a  $y$  command }
define z_here = 2 { info when the movement entry points to a  $z$  command }
define yz_OK = 3 { info corresponding to an unconstrained down command }
define y_OK = 4 { info corresponding to a down that can't become a  $z$  }
define z_OK = 5 { info corresponding to a down that can't become a  $y$  }
define d_fixed = 6 { info corresponding to a down that can't change }

```

636. When the *movement* procedure gets to the label *found*, the value of *info*(*p*) will be either *y_here* or *z_here*. If it is, say, *y_here*, the procedure generates a *y0* command (or a *w0* command), and marks all *info* fields between *q* and *p* so that *y* is not OK in that range.

```
<Generate a y0 or z0 command in order to reuse a previous appearance of w 636 > ≡
  info(q) ← info(p);
  if info(q) = y_here then
    begin dvi_out(o + y0 − down1); { y0 or w0 }
    while link(q) ≠ p do
      begin q ← link(q);
      case info(q) of
        yz_OK: info(q) ← z_OK;
        y_OK: info(q) ← d_fixed;
        othercases do_nothing
      endcases;
      end;
    end
  else begin dvi_out(o + z0 − down1); { z0 or x0 }
    while link(q) ≠ p do
      begin q ← link(q);
      case info(q) of
        yz_OK: info(q) ← y_OK;
        z_OK: info(q) ← d_fixed;
        othercases do_nothing
      endcases;
      end;
    end
  end
```

This code is used in section 634.

637. <Generate a *down* or *right* command for *w* and return 637 > ≡

```
  info(q) ← yz_OK;
  if abs(w) ≥ '40000000 then
    begin dvi_out(o + 3); { down4 or right4 }
    dvi_four(w); return;
    end;
  if abs(w) ≥ '100000 then
    begin dvi_out(o + 2); { down3 or right3 }
    if w < 0 then w ← w + '100000000;
    dvi_out(w div '200000); w ← w mod '200000; goto 2;
    end;
  if abs(w) ≥ '200 then
    begin dvi_out(o + 1); { down2 or right2 }
    if w < 0 then w ← w + '200000;
    goto 2;
    end;
  dvi_out(o); { down1 or right1 }
  if w < 0 then w ← w + '400;
  goto 1;
2: dvi_out(w div '400);
1: dvi_out(w mod '400); return
```

This code is used in section 634.

638. As we search through the stack, we are in one of three states, *y_seen*, *z_seen*, or *none_seen*, depending on whether we have encountered *y_here* or *z_here* nodes. These states are encoded as multiples of 6, so that they can be added to the *info* fields for quick decision-making.

```

define none_seen = 0 { no y_here or z_here nodes have been encountered yet }
define y_seen = 6 { we have seen y_here but not z_here }
define z_seen = 12 { we have seen z_here but not y_here }

⟨ Look at the other stack entries until deciding what sort of DVI command to generate; goto found if node
  p is a “hit” 638 ⟩ ≡
  p ← link(q); mstate ← none_seen;
  while p ≠ null do
    begin if width(p) = w then ⟨ Consider a node with matching width; goto found if it’s a hit 639 ⟩
    else case mstate + info(p) of
      none_seen + y_here: mstate ← y_seen;
      none_seen + z_here: mstate ← z_seen;
      y_seen + z_here, z_seen + y_here: goto not_found;
      othercases do_nothing
      endcases;
      p ← link(p);
    end;
  not_found:

```

This code is used in section 634.

639. We might find a valid hit in a *y* or *z* byte that is already gone from the buffer. But we can’t change bytes that are gone forever; “the moving finger writes, . . . ”

```

⟨ Consider a node with matching width; goto found if it’s a hit 639 ⟩ ≡
  case mstate + info(p) of
    none_seen + yz_OK, none_seen + y_OK, z_seen + yz_OK, z_seen + y_OK:
      if location(p) < dvi_gone then goto not_found
      else ⟨ Change buffered instruction to y or w and goto found 640 ⟩;
    none_seen + z_OK, y_seen + yz_OK, y_seen + z_OK:
      if location(p) < dvi_gone then goto not_found
      else ⟨ Change buffered instruction to z or x and goto found 641 ⟩;
    none_seen + y_here, none_seen + z_here, y_seen + z_here, z_seen + y_here: goto found;
    othercases do_nothing
    endcases

```

This code is used in section 638.

640. ⟨ Change buffered instruction to *y* or *w* and **goto found** 640 ⟩ ≡

```

begin k ← location(p) – dvi_offset;
if k < 0 then k ← k + dvi_buf_size;
  dvi_buf[k] ← dvi_buf[k] + y1 – down1; info(p) ← y_here; goto found;
end

```

This code is used in section 639.

641. ⟨ Change buffered instruction to *z* or *x* and **goto found** 641 ⟩ ≡

```

begin k ← location(p) – dvi_offset;
if k < 0 then k ← k + dvi_buf_size;
  dvi_buf[k] ← dvi_buf[k] + z1 – down1; info(p) ← z_here; goto found;
end

```

This code is used in section 639.

642. In case you are wondering when all the movement nodes are removed from TeX's memory, the answer is that they are recycled just before *hlist_out* and *vlist_out* finish outputting a box. This restores the down and right stacks to the state they were in before the box was output, except that some *info*'s may have become more restrictive.

```
procedure prune_movements(l : integer); { delete movement nodes with location  $\geq l$  }
  label done, exit;
  var p: pointer; { node being deleted }
  begin while down_ptr  $\neq$  null do
    begin if location(down_ptr)  $<$  l then goto done;
      p  $\leftarrow$  down_ptr; down_ptr  $\leftarrow$  link(p); free_node(p, movement_node_size);
    end;
  done: while right_ptr  $\neq$  null do
    begin if location(right_ptr)  $<$  l then return;
      p  $\leftarrow$  right_ptr; right_ptr  $\leftarrow$  link(p); free_node(p, movement_node_size);
    end;
  exit: end;
```

643. 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--~~X~~T) 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 *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(#) $\equiv$  (qo(subtype(#))) { direction mode of a box }
define set_box_lr(#) $\equiv$  subtype(#) $\leftarrow$  set_box_lr_end
define set_box_lr_end(#) $\equiv$  qi(#)
define left_to_right = 0
define right_to_left = 1
define reflected  $\equiv$  1 - cur_dir { the opposite of cur_dir }
define synch_h  $\equiv$ 
  if cur_h  $\neq$  dvi_h then
    begin movement(cur_h - dvi_h, right1); dvi_h  $\leftarrow$  cur_h;
  end
define synch_v  $\equiv$ 
  if cur_v  $\neq$  dvi_v then
    begin movement(cur_v - dvi_v, down1); dvi_v  $\leftarrow$  cur_v;
  end
{ Global variables 13 } + $\equiv$ 
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 }
```

644. *< Calculate DVI page dimensions and margins 644 >* ≡
 $\text{cur_h_offset} \leftarrow h_offset; \text{cur_v_offset} \leftarrow v_offset;$
 $\text{if } pdf_page_width \neq 0 \text{ then } cur_page_width \leftarrow pdf_page_width$
 $\text{else } cur_page_width \leftarrow width(p) + 2 * cur_h_offset + 2 * 4736286;$
 $\quad \{ 4736286 = 1\text{in}, \text{ the funny DVI origin offset} \}$
 $\text{if } pdf_page_height \neq 0 \text{ then } cur_page_height \leftarrow pdf_page_height$
 $\text{else } cur_page_height \leftarrow height(p) + depth(p) + 2 * cur_v_offset + 2 * 4736286$
 $\quad \{ 4736286 = 1\text{in}, \text{ the funny DVI origin offset} \}$

This code is used in section 645.

645. *< Initialize variables as *ship_out* begins 645 >* ≡
 $dvi_h \leftarrow 0; dvi_v \leftarrow 0; cur_h \leftarrow h_offset; dvi_f \leftarrow null_font;$
< Calculate DVI page dimensions and margins 644 >;
 $\text{ensure_dvi_open};$
 $\text{if } total_pages = 0 \text{ then}$
 $\quad \text{begin } dvi_out(pre); dvi_out(id_byte); \quad \{ \text{output the preamble} \}$
 $\quad dvi_four(25400000); dvi_four(473628672); \quad \{ \text{conversion ratio for sp} \}$
 $\quad prepare_mag; dvi_four(mag); \quad \{ \text{magnification factor is frozen} \}$
 $\quad old_setting \leftarrow selector; selector \leftarrow new_string; print(" \text{\TeX} \text{_output} "); print_int(year);$
 $\quad print_char("."); print_two(month); print_char("."); print_two(day); print_char(":");$
 $\quad print_two(time \text{ div } 60); print_two(time \text{ mod } 60); selector \leftarrow old_setting; dvi_out(cur_length);$
 $\quad \text{for } s \leftarrow str_start[str_ptr] \text{ to } pool_ptr - 1 \text{ do } dvi_out(so(str_pool[s]));$
 $\quad pool_ptr \leftarrow str_start[str_ptr]; \quad \{ \text{flush the current string} \}$
 $\quad \text{end}$

This code is used in section 668.

646. When *hlist_out* is called, its duty is to output the box represented by the *hlist_node* pointed to by *temp_ptr*. The reference point of that box has coordinates (*cur_h*, *cur_v*).

Similarly, when *vlist_out* is called, its duty is to output the box represented by the *vlist_node* pointed to by *temp_ptr*. The reference point of that box has coordinates (*cur_h*, *cur_v*).

procedure *vlist_out*; *forward*; { *hlist_out* and *vlist_out* are mutually recursive }

647. 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 1615 }
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 1714 };
left_edge ← cur_h;
while p ≠ null do { Output node p for hlist_out and move to the next node, maintaining the condition
    cur_v = base_line 648 };
{ Finish hlist_out for mixed direction typesetting 1715 };
prune_movements(save_loc);
if cur_s > 0 then dvi_pop(save_loc);
decr(cur_s);
end;

```

648. 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* 648 ⟩ ≡
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 649 ⟩;
    if c ≥ qi(128) then dvi_out(set1);
    dvi_out(go(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 650 ⟩
```

This code is used in section 647.

649. ⟨ Change font *dvi_f* to *f* 649 ⟩ ≡
begin if ¬*font_used[f]* then
 begin *dvi_font_def(f)*; *font_used[f]* ← true;
 end;
if *f* ≤ 64 + *font_base* then *dvi_out(f - font_base - 1 + fnt_num_0)*
else begin *dvi_out(fnt1)*; *dvi_out(f - font_base - 1)*;
 end;
dvi_f ← *f*;
end

This code is used in section 648.

650. ⟨ Output the non-*char_node* *p* for *hlist_out* and move to the next node 650 ⟩ ≡

```

begin case type(p) of
hlist_node, vlist_node: ⟨ Output a box in an hlist 651 ⟩;
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 1614 ⟩;
glue_node: ⟨ Move right or output leaders 653 ⟩;
margin_kern_node, kern_node: cur_h ← cur_h + width(p);
math_node: ⟨ Handle a math node in hlist_out 1716 ⟩;
ligature_node: ⟨ Make node p look like a char_node and goto reswitch 826 ⟩;
    ⟨ Cases of hlist_out that arise in mixed direction text only 1720 ⟩
othercases do_nothing
endcases;
goto next_p;
fin_rule: ⟨ Output a rule in an hlist 652 ⟩;
move_past: cur_h ← cur_h + rule_wd;
next_p: prev_p ← p; p ← link(p);
    end
```

This code is used in section 648.

651. \langle Output a box in an hlist 651 $\rangle \equiv$

```

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 650.

652. \langle Output a rule in an hlist 652 $\rangle \equiv$

```

if is_running(rule_ht) then rule_ht ← height(this_box);
if is_running(rule_dp) then rule_dp ← depth(this_box);
rule_ht ← rule_ht + rule_dp; { this is the rule thickness }
if (rule_ht > 0) ∧ (rule_wd > 0) then { we don't output empty rules }
begin synch_h; cur_v ← base_line + rule_dp; synch_v; dvi_out(set_rule); dvi_four(rule_ht);
dvi_four(rule_wd); cur_v ← base_line; dvi_h ← dvi_h + rule_wd;
end

```

This code is used in section 650.

653. $\text{define billion} \equiv \text{float_constant}(1000000000)$
 $\text{define vet_glue}(\#) \equiv \text{glue_temp} \leftarrow \#;$
 $\quad \text{if glue_temp} > \text{billion} \text{ then glue_temp} \leftarrow \text{billion}$
 $\quad \text{else if glue_temp} < -\text{billion} \text{ then glue_temp} \leftarrow -\text{billion}$
 $\text{define round_glue} \equiv g \leftarrow \text{glue_ptr}(p); \text{rule_wd} \leftarrow \text{width}(g) - \text{cur_g};$
 $\quad \text{if } g\text{-sign} \neq \text{normal} \text{ then}$
 $\quad \quad \text{begin if } g\text{-sign} = \text{stretching} \text{ then}$
 $\quad \quad \quad \text{begin if } \text{stretch_order}(g) = g\text{-order} \text{ then}$
 $\quad \quad \quad \quad \text{begin } \text{cur_glue} \leftarrow \text{cur_glue} + \text{stretch}(g); \text{vet_glue}(\text{float}(\text{glue_set}(\text{this_box})) * \text{cur_glue});$
 $\quad \quad \quad \quad \text{cur_g} \leftarrow \text{round}(\text{glue_temp});$
 $\quad \quad \quad \end;$
 $\quad \quad \end$
 $\quad \quad \text{else if } \text{shrink_order}(g) = g\text{-order} \text{ then}$
 $\quad \quad \quad \text{begin } \text{cur_glue} \leftarrow \text{cur_glue} - \text{shrink}(g); \text{vet_glue}(\text{float}(\text{glue_set}(\text{this_box})) * \text{cur_glue});$
 $\quad \quad \quad \text{cur_g} \leftarrow \text{round}(\text{glue_temp});$
 $\quad \quad \quad \end;$
 $\quad \quad \end$
 $\quad \text{rule_wd} \leftarrow \text{rule_wd} + \text{cur_g}$

\langle Move right or output leaders 653 $\rangle \equiv$

```

begin round_glue;
if eTeX_ex then {Handle a glue node for mixed direction typesetting 1699};
if subtype(p) ≥ a_leaders then
    {Output leaders in an hlist, goto fin_rule if a rule or to next_p if done 654};
    goto move_past;
end

```

This code is used in section 650.

654. ⟨Output leaders in an hlist, **goto** *fin_rule* if a rule or to *next_p* if done 654⟩ ≡

```

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 655 };
  while cur_h + leader_wd ≤ edge do
    { Output a leader box at cur_h, then advance cur_h by leader_wd + lx 656 };
    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 653.

655. The calculations related to leaders require a bit of care. First, in the case of *a_leaders* (aligned leaders), we want to move *cur_h* to *left_edge* plus the smallest multiple of *leader_wd* for which the result is not less than the current value of *cur_h*; i.e., *cur_h* should become *left_edge* + *leader_wd* × ⌈(*cur_h* - *left_edge*) / *leader_wd*⌉. The program here should work in all cases even though some implementations of Pascal give nonstandard results for the div operation when *cur_h* is less than *left_edge*.

In the case of *c_leaders* (centered leaders), we want to increase *cur_h* by half of the excess space not occupied by the leaders; and in the case of *x_leaders* (expanded leaders) we increase *cur_h* by 1/(*q* + 1) of this excess space, where *q* is the number of times the leader box will be replicated. Slight inaccuracies in the division might accumulate; half of this rounding error is placed at each end of the leaders.

⟨ Let *cur_h* be the position of the first box, and set *leader_wd* + *lx* to the spacing between corresponding parts of boxes 655 ⟩ ≡

```

if subtype(p) = a_leaders then
  begin save_h ← cur_h; cur_h ← left_edge + leader_wd * ((cur_h - left_edge) div leader_wd);
  if cur_h < save_h then cur_h ← cur_h + leader_wd;
  end
else begin lq ← rule_wd div leader_wd; { the number of box copies }
  lr ← rule_wd mod leader_wd; { the remaining space }
  if subtype(p) = c_leaders then cur_h ← cur_h + (lr div 2)
  else begin lx ← lr div (lq + 1); cur_h ← cur_h + ((lr - (lq - 1) * lx) div 2);
  end;
end

```

This code is used in sections 654 and 736.

656. 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 656 > ≡
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 654.

657. The *vlist_out* routine is similar to *hlist_out*, but a bit simpler.

```
procedure vlist_out; { output a vlist_node box }
label move_past, fin_rule, next_p;
var left_edge: scaled; { the left coordinate for this box }
top_edge: scaled; { the top 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 vlist }
save_loc: integer; { DVI byte location upon entry }
leader_box: pointer; { the leader box being replicated }
leader_ht: scaled; { height of leader box being replicated }
lx: scaled; { extra space between leader boxes }
outer_doing_leaders: boolean; { were we doing leaders? }
edge: scaled; { bottom boundary of leader space }
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; left_edge ← cur_h; cur_v ← cur_v − height(this_box); top_edge ← cur_v;
while p ≠ null do {Output node p for vlist_out and move to the next node, maintaining the condition
    cur_h = left_edge 658};
prune_movements(save_loc);
if cur_s > 0 then dvi_pop(save_loc);
decr(cur_s);
end;
```

658. {Output node *p* for *vlist_out* and move to the next node, maintaining the condition

```
    cur_h = left_edge 658} ≡
begin if is_char_node(p) then confusion("vlistout")
else {Output the non-char_node p for vlist_out 659};
next_p: p ← link(p);
end
```

This code is used in section 657.

```

659. <Output the non-char_node p for vlist_out 659> ≡
begin case type(p) of
hlist_node, vlist_node: <Output a box in a vlist 660>;
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 a vlist 1613>;
glue_node: <Move down or output leaders 662>;
kern_node: cur_v ← cur_v + width(p);
othercases do_nothing
endcases;
endcases;
goto next_p;
fin_rule: <Output a rule in a vlist, goto next_p 661>;
move_past: cur_v ← cur_v + rule_ht;
end

```

This code is used in section 658.

660. 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 660> ≡
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 659.

```

661. <Output a rule in a vlist, goto next_p 661> ≡
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 659.

662. \langle Move down or output leaders 662 $\rangle \equiv$

```

begin  $g \leftarrow glue\_ptr(p)$ ;  $rule\_ht \leftarrow width(g) - cur\_g$ ;
if  $g\_sign \neq normal$  then
  begin if  $g\_sign = stretching$  then
    begin if  $stretch\_order(g) = g\_order$  then
      begin  $cur\_glue \leftarrow cur\_glue + stretch(g)$ ;  $vet\_glue(float(glue\_set(this\_box)) * cur\_glue)$ ;
          $cur\_g \leftarrow round(glue\_temp)$ ;
         end;
      end
    else if  $shrink\_order(g) = g\_order$  then
      begin  $cur\_glue \leftarrow cur\_glue - shrink(g)$ ;  $vet\_glue(float(glue\_set(this\_box)) * cur\_glue)$ ;
          $cur\_g \leftarrow round(glue\_temp)$ ;
         end;
      end;
    end;
  rule_ht  $\leftarrow rule\_ht + cur\_g$ ;
  if  $subtype(p) \geq a\_leaders$  then
    ⟨ Output leaders in a vlist, goto fin_rule if a rule or to next_p if done 663 ⟩;
    goto move_past;
  end
end;
```

This code is used in section 659.

663. \langle Output leaders in a vlist, goto fin_rule if a rule or to next_p if done 663 $\rangle \equiv$

```

begin  $leader\_box \leftarrow leader\_ptr(p)$ ;
if  $type(leader\_box) = rule\_node$  then
  begin  $rule\_wd \leftarrow width(leader\_box)$ ;  $rule\_dp \leftarrow 0$ ; goto fin_rule;
  end;
 $leader\_ht \leftarrow height(leader\_box) + depth(leader\_box)$ ;
if ( $leader\_ht > 0$ )  $\wedge$  ( $rule\_ht > 0$ ) then
  begin  $rule\_ht \leftarrow rule\_ht + 10$ ; { compensate for floating-point rounding }
  edge  $\leftarrow cur\_v + rule\_ht$ ;  $lx \leftarrow 0$ ; { Let  $cur\_v$  be the position of the first box, and set  $leader\_ht + lx$  to
  the spacing between corresponding parts of boxes 664 };
  while  $cur\_v + leader\_ht \leq edge$  do
    ⟨ Output a leader box at  $cur\_v$ , then advance  $cur\_v$  by  $leader\_ht + lx$  665 ⟩;
     $cur\_v \leftarrow edge - 10$ ; goto next_p;
  end;
end
```

This code is used in section 662.

664. \langle Let cur_v be the position of the first box, and set $leader_ht + lx$ to the spacing between
corresponding parts of boxes 664 $\rangle \equiv$

```

if  $subtype(p) = a\_leaders$  then
  begin  $save\_v \leftarrow cur\_v$ ;  $cur\_v \leftarrow top\_edge + leader\_ht * ((cur\_v - top\_edge) \text{ div } leader\_ht)$ ;
  if  $cur\_v < save\_v$  then  $cur\_v \leftarrow cur\_v + leader\_ht$ ;
  end
else begin  $lq \leftarrow rule\_ht \text{ div } leader\_ht$ ; { the number of box copies }
   $lr \leftarrow rule\_ht \text{ mod } leader\_ht$ ; { the remaining space }
  if  $subtype(p) = c\_leaders$  then  $cur\_v \leftarrow cur\_v + (lr \text{ div } 2)$ 
  else begin  $lx \leftarrow lr \text{ div } (lq + 1)$ ;  $cur\_v \leftarrow cur\_v + ((lr - (lq - 1) * lx) \text{ div } 2)$ ;
  end;
end
```

This code is used in sections 663 and 745.

665. 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 665> ≡
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 663.

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

```
procedure dvi_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 668};
if eTeX_ex then {Check for LR anomalies at the end of ship_out 1730};
if tracing_output ≤ 0 then print_char("]");
dead_cycles ← 0; update_terminal; { progress report }
{ Flush the box from memory, showing statistics if requested 667};
end;
```

667. \langle Flush the box from memory, showing statistics if requested 667 $\rangle \equiv$

```

stat if tracing_stats > 1 then
  begin print_nl("Memory_usage_before:"); print_int(var_used); print_char("&");
  print_int(dyn_used); print_char(";");
  end;
tats
flush_node_list(p);
stat if tracing_stats > 1 then
  begin print("after:"); print_int(var_used); print_char("&"); print_int(dyn_used);
  print(";still_un触ed:");
  print_int(hi_mem_min - lo_mem_max - 1); print_ln;
  end;
tats

```

This code is used in sections 666 and 750.

668. \langle Ship box p out 668 $\rangle \equiv$

```

⟨ Update the values of max_h and max_v; but if the page is too large, goto done 669 ⟩;
⟨ Initialize variables as ship_out begins 645 ⟩;
page_loc ← dvi_offset + dvi_ptr; dvi_out(bop);
for k ← 0 to 9 do dvi_four(count(k));
dvi_four(last_bop); last_bop ← page_loc; cur_v ← height(p) + v_offset; temp_ptr ← p;
if type(p) = vlist_node then vlist_out else hlist_out;
dvi_out(eop); incr(total_pages); cur_s ← -1;
done:

```

This code is used in section 666.

669. Sometimes the user will generate a huge page because other error messages are being ignored. Such pages are not output to the dvi file, since they may confuse the printing software.

\langle Update the values of max_h and max_v; but if the page is too large, goto done 669 $\rangle \equiv$

```

if (height(p) > max_dimen) ∨ (depth(p) > max_dimen) ∨
  (height(p) + depth(p) + v_offset > max_dimen) ∨ (width(p) + h_offset > max_dimen) then
  begin print_err("Huge_page_cannot_be_shipped_out");
  help2("The_page_just_created_is_more_than_18_feet_tall_or")
  ("more_than_18_feet_wide,_so_I_suspect_something_went_wrong."); error;
  if tracing_output ≤ 0 then
    begin begin_diagnostic; print_nl("The_following_box_has_been_deleted:"); show_box(p);
    end_diagnostic(true);
    end;
  goto done;
end;
if height(p) + depth(p) + v_offset > max_v then max_v ← height(p) + depth(p) + v_offset;
if width(p) + h_offset > max_h then max_h ← width(p) + h_offset

```

This code is used in sections 668 and 751.

670. At the end of the program, we must finish things off by writing the postamble. If *total_pages* = 0, the DVI file was never opened. If *total_pages* \geq 65536, the DVI file will lie. And if *max_push* \geq 65536, the user deserves whatever chaos might ensue.

An integer variable *k* will be declared for use by this routine.

```
<Finish the DVI file 670> ≡
  while cur_s > -1 do
    begin if cur_s > 0 then dvi_out(pop)
    else begin dvi_out(eop); incr(total_pages);
    end;
    decr(cur_s);
    end;
  if total_pages = 0 then print_nl("No\u2014pages\u2014of\u2014output .")
  else begin dvi_out(post); {beginning of the postamble}
    dvi_four(last_bop); last_bop ← dvi_offset + dvi_ptr - 5; {post location}
    dvi_four(25400000); dvi_four(473628672); {conversion ratio for sp}
    prepare_mag; dvi_four(mag); {magnification factor}
    dvi_four(max_v); dvi_four(max_h);
    dvi_out(max_push div 256); dvi_out(max_push mod 256);
    dvi_out((total_pages div 256) mod 256); dvi_out(total_pages mod 256);
    {Output the font definitions for all fonts that were used 671};
    dvi_out(post_post); dvi_out(last_bop); dvi_out(id_byte);
    k ← 4 + ((dvi_buf_size - dvi_ptr) mod 4); {the number of 223's}
    while k > 0 do
      begin dvi_out(223); decr(k);
      end;
    {Empty the last bytes out of dvi_buf 626};
    print_nl("Output\u2014written\u2014on\u2014"); slow_print(output_file_name); print(" \u2014("); print_int(total_pages);
    print(" \u2014page");
    if total_pages ≠ 1 then print_char("s");
    print(", \u2014"); print_int(dvi_offset + dvi_ptr); print(" \u2014bytes ."); b_close(dvi_file);
    end
```

This code is used in section 1513.

671. {Output the font definitions for all fonts that were used 671} ≡

```
  while font_ptr > font_base do
    begin if font_used[font_ptr] then dvi_font_def(font_ptr);
    decr(font_ptr);
    end
```

This code is used in section 670.

672. pdfTeX basic. Initialize pdfTeX's parameters to some useful default value. Helpful in case one forgets to set them during INITEX run.

⟨ Initialize table entries (done by INITEX only) 182 ⟩ +≡

```
pdf_h_origin ← (one_hundred_inch + 50) div 100; pdf_v_origin ← (one_hundred_inch + 50) div 100;  
pdf_compress_level ← 9; pdf_objcompresslevel ← 0; pdf_decimal_digits ← 3; pdf_image_resolution ← 72;  
pdf_major_version ← 1; pdf_minor_version ← 4; pdf_gamma ← 1000; pdf_image_gamma ← 2200;  
pdf_image_hicolor ← 1; pdf_image_apply_gamma ← 0; pdf_px_dimen ← one_bp; pdf_draftmode ← 0;
```

673. The subroutines define the corresponding macros so we can use them in C.

```

define flushable(#)  $\equiv$  (# = str_ptr - 1)
define is_valid_char(#)  $\equiv$  ((font_bc[f]  $\leq$  #)  $\wedge$  (#  $\leq$  font_ec[f])  $\wedge$  char_exists(char_info(f)(#)))
function get_pdf_compress_level: integer;
  begin get_pdf_compress_level  $\leftarrow$  pdf_compress_level;
  end;
function get_pdf_suppress_warning_dup_map: integer;
  begin get_pdf_suppress_warning_dup_map  $\leftarrow$  pdf_suppress_warning_dup_map;
  end;
function get_pdf_suppress_warning_page_group: integer;
  begin get_pdf_suppress_warning_page_group  $\leftarrow$  pdf_suppress_warning_page_group;
  end;
function get_pdf_suppress_ptex_info: integer;
  begin get_pdf_suppress_ptex_info  $\leftarrow$  pdf_suppress_ptex_info;
  end;
function get_pdf OMIT_charset: integer;
  begin get_pdf OMIT_charset  $\leftarrow$  pdf OMIT_charset;
  end;
function get_ptex_use_underscore: boolean;
  begin get_ptex_use_underscore  $\leftarrow$  (pdf_ptex_use_underscore > 0)  $\vee$  (pdf_major_version  $\geq$  2)
  end;
function get_nullfont: internal_font_number;
  begin get_nullfont  $\leftarrow$  null_font;
  end;
function get_fontbase: internal_font_number;
  begin get_fontbase  $\leftarrow$  font_base;
  end;
function get_nulucs: pointer;
  begin get_nulucs  $\leftarrow$  null_cs;
  end;
function get_nullptr: pointer;
  begin get_nullptr  $\leftarrow$  null;
  end;
function get_tex_int(code : integer): integer;
  begin get_tex_int  $\leftarrow$  int_par(code);
  end;
function get_tex_dimen(code : integer): scaled;
  begin get_tex_dimen  $\leftarrow$  dimen_par(code);
  end;
function get_x_height(f : internal_font_number): scaled;
  begin get_x_height  $\leftarrow$  x_height(f);
  end;
function get_charwidth(f : internal_font_number; c : eight_bits): scaled;
  begin if is_valid_char(c) then get_charwidth  $\leftarrow$  char_width(f)(char_info(f)(c))
    else get_charwidth  $\leftarrow$  0;
  end;
function get_charheight(f : internal_font_number; c : eight_bits): scaled;
  begin if is_valid_char(c) then get_charheight  $\leftarrow$  char_height(f)(height_depth(char_info(f)(c)))
    else get_charheight  $\leftarrow$  0;
  end;
function get_chardepth(f : internal_font_number; c : eight_bits): scaled;
  begin if is_valid_char(c) then get_chardepth  $\leftarrow$  char_depth(f)(height_depth(char_info(f)(c)))
    else get_chardepth  $\leftarrow$  0;
  end;

```

```

else get_chardepth ← 0;
end;
function get_quad(f : internal_font_number): scaled;
begin get_quad ← quad(f);
end;
function get_slant(f : internal_font_number): scaled;
begin get_slant ← slant(f);
end;

```

674. Helper for debugging purposes:

```

procedure short_display_n(p, m : integer); { prints highlights of list p }
  var n: integer; { for replacement counts }
    i: integer;
  begin i ← 0; font_in_short_display ← null_font;
  if p = null then return;
  while p > mem_min do
    begin if is_char_node(p) then
      begin if p ≤ mem_end then
        begin if font(p) ≠ font_in_short_display then
          begin if (font(p) < font_base) ∨ (font(p) > font_max) then print_char("*")
          else print_font_identifier(font(p));
          print_char(" "); font_in_short_display ← font(p);
          end;
        print_ASCII(qo(character(p)));
        end;
      end
    else begin if (type(p) = glue_node) ∨ (type(p) = disc_node) ∨ (type(p) = penalty_node) ∨ ((type(p) =
      kern_node) ∧ (subtype(p) = explicit)) then incr(i);
    if i ≥ m then return;
    if (type(p) = disc_node) then
      begin print("|"); short_display(pre_break(p)); print("|"); short_display(post_break(p));
      print("|"); n ← replace_count(p);
      while n > 0 do
        begin if link(p) ≠ null then p ← link(p);
        decr(n);
        end;
      end
    else ⟨Print a short indication of the contents of node p 193⟩;
    end;
    p ← link(p);
    if p = null then return;
    end;
    update_terminal;
  end;

```

675. Sometimes it is necessary to allocate memory for PDF output that cannot be deallocated then, so we use *pdf_mem* for this purpose.

```

⟨Constants in the outer block 11⟩ +≡
inf_pdf_mem_size = 10000; { min size of the pdf_mem array }
sup_pdf_mem_size = 10000000; { max size of the pdf_mem array }

```

676. ⟨ Global variables 13 ⟩ +≡

pdf_mem_size: integer;
pdf_mem: ↑integer;
pdf_mem_ptr: integer;

677. ⟨ Set initial values of key variables 21 ⟩ +≡

pdf_mem_ptr ← 1; { the first word is not used so we can use zero as a value for testing whether a pointer
 to *pdf_mem* is valid }
pdf_mem_size ← *inf_pdf_mem_size*; { allocated size of *pdf_mem* array }

678. We use *pdf_get_mem* to allocate memory in *pdf_mem*.

```
function pdf_get_mem(s : integer): integer; { allocate s words in pdf_mem }
  var a: integer;
begin if s > sup_pdf_mem_size - pdf_mem_ptr then
  overflow("PDF_memory_size_(pdf_mem_size)", pdf_mem_size);
if pdf_mem_ptr + s > pdf_mem_size then
  begin a ← 0.2 * pdf_mem_size;
  if pdf_mem_ptr + s > pdf_mem_size + a then pdf_mem_size ← pdf_mem_ptr + s
  else if pdf_mem_size < sup_pdf_mem_size - a then pdf_mem_size ← pdf_mem_size + a
  else pdf_mem_size ← sup_pdf_mem_size;
  pdf_mem ← xrealloc_array(pdf_mem, integer, pdf_mem_size);
  end;
pdf_get_mem ← pdf_mem_ptr; pdf_mem_ptr ← pdf_mem_ptr + s;
end;
```

679. pdfTeX output low-level subroutines. We use the similar subroutines to handle the output buffer for PDF output. When compress is used, the state of writing to buffer is held in *zip_write_state*. We must write the header of PDF output file in initialization to ensure that it will be the first written bytes.

⟨ Constants in the outer block 11 ⟩ +≡

```
pdf_op_buf_size = 16384; { size of the PDF output buffer }
inf_pdf_os_buf_size = 1; { initial value of pdf_os_buf_size }
sup_pdf_os_buf_size = 5000000; { arbitrary upper hard limit of pdf_os_buf_size }
pdf_os_max_objs = 100; { maximum number of objects in object stream }
```

680. The following macros are similar as for DVI buffer handling:

```

define pdf_offset ≡ (pdf_gone + pdf_ptr)
    { the file offset of last byte in PDF buffer that pdf_ptr points to }
define no_zip ≡ 0 { no ZIP compression }
define zip_writing ≡ 1 { ZIP compression being used }
define zip_finish ≡ 2 { finish ZIP compression }
define pdf_quick_out(#) ≡ { output a byte to PDF buffer without checking of overflow }
    begin pdf_buf[pdf_ptr] ← #; incr(pdf_ptr);
    end
define pdf_room(#) ≡ { make sure that there are at least n bytes free in PDF buffer }
    begin if pdf_os_mode ∧ (# + pdf_ptr > pdf_buf_size) then pdf_os_get_os_buf(#)
    else if ¬pdf_os_mode ∧ (# > pdf_buf_size) then overflow("PDF\output\buffer", pdf_op_buf_size)
    else if ¬pdf_os_mode ∧ (# + pdf_ptr > pdf_buf_size) then pdf_flush;
    end
define pdf_out(#) ≡ { do the same as pdf_quick_out and flush the PDF buffer if necessary }
    begin pdf_room(1); pdf_quick_out(#);
    end

{ Global variables 13 } +≡
pdf_file: byte_file; { the PDF output file }
pdf_buf: ↑eight_bits; { pointer to the PDF output buffer or PDF object stream buffer }
pdf_buf_size: integer; { end of PDF output buffer or PDF object stream buffer }
pdf_ptr: integer; { pointer to the first unused byte in the PDF buffer or object stream buffer }
pdf_op_buf: ↑eight_bits; { the PDF output buffer }
pdf_os_buf: ↑eight_bits; { the PDF object stream buffer }
pdf_os_buf_size: integer; { current size of the PDF object stream buffer, grows dynamically }
pdf_os_objnum: ↑integer; { array of object numbers within object stream }
pdf_os_objoff: ↑integer; { array of object offsets within object stream }
pdf_os_objidx: pointer; { pointer into pdf_os_objnum and pdf_os_objoff }
pdf_os_cntr: integer; { counter for object stream objects }
pdf_op_ptr: integer; { store for PDF buffer pdf_ptr while inside object streams }
pdf_os_ptr: integer; { store for object stream pdf_ptr while outside object streams }
pdf_os_mode: boolean; { true if producing object stream }
pdf_os_enable: boolean; { true if object streams are globally enabled }
pdf_os_cur_objnum: integer; { number of current object stream object }
pdf_gone: longinteger; { number of bytes that were flushed to output }
pdf_save_offset: longinteger; { to save pdf_offset }
zip_write_state: integer; { which state of compression we are in }
fixed_pdf_major_version: integer; { fixed major part of the PDF version }
fixed_pdf_minor_version: integer; { fixed minor part of the PDF version }
fixed_pdf_objcompresslevel: integer; { fixed level for activating PDF object streams }
pdf_version_written: boolean; { flag if the PDF version has been written }
fixed_pdfoutput: integer; { fixed output format }
fixed_pdfoutput_set: boolean; { fixed_pdfoutput has been set? }
fixed_gamma: integer;
fixed_image_gamma: integer;
fixed_image_hicolor: boolean;
fixed_image_apply_gamma: integer;
epochseconds: integer;
microseconds: integer;
fixed_pdf_draftmode: integer; { fixed pdfdraftmode }
fixed_pdf_draftmode_set: boolean; { fixed_pdf_draftmode has been set? }
pdf_page_group_val: integer;

```

681. ⟨ Set initial values of key variables 21 ⟩ +≡

```
pdf_gone ← 0; pdf_os_mode ← false; pdf_ptr ← 0; pdf_op_ptr ← 0; pdf_os_ptr ← 0;
pdf_os_cur_objnum ← 0; pdf_os_cntr ← 0; pdf_buf_size ← pdf_op_buf_size;
pdf_os_buf_size ← inf_pdf_os_buf_size; pdf_buf ← pdf_op_buf; pdf_seek_write_length ← false;
zip_write_state ← no_zip; pdf_version_written ← false; fixed_pdfoutput_set ← false;
fixed_pdf_draftmode_set ← false;
```

682.

```
function fix_int(val, min, max : integer): integer;
begin if val < min then fix_int ← min
else if val > max then fix_int ← max
else fix_int ← val;
end;
```

683. This ensures that *pdf_major_version* and *pdf_minor_version* are set to reasonable values before any bytes have been written to the generated PDF file. We also save their current values in case the user tries to change them later, along with *pdf_objcompresslevel*, *pdf_image_hicolor*, and various other parameters that must be fixed before any PDF output happens.

Here also the PDF file is opened by *ensure_pdf_open* and the PDF header is written.

```

procedure check_pdfversion;
begin if ¬pdf_version_written then
  begin pdf_version_written ← true;
  if pdf_major_version < 1 then
    begin print_err("pdfTeX_error_(invalid_pdfmajorversion)"); print_ln;
    help2("The_pdfmajorversion_must_be_1_or_greater.")
    ("I_changed_this_to_1."); int_error(pdf_major_version); pdf_major_version ← 1;
    end;
  if (pdf_minor_version < 0) ∨ (pdf_minor_version > 9) then
    begin print_err("pdfTeX_error_(invalid_pdfminorversion)"); print_ln;
    help2("The_pdfminorversion_must_be_between_0_and_9.")
    ("I_changed_this_to_4."); int_error(pdf_minor_version); pdf_minor_version ← 4;
    end;
  fixed_pdf_major_version ← pdf_major_version; fixed_pdf_minor_version ← pdf_minor_version;
  fixed_gamma ← fix_int(pdf_gamma, 0, 1000000);
  fixed_image_gamma ← fix_int(pdf_image_gamma, 0, 1000000);
  fixed_image_hicolor ← fix_int(pdf_image_hicolor, 0, 1);
  fixed_image_apply_gamma ← fix_int(pdf_image_apply_gamma, 0, 1);
  fixed_pdf_objcompresslevel ← fix_int(pdf_objcompresslevel, 0, 3);
  fixed_pdf_draftmode ← fix_int(pdf_draftmode, 0, 1);
  fixed_inclusion_copy_font ← fix_int(pdf_inclusion_copy_font, 0, 1);
  if ((fixed_pdf_major_version > 1) ∨ (fixed_pdf_minor_version ≥ 5)) ∧ (fixed_pdf_objcompresslevel > 0)
    then pdf_os_enable ← true
  else begin if fixed_pdf_objcompresslevel > 0 then
    begin pdf_warning("Object_streams",
      "\pdfobjcompresslevel>0 requires PDF-1.5 or greater. Object_streams_disabled now.", true, true); fixed_pdf_objcompresslevel ← 0;
    end;
    pdf_os_enable ← false;
  end;
  ensure_pdf_open; fix_pfdoutput; pdf_print("%PDF-"); pdf_print_int(fixed_pdf_major_version);
  pdf_print("."); pdf_print_int_ln(fixed_pdf_minor_version); pdf_print("%"); pdf_out(208); { 'P' + 128 }
  pdf_out(212); { 'T' + 128 }
  pdf_out(197); { 'E' + 128 }
  pdf_out(216); { 'X' + 128 }
  pdf_print_nl;
end
else begin
  if (fixed_pdf_minor_version ≠ pdf_minor_version) ∨ (fixed_pdf_major_version ≠ pdf_major_version)
    then pdf_error("setup",
      "PDF_version_cannot_be_changed_after_data_is_written_to_the_PDF_file");
end;
end;
```

684. Checks that we have a name for the generated PDF file and that it's open.

```
procedure ensure_pdf_open;
begin if output_file_name ≠ 0 then return;
if job_name = 0 then open_log_file;
pack_job_name(".pdf");
if fixed_pdf_draftmode = 0 then
  while ¬b_open_out(pdf_file) do prompt_file_name("file_name_for_output", ".pdf");
output_file_name ← b_make_name_string(pdf_file);
end;
```

685. The PDF buffer is flushed by calling *pdf_flush*, which checks the variable *zip_write_state* and will compress the buffer before flushing if necessary. We call *pdf_begin_stream* to begin a stream and *pdf_end_stream* to finish it. The stream contents will be compressed if compression is turn on.

```
procedure pdf_flush; { flush out the pdf_buf }
var saved_pdf_gone: longinteger;
begin if ¬pdf_os_mode then
begin saved_pdf_gone ← pdf_gone;
case zip_write_state of
no_zip: if pdf_ptr > 0 then
  begin if fixed_pdf_draftmode = 0 then write_pdf(0, pdf_ptr - 1);
  pdf_gone ← pdf_gone + pdf_ptr; pdf_last_byte ← pdf_buf[pdf_ptr - 1];
  end;
zip_writing: if fixed_pdf_draftmode = 0 then write_zip(false);
zip_finish: begin if fixed_pdf_draftmode = 0 then write_zip(true);
  zip_write_state ← no_zip;
  end;
end; pdf_ptr ← 0;
if saved_pdf_gone > pdf_gone then
  pdf_error("file_size", "File_size_exceeds_architectural_limits_(pdf_gone_wraps_around)");
end;
end;

procedure pdf_begin_stream; { begin a stream }
begin pdf_print_ln("/Length"); pdf_seek_write_length ← true;
{ fill in length at pdf_end_stream call }
pdf_stream_length_offset ← pdf_offset - 11; pdf_stream_length ← 0; pdf_last_byte ← 0;
if pdf_compress_level > 0 then
  begin pdf_print_ln("/Filter/FlateDecode"); pdf_print_ln(">>"); pdf_print_ln("stream"); pdf_flush;
  zip_write_state ← zip_writing;
  end
else begin pdf_print_ln(">>"); pdf_print_ln("stream"); pdf_save_offset ← pdf_offset;
  end;
end;

procedure pdf_end_stream; { end a stream }
begin if zip_write_state = zip_writing then zip_write_state ← zip_finish
else pdf_stream_length ← pdf_offset - pdf_save_offset;
pdf_flush;
if pdf_seek_write_length then write_stream_length(pdf_stream_length, pdf_stream_length_offset);
pdf_seek_write_length ← false; pdf_out(pdf_new_line_char); pdf_print_ln("endstream"); pdf_end_obj;
end;
```

686. Basic printing procedures for PDF output are very similar to TeX basic printing ones but the output is going to PDF buffer. Subroutines with suffix *_ln* append a new-line character to the PDF output.

```

define pdf_new_line_char ≡ 10 { new-line character for UNIX platforms }
define pdf_print_nl ≡ { output a new-line character to PDF buffer }
    pdf_out(pdf_new_line_char)
define pdf_print_ln(#) ≡ { print out a string to PDF buffer followed by a new-line character }
    begin pdf_print(#); pdf_print_nl;
    end
define pdf_print_int_ln(#) ≡ { print out an integer to PDF buffer followed by a new-line character }
    begin pdf.print_int(#); pdf.print_nl;
    end

⟨ Declare procedures that need to be declared forward for pdfTeX 686 ⟩ ≡
procedure pdf_error(t, p : str_number);
    begin normalize_selector; print_err("pdfTeX_error");
    if t ≠ 0 then
        begin print("✉("); print(t); print(")");
        end;
    print("✉"); print(p); succumb;
    end;

procedure pdf_warning(t, p : str_number; prepend_nl, append_nl : boolean);
    begin if interaction = error_stop_mode then wake_up_terminal;
    if prepend_nl then print_ln;
    print("pdfTeX_warning");
    if t ≠ 0 then
        begin print("✉("); print(t); print(")");
        end;
    print("✉"); print(p);
    if append_nl then print_ln;
    if history = spotless then history ← warning_issued;
    end;

procedure pdf_os_get_os_buf(s : integer);
    { check that s bytes more fit into pdf_os_buf; increase it if required }
    var a: integer;
    begin if s > sup_pdf_os_buf_size - pdf_ptr then
        overflow("PDF_object_stream_buffer", pdf_os_buf_size);
    if pdf_ptr + s > pdf_os_buf_size then
        begin a ← 0.2 * pdf_os_buf_size;
        if pdf_ptr + s > pdf_os_buf_size + a then pdf_os_buf_size ← pdf_ptr + s
        else if pdf_os_buf_size < sup_pdf_os_buf_size - a then pdf_os_buf_size ← pdf_os_buf_size + a
        else pdf_os_buf_size ← sup_pdf_os_buf_size;
        pdf_os_buf ← xrealloc_array(pdf_os_buf, eight_bits, pdf_os_buf_size); pdf_buf ← pdf_os_buf;
        pdf_buf_size ← pdf_os_buf_size;
        end;
    end;
end;

procedure remove_last_space;
    begin if (pdf_ptr > 0) ∧ (pdf_buf[pdf_ptr - 1] = 32) then decr(pdf_ptr);
    end;

procedure pdf_print_octal(n : integer); { prints an integer in octal form to PDF buffer }
    var k: 0 .. 23; { index to current digit; we assume that n < 1023 }
    begin k ← 0;
    repeat dig[k] ← n mod 8; n ← n div 8; incr(k);
    until n = 0;

```

```

if  $k = 1$  then
  begin  $\text{pdf\_out}("0"); \text{pdf\_out}("0");$ 
  end;
if  $k = 2$  then  $\text{pdf\_out}("0");$ 
while  $k > 0$  do
  begin  $\text{decr}(k); \text{pdf\_out}("0" + \text{dig}[k]);$ 
  end;
end;
procedure  $\text{pdf\_print\_char}(f : \text{internal\_font\_number}; c : \text{integer})$ ;
  { print out a character to PDF buffer; the character will be printed in octal form in the following
    cases: chars  $j = 32$ , backslash (92), left parenthesis (40) and right parenthesis (41) }
begin  $\text{pdf\_mark\_char}(f, c);$ 
if  $(c \leq 32) \vee (c = 92) \vee (c = 40) \vee (c = 41) \vee (c > 127)$  then
  begin  $\text{pdf\_out}(92);$  { output a backslash }
   $\text{pdf\_print\_octal}(c);$ 
  end
else  $\text{pdf\_out}(c);$ 
end;
procedure  $\text{pdf\_print}(s : \text{str\_number});$  { print out a string to PDF buffer }
  var  $j : \text{pool\_pointer};$  { current character code position }
   $c : \text{integer};$ 
begin  $j \leftarrow \text{str\_start}[s];$ 
while  $j < \text{str\_start}[s + 1]$  do
  begin  $c \leftarrow \text{str\_pool}[j]; \text{pdf\_out}(c); \text{incr}(j);$ 
  end;
end;
function  $\text{str\_in\_str}(s, r : \text{str\_number}; i : \text{integer}) : \text{boolean};$  { test equality of strings }
label not_found; { loop exit }
var  $j, k : \text{pool\_pointer};$  { running indices }
begin  $\text{str\_in\_str} \leftarrow \text{false};$ 
if  $\text{length}(s) < i + \text{length}(r)$  then return;
 $j \leftarrow i + \text{str\_start}[s]; k \leftarrow \text{str\_start}[r];$ 
while  $(j < \text{str\_start}[s + 1]) \wedge (k < \text{str\_start}[r + 1])$  do
  begin if  $\text{str\_pool}[j] \neq \text{str\_pool}[k]$  then return;
     $\text{incr}(j); \text{incr}(k);$ 
  end;
   $\text{str\_in\_str} \leftarrow \text{true};$ 
end;
procedure  $\text{pdf\_print\_int}(n : \text{longinteger});$  { print out a integer to PDF buffer }
  var  $k : \text{integer};$  { index to current digit ( $0 \leq k \leq 23$ ); we assume that  $n < 10^{23}$  }
   $m : \text{longinteger};$  { used to negate  $n$  in possibly dangerous cases }
begin  $k \leftarrow 0;$ 
if  $n < 0$  then
  begin  $\text{pdf\_out}("-");$ 
  if  $n > -100000000$  then  $\text{negate}(n)$ 
  else begin  $m \leftarrow -1 - n; n \leftarrow m \text{ div } 10; m \leftarrow (m \text{ mod } 10) + 1; k \leftarrow 1;$ 
    if  $m < 10$  then  $\text{dig}[0] \leftarrow m$ 
    else begin  $\text{dig}[0] \leftarrow 0; \text{incr}(n);$ 
      end;
    end;
  end;
repeat  $\text{dig}[k] \leftarrow n \text{ mod } 10; n \leftarrow n \text{ div } 10; \text{incr}(k);$ 

```

```

until  $n = 0$ ;
pdf_room( $k$ );
while  $k > 0$  do
begin  $decr(k)$ ; pdf_quick_out("0" + dig[ $k$ ]);
end;
end;

procedure pdf_print_two( $n : integer$ ); { prints two least significant digits in decimal form to PDF buffer }
begin  $n \leftarrow abs(n) \bmod 100$ ; pdf_out("0" + ( $n \div 10$ )); pdf_out("0" + ( $n \bmod 10$ ));
end;

function tokens_to_string( $p : pointer$ ): str_number; { return a string from tokens list }
begin if selector = new_string then
pdf_error("tokens", "tokens_to_string() called while selector = new_string");
old_setting  $\leftarrow$  selector; selector  $\leftarrow$  new_string; show_token_list(link( $p$ ), null, pool_size - pool_ptr);
selector  $\leftarrow$  old_setting; last_tokens_string  $\leftarrow$  make_string; tokens_to_string  $\leftarrow$  last_tokens_string;
end;

```

See also sections 689, 698, 699, 700, 703, 1545, and 1555.

This code is used in section 190.

687. To print *scaled* value to PDF output we need some subroutines to ensure accuracy.

```

define max_integer  $\equiv$  "7FFFFFFF {  $2^{31} - 1$  }
define call_func(#)  $\equiv$ 
begin if #  $\neq 0$  then do_nothing
end

⟨ Global variables 13 ⟩  $\equiv$ 
one_bp: scaled; { scaled value corresponds to 1bp }
one_hundred_bp: scaled; { scaled value corresponds to 100bp }
one_hundred_inch: scaled; { scaled value corresponds to 100in }
ten_pow: array [0 .. 9] of integer; {  $10^0..10^9$  }
scaled_out: integer; { amount of scaled that was taken out in divide_scaled }
init_pdf_output: boolean;
adv_char_width_s: integer; { to save result of calculation done in adv_char_width }
adv_char_width_s_out: scaled;

```

688. ⟨ Set initial values of key variables 21 ⟩ \equiv

```

one_bp  $\leftarrow$  65782; { 65781.76 }
one_hundred_bp  $\leftarrow$  6578176; one_hundred_inch  $\leftarrow$  473628672; ten_pow[0]  $\leftarrow$  1;
for  $i \leftarrow 1$  to 9 do ten_pow[i]  $\leftarrow$  10 * ten_pow[i - 1];
init_pdf_output  $\leftarrow$  false;

```

689. The following function divides s by m . dd is number of decimal digits.

```
< Declare procedures that need to be declared forward for pdfTeX 686 > +≡
function divide_scaled(s, m : scaled; dd : integer): scaled;
  var q, r: scaled; sign, i: integer;
  begin sign ← 1;
  if s < 0 then
    begin sign ← -sign; s ← -s;
    end;
  if m < 0 then
    begin sign ← -sign; m ← -m;
    end;
  if m = 0 then pdf_error("arithmetic", "divided_by_zero")
  else if m ≥ (max_integer div 10) then pdf_error("arithmetic", "number_too_big");
  q ← s div m; r ← s mod m;
  for i ← 1 to dd do
    begin q ← 10 * q + (10 * r) div m; r ← (10 * r) mod m;
    end;
  if 2 * r ≥ m then
    begin incr(q); r ← r - m;
    end;
  scaled_out ← sign * (s - (r div ten_pow[dd])); divide_scaled ← sign * q;
  end;
function round_xn_over_d(x : scaled; n, d : integer): scaled;
  var positive: boolean; { was x ≥ 0? }
  t, u, v: nonnegative_integer; { intermediate quantities }
  begin if x ≥ 0 then positive ← true
  else begin negate(x); positive ← false;
  end;
  t ← (x mod `100000) * n; u ← (x div `100000) * n + (t div `100000);
  v ← (u mod d) * `100000 + (t mod `100000);
  if u div d ≥ `100000 then arith_error ← true
  else u ← `100000 * (u div d) + (v div d);
  v ← v mod d;
  if 2 * v ≥ d then incr(u);
  if positive then round_xn_over_d ← u
  else round_xn_over_d ← -u;
  end;
```

690. Next subroutines are needed for controlling spacing in PDF page description. For a given character c from a font f , the procedure *adv_char_width* advances pdf_h by *about* the amount w , which is the character width. But we cannot simply add w to pdf_h . Instead we have to bring the required shift into the same raster, on which also the */Widths* array values, as they appear in the PDF file, are based. The *scaled_out* value is the w value moved into this raster. The */Widths* values are used by the PDF reader independently to update its positions. So one has to be sure, that calculations are properly synchronized. Currently the */Widths* array values are output with one digit after the decimal point, therefore the raster on which *adv_char_width* is operating is 1/10000 of the *pdf_font_size*.

For PK fonts things are more complicated, as we have to deal with scaling bitmaps as well.

```

procedure adv_char_width(f : internal_font_number; c : eight_bits; dd : eight_bits);
    { update pdf_delta_h by character width w from font f }
    var w, s_out: scaled; s: integer;
    begin w ← char_width(f)(char_info(f)(c));
    if isscalable(f) then
        begin if pdf_cur_Tm_a = 0 then
            begin s ← divide_scaled(w, pdf_font_size[f], dd); s_out ← scaled_out;
            pdf_delta_h ← pdf_delta_h + s_out;
            end
        else begin s ← divide_scaled(round_xn_over_d(w, 1000, 1000 + pdf_cur_Tm_a), pdf_font_size[f], dd);
            s_out ← round_xn_over_d(round_xn_over_d(pdf_font_size[f], abs(s), 10000), 1000 + pdf_cur_Tm_a, 1000);
            if s < 0 then s_out ← -s_out;
            pdf_delta_h ← pdf_delta_h + s_out;
            end;
        adv_char_width_s ← s; adv_char_width_s_out ← s_out;
        end
    else pdf_delta_h ← pdf_delta_h + get_pk_char_width(f, w);
    end;
procedure pdf_print_real(m, d : integer); { print m/10d as real }
begin if m < 0 then
    begin pdf_out("-"); m ← -m;
    end;
pdf_print_int(m div ten_pow[d]); m ← m mod ten_pow[d];
if m > 0 then
    begin pdf_out("."); decr(d);
    while m < ten_pow[d] do
        begin pdf_out("0"); decr(d);
        end;
    while m mod 10 = 0 do m ← m div 10;
    pdf_print_int(m);
    end;
end;
procedure pdf_print_bp(s : scaled); { print scaled as bp }
begin pdf_print_real(divide_scaled(s, one_hundred_bp, fixed_decimal_digits + 2), fixed_decimal_digits);
end;
procedure pdf_print_mag_bp(s : scaled); { take mag into account }
begin prepare_mag;
if mag ≠ 1000 then s ← round_xn_over_d(s, mag, 1000);
pdf_print_bp(s);
end;
```

691. PDF page description.

```

define pdf_x(#)  $\equiv$  ((#) – pdf_origin_h) { convert x-coordinate from DVI to PDF }
define pdf_y(#)  $\equiv$  (pdf_origin_v – (#)) { convert y-coordinate from DVI to PDF }
define dvi_x(#)  $\equiv$  ((#) + pdf_origin_h) { convert x-coordinate from PDF to DVI }
define dvi_y(#)  $\equiv$  (pdf_origin_v – (#)) { convert y-coordinate from PDF to DVI }

{Global variables 13}  $\equiv$ 

pdf_f: internal_font_number; { the current font in PDF output page }
pdf_h: scaled; { current horizontal coordinate in PDF output page }
pdf_v: scaled; { current vertical coordinate in PDF output page }
pdf_tj_start_h: scaled; { horizontal coordinate in PDF output page just before TJ array start }
cur_delta_h: scaled; { horizontal cur_h offset from pdf_tj_start_h }
pdf_delta_h: scaled; { horizontal offset from pdf_tj_start_h }
pdf_origin_h: scaled; { current horizontal origin in PDF output page }
pdf_origin_v: scaled; { current vertical origin in PDF output page }
pdf_doing_string: boolean; { we are writing string to PDF file? }
pdf_doing_text: boolean; { we are writing text section to PDF file? }
min_bp_val: scaled;
min_font_val: scaled; { (TJ array system) }
fixed_pk_resolution: integer;
fixed_decimal_digits: integer;
fixed_gen_tounicode: integer;
fixed_inclusion_copy_font: integer;
pk_scale_factor: integer;
pdf_output_option: integer;
pdf_output_value: integer;
pdf_draftmode_option: integer;
pdf_draftmode_value: integer;
pdf_cur_Tm_a: integer; { a value of the current text matrix, i.e., the current horizontal scaling factor }
pdf_last_f: internal_font_number; { last font in PDF output page }
pdf_last_fs: internal_font_number; { last font size in PDF output page }
pdf_dummy_font: internal_font_number; { font used to insert artificial interword spaces }

```

692. Following procedures implement low-level subroutines to convert TeX internal structures to PDF page description.

```

procedure pdf_set_origin(h, v : scaled); { set the origin to h, v }
begin if (abs(h - pdf_origin_h) ≥ min_bp_val) ∨ (abs(v - pdf_origin_v) ≥ min_bp_val) then
  begin pdf_print("1\u00000\u0001\u000"); pdf_print_bp(h - pdf_origin_h);
    pdf_origin_h ← pdf_origin_h + scaled_out; pdf_out(" "); pdf_print_bp(pdf_origin_v - v);
    pdf_origin_v ← pdf_origin_v - scaled_out; pdf_print_ln(" \u000cm");
  end;
  pdf_h ← pdf_origin_h; pdf_tj_start_h ← pdf_h; pdf_v ← pdf_origin_v;
end;

procedure pdf_set_origin_temp(h, v : scaled); { set the origin to h, v inside group }
begin if (abs(h - pdf_origin_h) ≥ min_bp_val) ∨ (abs(v - pdf_origin_v) ≥ min_bp_val) then
  begin pdf_print("1\u00000\u0001\u000"); pdf_print_bp(h - pdf_origin_h); pdf_out(" ");
    pdf_print_bp(pdf_origin_v - v); pdf_print_ln(" \u000cm");
  end;
end;

procedure pdf_end_string; { end the current string }
begin if pdf_doing_string then
  begin pdf_print(") TJ"); pdf_doing_string ← false;
  end;
end;

procedure pdf_end_string_nl; { end the current string, with new-line }
begin if pdf_doing_string then
  begin pdf_print_ln(") TJ"); pdf_doing_string ← false;
  end;
end;

procedure pdf_set_textmatrix(v, v_out : scaled; f : internal_font_number);
{ set the next starting point to cur_h, cur_v }
var pdf_new_Tm_a: integer; { a value of the new text matrix }
begin pdf_out(" ");
if f = pdf_f then pdf_new_Tm_a ← pdf_cur_Tm_a
else if not pdf_font_auto_expand[f] then pdf_new_Tm_a ← 0
else pdf_new_Tm_a ← pdf_font_expand_ratio[f]
if (pdf_new_Tm_a ≠ 0) ∨ ((pdf_new_Tm_a = 0) ∧ (pdf_cur_Tm_a ≠ 0)) then
  begin pdf_print_real(1000 + pdf_new_Tm_a, 3); pdf_print(" \u00000\u0001\u000");
    pdf_print_bp(cur_h - pdf_origin_h); pdf_h ← pdf_origin_h + scaled_out; pdf_out(" ");
    pdf_print_bp(pdf_origin_v - cur_v); pdf_v ← pdf_origin_v - scaled_out; pdf_print(" \u000Tm");
    pdf_cur_Tm_a ← pdf_new_Tm_a; pdfassert(pdf_cur_Tm_a > -1000);
  end
else begin pdf_print_bp(cur_h - pdf_tj_start_h); { works only for unexpanded fonts }
  pdf_h ← pdf_tj_start_h + scaled_out; pdf_out(" "); pdf_print_real(v, fixed_decimal_digits);
  { use v and v_out to avoid duplicate calculation }
  pdf_v ← pdf_v - v_out; pdf_print(" \u000Td");
end;
pdf_tj_start_h ← pdf_h; pdf_delta_h ← 0;
end;

procedure pdf_use_font(f : internal_font_number; fontnum : integer);
{ mark f as a used font; set font_used[f], pdf_font_size[f] and pdf_font_num[f] }
begin call_func(divide_scaled(font_size[f], one_hundred_bp, 6)); pdf_font_size[f] ← scaled_out;
font_used[f] ← true; pdfassert((fontnum > 0) ∨ ((fontnum < 0) ∧ (pdf_font_num[-fontnum] > 0)));
pdf_font_num[f] ← fontnum;
if pdf_move_chars > 0 then

```

```
begin pdf_warning(0, "Primitive \pdfmovechars is obsolete.", true, true); pdf_move_chars ← 0;  
  { warn only once }  
end;  
end;
```

693. To set PDF font we need to find out fonts with the same name, because TeX can load the same font several times for various sizes. For such fonts we define only one font resource. The array *pdf_font_num* holds the object number of font resource. A negative value of an entry of *pdf_font_num* indicates that the corresponding font shares the font resource with the font.

```

define pdf_print_resname_prefix ≡
  if pdf_resname_prefix ≠ 0 then pdf_print(pdf_resname_prefix)

procedure pdf_init_font(f : internal_font_number); { create a font object }
  var k, b: internal_font_number; i: integer;
  begin pdfassert(¬font_used[f]); { if f is auto expanded then ensure the base font is initialized }
  if pdf_font_auto_expand[f] ∧ (pdf_font_blink[f] ≠ null_font) then
    begin b ← pdf_font_blink[f];
    if ¬isscalable(b) then
      pdf_error("font_expansion", "auto_expansion_is_only_possible_with_scalable_fonts");
    if ¬font_used[b] then pdf_init_font(b);
    pdf_font_map[f] ← pdf_font_map[b];
  end; { check whether f can share the font object with some k: we have 2 cases here: 1) f and k have
    the same tfm name (so they have been loaded at different sizes, e.g., 'cmr10' and 'cmr10 at 11pt');
    2) f has been auto expanded from k }
  if isscaleable(f) then
    begin i ← head.tab[obj_type_font];
    while i ≠ 0 do
      begin k ← obj_info(i);
      if isscaleable(k) ∧ (pdf_font_map[k] = pdf_font_map[f]) ∧ (str_eq_str(font_name[k],
        font_name[f]) ∨ (pdf_font_auto_expand[f] ∧ (pdf_font_blink[f] ≠
        null_font) ∧ str_eq_str(font_name[k], font_name[pdf_font_blink[f]]))) then
        begin pdfassert(pdf_font_num[k] ≠ 0);
        if pdf_font_num[k] < 0 then pdf_use_font(f, pdf_font_num[k])
        else pdf_use_font(f, -k);
        return;
      end;
      i ← obj_link(i);
    end;
  end; { create a new font object for f }
  pdf_create_obj(obj_type_font, f); pdf_font_has_space_char[f] ← hasspacechar(f); pdf_use_font(f, obj_ptr);
  end;
procedure pdf_init_font_cur_val;
  begin pdf_init_font(cur_val);
  end;
procedure pdf_set_font(f : internal_font_number); { set the actual font on PDF page }
  label found, found1;
  var p: pointer; k: internal_font_number;
  begin if ¬font_used[f] then pdf_init_font(f);
  set_ff(f); { set ff to the tfm number of the font sharing the font object with f; ff is either f or some
    font with the same tfm name at different size and/or expansion }
  k ← ff; p ← pdf_font_list;
  while p ≠ null do
    begin set_ff(info(p));
    if ff = k then goto found;
    p ← link(p);
  end;
  pdf_append_list(f)(pdf_font_list); { f not found in pdf_font_list, append it now }
  found: if (k = pdf_last_f) ∧ (font_size[f] = pdf_last_fs) then return;

```

```

pdf_print("/F"); pdf_print_int(k); pdf_print_resname_prefix; pdf_out("U");
pdf_print_real(divide_scaled(font_size[f], one_hundred_bp, 6), 4); pdf_print("UTf"); pdf_last_f ← k;
pdf_last_fs ← font_size[f];
end;
procedure pdf_begin_text; { begin a text section }
begin pdf_set_origin(0, cur_page_height); pdf_print_ln("BT"); pdf_doing_text ← true; pdf_f ← null_font;
pdf_last_f ← null_font; pdf_last_fs ← 0; pdf_doing_string ← false; pdf_cur_Tm_a ← 0;
end;
procedure pdf_read_dummy_font;
begin if pdf_dummy_font = null_font then
begin pdf_dummy_font ← read_font_info(null_cs, pdf_space_font_name, "", -1000); pdfmaplinesp;
pdf_mark_char(pdf_dummy_font, 32);
end;
end;
procedure pdf_insert_interword_space; { insert an artificial interword space }
begin pdf_read_dummy_font; pdf_set_font(pdf_dummy_font); pdf_print("(U)Tj");
end;
procedure pdf_begin_string(f : internal_font_number); { begin to draw a string }
var s_out, v, v_out: scaled; save_pdf_delta_h: scaled; s: integer; must_end_string: boolean;
{ must we end the current string? }
must_insert_space: boolean; { must we insert an interword space? }
begin if ¬pdf_doing_text then pdf_begin_text;
if f ≠ pdf_f then
begin pdf_end_string; pdf_set_font(f);
end;
if pdf_cur_Tm_a = 0 then
begin s ← divide_scaled(cur_h - (pdf_tj_start_h + pdf_delta_h), pdf_font_size[f], 3); s_out ← scaled_out;
end
else begin s ← divide_scaled(round_xn_over_d(cur_h - (pdf_tj_start_h + pdf_delta_h), 1000,
1000 + pdf_cur_Tm_a), pdf_font_size[f], 3);
if abs(s) < '100000 then
begin s_out ← round_xn_over_d(round_xn_over_d(pdf_font_size[f], abs(s), 1000),
1000 + pdf_cur_Tm_a, 1000);
if s < 0 then s_out ← -s_out;
end; { no need to calculate s_out when abs(s) ≥ '100000, since the text matrix will be reset below }
end;
if abs(cur_v - pdf_v) ≥ min_bp_val then
begin v ← divide_scaled(pdf_v - cur_v, one_hundred_bp, fixed_decimal_digits + 2); v_out ← scaled_out;
end
else begin v ← 0; v_out ← 0;
end;
must_insert_space ← false; must_end_string ← false;
if (f ≠ pdf_f) ∨ (v ≠ 0) ∨ (abs(s) ≥ '100000) then
begin must_end_string ← true;
end;
if gen_faked_interword_space ∧ pdf_doing_string ∧ (¬must_end_string) ∧ (s_out >
space(f) - space_shrink(f) - 65536) ∧ (v = 0) then
begin must_insert_space ← true;
end;
if (must_insert_space) then
begin { insert a real space char from the font when possible }
if pdf_font_has_space_char[f] then

```

```

begin pdf_out(" "); save_pdf_delta_h ← pdf_delta_h; adv_char_width(f, 32, 3);
  { to get adv_char_width_s and adv_char_width_s_out }
  s ← s - adv_char_width_s; s_out ← s_out - adv_char_width_s_out; pdf_mark_char(f, 32);
end
else must_end_string ← true;
end;
if must_end_string then
begin pdf_end_string; { insert a space char from the dummy font if needed }
if (must_insert_space) ∧ (¬pdf_font_has_space_char[f]) then
begin pdf_insert_interword_space; { this will change pdf_f }
pdf_set_font(f);
end;
pdf_set_textmatrix(v, v_out, f); pdf_f ← f; s ← 0;
end;
if ¬pdf_doing_string then
begin pdf_print("[" );
if s = 0 then pdf_out("(" );
end;
if s ≠ 0 then
begin if pdf_doing_string then pdf_out(")");
pdf_print_int(−s); pdf_out("(" ); pdf_delta_h ← pdf_delta_h + s_out;
end;
pdf_doing_string ← true;
end;
procedure pdf_insert_fake_space;
var s: integer; { to save gen_faked_interword_space }
begin s ← gen_faked_interword_space; gen_faked_interword_space ← 0;
{ to prevent inserting another fake space in pdf_begin_string }
pdf_read_dummy_font; pdf_begin_string(pdf_dummy_font); pdf_print("[" );
pdf_end_string_nl;
gen_faked_interword_space ← s;
end;
procedure pdf_end_text; { end a text section }
begin if pdf_doing_text then
begin pdf_end_string_nl; pdf_print_ln("ET"); pdf_doing_text ← false;
end;
end;
procedure pdf_set_rule(x, y, w, h : scaled); { draw a rule }
begin pdf_end_text; pdf_print_ln("q");
if h ≤ one_bp then
begin pdf_set_origin_temp(x, y − (h + 1)/2); pdf_print("[]0d0J"); pdf_print_bp(h);
pdf_print("w00m"); pdf_print_bp(w); pdf_print_ln("0lS");
end
else if w ≤ one_bp then
begin pdf_set_origin_temp(x + (w + 1)/2, y); pdf_print("[]0d0J"); pdf_print_bp(w);
pdf_print("w00m0"); pdf_print_bp(h); pdf_print_ln("1S");
end
else begin pdf_set_origin_temp(x, y); pdf_print("00"); pdf_print_bp(w); pdf_out("[" );
pdf_print_bp(h); pdf_print_ln("ref");
end;
pdf_print_ln("Q");
end;
procedure pdf_rectangle(left, top, right, bottom : scaled); { output a rectangle specification to PDF file }

```

```

begin prepare_mag; pdf_print("/Rect["); pdf_print_mag_bp(pdf_x(left)); pdf_out("]");
pdf_print_mag_bp(pdf_y(bottom)); pdf_out("]"); pdf_print_mag_bp(pdf_x(right)); pdf_out("]");
pdf_print_mag_bp(pdf_y(top)); pdf_print_ln("] ");
end; { Prints first len characters of string s (if it's that long). There must be a better way to print a
substring? }

procedure slow_print_substr(s, max_len : integer);
var j: pool_pointer; { current character code position }
begin if (s ≥ str_ptr) ∨ (s < 256) then print(s)
else begin j ← str_start[s];
while (j < str_start[s + 1]) ∧ (j ≤ str_start[s] + max_len) do
begin print(so(str_pool[j])); incr(j);
end;
end;
if j < str_start[s + 1] then print("..."); { indicate truncation }
end;

procedure literal(s : str_number; literal_mode : integer; warn : boolean);
var j: pool_pointer; { current character code position }
begin j ← str_start[s];
if literal_mode = scan_special then
begin if ¬(str_in_str(s, "PDF:", 0) ∨ str_in_str(s, "pdf:", 0)) then
begin if warn ∧ ¬(str_in_str(s, "SRC:", 0) ∨ str_in_str(s, "src:", 0) ∨ (length(s) = 0)) then
begin print_nl("Non-PDF special ignored!"); print_nl("<special>]");
slow_print_substr(s, 64); { length of printed line should be j=78; good enough. }
print_ln;
end;
return;
end;
j ← j + length("PDF:");
if str_in_str(s, "direct:", length("PDF:")) then
begin j ← j + length("direct:"); literal_mode ← direct_always;
end
else if str_in_str(s, "page:", length("PDF:")) then
begin j ← j + length("page:"); literal_mode ← direct_page;
end
else literal_mode ← set_origin;
end;
case literal_mode of
set_origin: begin pdf_end_text; pdf_set_origin(cur_h, cur_v);
end;
direct_page: pdf_end_text;
direct_always: pdf_end_string_nl;
othercases confusion("literal1")
endcases;
while j < str_start[s + 1] do
begin pdf_out(str_pool[j]); incr(j);
end;
pdf_print_nl;
end;

```

694. The cross-reference table. The cross-reference table *obj_tab* is an array of *obj_tab_size* of *obj_entry*. ■
Each entry contains five integer fields and represents an object in PDF file whose object number is the index of this entry in *obj_tab*. Objects in *obj_tab* maybe linked into list; objects in such a linked list have the same type.

⟨ Types in the outer block 18 ⟩ +≡
obj_entry = record *int0*, *int1*: integer;
 int2: longinteger;
 int3, *int4*: integer;
end;

695. The first field contains information representing identifier of this object. It is usually a number for most of object types, but it may be a string number for named destination or named thread.

The second field of *obj_entry* contains link to the next object in *obj_tab* if this object is linked in a list.

The third field holds the byte offset of the object in the output PDF file, or its byte offset within an object stream. As long as the object is not written, this field is used for flags about the write status of the object; then it has a negative value.

The fourth field holds the object number of the object stream, into which the object is included.

The last field usually represents the pointer to some auxiliary data structure depending on the object type; however it may be used as a counter as well.

```

define obj_info(#)≡ obj_tab[#].int0 { information representing identifier of this object }
define obj_link(#)≡ obj_tab[#].int1 { link to the next entry in linked list }
define obj_offset(#)≡ obj_tab[#].int2 { negative (flags), or byte offset for this object in PDF output
    file, or object stream number for this object }
define obj_os_idx(#)≡ obj_tab[#].int3 { index of this object in object stream }
define obj_aux(#)≡ obj_tab[#].int4 { auxiliary pointer }
define set_obj_fresh(#)≡ obj_offset(#)← -2
define set_obj_scheduled(#)≡
    if obj_offset(#)= -2 then obj_offset(#)← -1
define is_obj_scheduled(#)≡ (obj_offset(#)> -2)
define is_obj_written(#)≡ (obj_offset(#)> -1)
    { types of objects }
define obj_type_others $\equiv$  0 { objects which are not linked in any list }
define obj_type_page $\equiv$  1 { index of linked list of Page objects }
define obj_type_pages $\equiv$  2 { index of linked list of Pages objects }
define obj_type_font $\equiv$  3 { index of linked list of Fonts objects }
define obj_type_outline $\equiv$  4 { index of linked list of outline objects }
define obj_type_dest $\equiv$  5 { index of linked list of destination objects }
define obj_type_struct_dest $\equiv$  6 { index of linked list of structure destination objects }
define obj_type_obj $\equiv$  7 { index of linked list of raw objects }
define obj_type_xform $\equiv$  8 { index of linked list of XObject forms }
define obj_type_ximage $\equiv$  9 { index of linked list of XObject image }
define obj_type_thread $\equiv$  10 { index of linked list of num article threads }
define head_tab_max $\equiv$  obj_type_thread { max index of head_tab }
    { max number of kids for balanced trees }
define pages_tree_kids_max $\equiv$  6 { max number of kids of Pages tree node }
define name_tree_kids_max $\equiv$  6 { max number of kids of node of name tree for name destinations }
    { when a whatsit node representing annotation is created, words 1 .. 3 are width, height and
    depth of this annotation; after skipping out words 1 .. 4 are rectangle specification of
    annotation. For whatsit node representing destination pdf_left and pdf_top are used for some
    types of destinations }
    { coordinates of destinations/threads/annotations (in whatsit node) }
define pdf_left(#)≡ mem[# + 1].sc
define pdf_top(#)≡ mem[# + 2].sc
define pdf_right(#)≡ mem[# + 3].sc
define pdf_bottom(#)≡ mem[# + 4].sc
    { dimension of destinations/threads/annotations (in whatsit node) }
define pdf_width(#)≡ mem[# + 1].sc
define pdf_height(#)≡ mem[# + 2].sc
define pdf_depth(#)≡ mem[# + 3].sc
    { data structure for \pdfliteral }
define pdf_literal_data(#)≡ link(# + 1) { data }
```

```

define pdf_literal_mode(#) $\equiv$  info(# + 1)
    { mode of resetting the text matrix while writing data to the page stream }
    { modes of setting the current transformation matrix (CTM) }
define set_origin $\equiv$  0 { end text (ET) if needed, set CTM to current point }
define direct_page $\equiv$  1 { end text (ET) if needed, but don't change the CTM }
define direct_always $\equiv$  2 { don't end text, don't change the CTM }
define scan_special $\equiv$  3 { look into special text }

    { data structure for \pdfcolorstack }
define pdf_colorstack_node_size $\equiv$  3
define pdf_colorstack_setter_node_size $\equiv$  3
define pdf_colorstack_getter_node_size $\equiv$  2
define pdf_colorstack_stack(#) $\equiv$  link(# + 1) { stack number }
define pdf_colorstack_cmd(#) $\equiv$  info(# + 1) { command: set, push, pop, current }
define pdf_colorstack_data(#) $\equiv$  link(# + 2) { data }

    { color stack commands }
define colorstack_set $\equiv$  0
define colorstack_push $\equiv$  1
define colorstack_data $\equiv$  1 { last value where data field is set }
define colorstack_pop $\equiv$  2
define colorstack_current $\equiv$  3

    { data structure for \pdfsetmatrix }
define pdf_setmatrix_node_size $\equiv$  2
define pdf_setmatrix_data(#) $\equiv$  link(# + 1) { data }

    { data structure for \pdfsave }
define pdf_save_node_size $\equiv$  2
    { data structure for \pdfrestore }
define pdf_restore_node_size $\equiv$  2

    { data structure for \pdfobj and \pdfrefobj }
define pdf_refobj_node_size $\equiv$  2 { size of whatsit node representing the raw object }
define pdf_obj_objnum(#) $\equiv$  info(# + 1) { number of the raw object }
define obj_data_ptr $\equiv$  obj_aux { pointer to pdf_mem }
define pdfmem_obj_size $\equiv$  4 { size of memory in pdf_mem which obj_data_ptr holds }
define obj_obj_data(#) $\equiv$  pdf_mem[obj_data_ptr(#)] { object data }
define obj_obj_is_stream(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 1
    { will this object be written as a stream instead of a dictionary? }
define obj_obj_stream_attr(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 2 { additional object attributes for streams }
define obj_obj_is_file(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 3 { data should be read from an external file? }

    { data structure for \pdfxform and \pdfrefxform }
define pdf_refxform_node_size $\equiv$  5 { size of whatsit node for xform; words 1..3 are form dimensions }
define pdf_xform_objnum(#) $\equiv$  info(# + 4) { object number }
define pdfmem_xform_size $\equiv$  6 { size of memory in pdf_mem which obj_data_ptr holds }
define obj_xform_width(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 0
define obj_xform_height(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 1
define obj_xform_depth(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 2
define obj_xform_box(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 3
    { this field holds pointer to the corresponding box }
define obj_xform_attr(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 4 { additional xform attributes }
define obj_xform_resources(#) $\equiv$  pdf_mem[obj_data_ptr(#)] + 5 { additional xform Resources }

    { data structure for \pdfximage and \pdfrefximage }
define pdf_refximage_node_size $\equiv$  5 { size of whatsit node for ximage; words 1..3 are image dimensions }

```

```

define pdf_ximage_objnum(#) ≡ info(# + 4) { object number }
define pdfmem_ximage_size ≡ 5 { size of memory in pdf_mem which obj_data_ptr holds }
define obj_ximage_width(#) ≡ pdf_mem[obj_data_ptr(#) + 0]
define obj_ximage_height(#) ≡ pdf_mem[obj_data_ptr(#) + 1]
define obj_ximage_depth(#) ≡ pdf_mem[obj_data_ptr(#) + 2]
define obj_ximage_attr(#) ≡ pdf_mem[obj_data_ptr(#) + 3] { additional ximage attributes }
define obj_ximage_data(#) ≡ pdf_mem[obj_data_ptr(#) + 4] { pointer to image data }

{ data structure of annotations; words 1..4 represent the coordinates of the annotation }
define obj_annot_ptr ≡ obj_aux { pointer to corresponding whatsit node }
define pdf_annot_node_size ≡ 7 { size of whatsit node representing annotation }
define pdf_annot_data(#) ≡ info(# + 5) { raw data of general annotations }
define pdf_link_attr(#) ≡ info(# + 5) { attributes of link annotations }
define pdf_link_action(#) ≡ link(# + 5) { pointer to action structure }
define pdf_annot_objnum(#) ≡ mem[# + 6].int { object number of corresponding object }
define pdf_link_objnum(#) ≡ mem[# + 6].int { object number of corresponding object }

{ types of actions }
define pdf_action_page ≡ 0 { GoTo action }
define pdf_action_goto ≡ 1 { GoTo action }
define pdf_action_thread ≡ 2 { Thread action }
define pdf_action_user ≡ 3 { user-defined action }

{ data structure of actions }
define pdf_action_size ≡ 4 { size of action structure in mem }
define pdf_action_type ≡ type { action type }
define pdf_action_named_id ≡ subtype { identifier is type of name }
define pdf_action_id ≡ link { destination/thread name identifier }
define pdf_action_file(#) ≡ info(# + 1) { file name for external action }
define pdf_action_new_window(#) ≡ link(# + 1) { open a new window? }
define pdf_action_page_tokens(#) ≡ info(# + 2) { specification of GoTo page action }
define pdf_action_user_tokens(#) ≡ info(# + 2) { user-defined action string }
define pdf_action_refcount(#) ≡ link(# + 2) { counter of references to this action }
define pdf_action_struct_id(#) ≡ link(# + 3) { structure destination identifier }

{ data structure of outlines; it's not able to write out outline entries before all outline entries
  are defined, so memory allocated for outline entries can't not be deallocated and will stay in
  memory. For this reason we will store data of outline entries in pdf_mem instead of mem }
define pdfmem_outline_size ≡ 8 { size of memory in pdf_mem which obj_outline_ptr points to }
define obj_outline_count ≡ obj_info { count of all opened children }
define obj_outline_ptr ≡ obj_aux { pointer to pdf_mem }
define obj_outline_title(#) ≡ pdf_mem[obj_outline_ptr(#)]
define obj_outline_parent(#) ≡ pdf_mem[obj_outline_ptr(#) + 1]
define obj_outline_prev(#) ≡ pdf_mem[obj_outline_ptr(#) + 2]
define obj_outline_next(#) ≡ pdf_mem[obj_outline_ptr(#) + 3]
define obj_outline_first(#) ≡ pdf_mem[obj_outline_ptr(#) + 4]
define obj_outline_last(#) ≡ pdf_mem[obj_outline_ptr(#) + 5]
define obj_outline_action_objnum(#) ≡ pdf_mem[obj_outline_ptr(#) + 6] { object number of action }
define obj_outline_attr(#) ≡ pdf_mem[obj_outline_ptr(#) + 7]

{ types of destinations }
define pdf_dest_xyz ≡ 0
define pdf_dest_fit ≡ 1
define pdf_dest_fith ≡ 2
define pdf_dest_fitv ≡ 3
define pdf_dest_fitb ≡ 4

```

```

define pdf_dest_fitbh ≡ 5
define pdf_dest_fitbv ≡ 6
define pdf_dest_fitr ≡ 7
    { data structure of structure and regular destinations }
define obj_dest_ptr ≡ obj_aux { pointer to pdf_dest_node }
define pdf_dest_node_size ≡ 7
    { size of whatsit node for destination; words 1 .. 4 hold dest dimensions, word 6 identifier
      type, subtype and identifier of destination, word 6 the corresponding object number }
define pdf_dest_type(#) ≡ type(# + 5) { type of destination }
define pdf_dest_named_id(#) ≡ subtype(# + 5) { is named identifier? }
define pdf_dest_id(#) ≡ link(# + 5) { destination identifier }
define pdf_dest_xyz_zoom(#) ≡ info(# + 6) { zoom factor for destxyz destination }
define pdf_dest_objnum(#) ≡ link(# + 6) { object number of corresponding object }
    { data structure of threads; words 1..4 represent the coordinates of the corners }
define pdf_thread_node_size ≡ 7
define pdf_thread_named_id(#) ≡ subtype(# + 5) { is a named identifier }
define pdf_thread_id(#) ≡ link(# + 5) { thread identifier }
define pdf_thread_attr(#) ≡ info(# + 6) { attributes of thread }
define obj_thread_first ≡ obj_aux { pointer to the first bead }
    { data structure of beads }
define pdftmem_bead_size ≡ 5 { size of memory in pdf_mem which obj_bead_ptr points to }
define obj_bead_ptr ≡ obj_aux { pointer to pdf_mem }
define obj_bead_rect(#) ≡ pdf_mem[obj_bead_ptr(#)]
define obj_bead_page(#) ≡ pdf_mem[obj_bead_ptr(#) + 1]
define obj_bead_next(#) ≡ pdf_mem[obj_bead_ptr(#) + 2]
define obj_bead_prev(#) ≡ pdf_mem[obj_bead_ptr(#) + 3]
define obj_bead_attr(#) ≡ pdf_mem[obj_bead_ptr(#) + 4]
define obj_bead_data ≡ obj_bead_rect { pointer to the corresponding whatsit node; obj_bead_rect is
      needed only when the bead rectangle has been written out and after that obj_bead_data is not
      needed any more so we can use this field for both }
    { data structure of snap node }
define snap_node_size ≡ 3
define snap_glue_ptr(#) ≡ info(# + 1)
define final_skip(#) ≡ mem[# + 2].sc { the amount to skip }
    { data structure of snap compensation node }
define snapy_comp_ratio(#) ≡ mem[# + 1].int

⟨ Constants in the outer block 11 ⟩ +≡
inf_obj_tab_size = 1000; { min size of the cross-reference table for PDF output }
sup_obj_tab_size = 8388607; { max size of the cross-reference table for PDF output }
inf_dest_names_size = 1000; { min size of the destination names table for PDF output }
sup_dest_names_size = 500000; { max size of the destination names table for PDF output }
inf_pk_dpi = 72; { min PK pixel density value from texmf.cnf }
sup_pk_dpi = 8000; { max PK pixel density value from texmf.cnf }
pdf_objtype_max = head_tab_max;

```

696. $\langle \text{Global variables } 13 \rangle + \equiv$

`obj_tab_size: integer;`
`obj_tab: $\uparrow obj_entry$;`
`head_tab: array [1 .. head_tab_max] of integer;`
`pages_tail: integer;`
`obj_ptr: integer; { user objects counter }`
`sys_obj_ptr: integer; { system objects counter, including object streams }`
`pdf_last_pages: integer; { pointer to most recently generated pages object }`
`pdf_last_page: integer; { pointer to most recently generated page object }`
`pdf_last_stream: integer; { pointer to most recently generated stream }`
`pdf_stream_length: longinteger; { length of most recently generated stream }`
`pdf_stream_length_offset: longinteger; { file offset of the last stream length }`
`pdf_seek_write_length: boolean; { flag whether to seek back and write /Length }`
`pdf_last_byte: eight_bits; { byte most recently written to PDF file; for endstream in new line }`
`pdf_append_list_arg: integer; { for use with pdf_append_list }`
`ff: integer; { for use with set_ff }`
`pdf_box_spec_media: integer;`
`pdf_box_spec_crop: integer;`
`pdf_box_spec_bleed: integer;`
`pdf_box_spec_trim: integer;`
`pdf_box_spec_art: integer;`

697. $\langle \text{Set initial values of key variables } 21 \rangle + \equiv$

`obj_ptr \leftarrow 0; sys_obj_ptr \leftarrow 0; obj_tab_size \leftarrow inf_obj_tab_size; { allocated size of obj_tab array }`
`dest_names_size \leftarrow inf_dest_names_size; { allocated size of dest_names array }`
`for $k \leftarrow 1$ to head.tab_max do head.tab[k] \leftarrow 0;`
`pdf_box_spec_media \leftarrow 1; pdf_box_spec_crop \leftarrow 2; pdf_box_spec_bleed \leftarrow 3; pdf_box_spec_trim \leftarrow 4;`
`pdf_box_spec_art \leftarrow 5; pdf_dummy_font \leftarrow null_font;`

698. Here we implement subroutines for work with objects and related things. Some of them are used in former parts too, so we need to declare them forward.

```

define pdf_append_list_end(#+) ≡ # ← append_ptr(#+, pdf_append_list_arg);
      end
define pdf_append_list(#+)
      begin pdf_append_list_arg ← #; pdf_append_list_end
define set_ff(#+)
      begin if pdf_font_num[#+] < 0 then ff ← -pdf_font_num[#+]
            else ff ← #;
            end
< Declare procedures that need to be declared forward for pdfTeX 686 > +≡
procedure append_dest_name(s : str_number; n : integer);
  var a: integer;
  begin if pdf_dest_names_ptr = sup_dest_names_size then
        overflow("number_of_destination_names_(dest_names_size)", dest_names_size);
  if pdf_dest_names_ptr = dest_names_size then
    begin a ← 0.2 * dest_names_size;
    if dest_names_size < sup_dest_names_size - a then dest_names_size ← dest_names_size + a
    else dest_names_size ← sup_dest_names_size;
    dest_names ← xrealloc_array(dest_names, dest_name_entry, dest_names_size);
    end;
  dest_names[pdf_dest_names_ptr].objname ← s; dest_names[pdf_dest_names_ptr].objnum ← n;
  incr(pdf_dest_names_ptr);
  end;
procedure pdf_create_obj(t, i : integer); { create an object with type t and identifier i }
  label done;
  var a, p, q: integer;
  begin if sys_obj_ptr = sup_obj_tab_size then overflow("indirect_objects_table_size", obj_tab_size);
  if sys_obj_ptr = obj_tab_size then
    begin a ← 0.2 * obj_tab_size;
    if obj_tab_size < sup_obj_tab_size - a then obj_tab_size ← obj_tab_size + a
    else obj_tab_size ← sup_obj_tab_size;
    obj_tab ← xrealloc_array(obj_tab, obj_entry, obj_tab_size);
    end;
  incr(sys_obj_ptr); obj_ptr ← sys_obj_ptr; obj_info(obj_ptr) ← i; set_obj_fresh(obj_ptr);
  obj_aux(obj_ptr) ← 0; avl_put_obj(obj_ptr, t);
  if t = obj_type_page then
    begin p ← head_tab[t]; { find the right position to insert newly created object }
    if (p = 0) ∨ (obj_info(p) < i) then
      begin obj_link(obj_ptr) ← p; head_tab[t] ← obj_ptr;
      end
    else begin while p ≠ 0 do
          begin if obj_info(p) < i then goto done;
          q ← p; p ← obj_link(p);
          end;
    done: obj_link(q) ← obj_ptr; obj_link(obj_ptr) ← p;
    end;
  end
  else if t ≠ obj_type_others then
    begin obj_link(obj_ptr) ← head_tab[t]; head_tab[t] ← obj_ptr;
    if (t = obj_type_dest) ∧ (i < 0) then append_dest_name(-obj_info(obj_ptr), obj_ptr);
    end;

```

```

end;
function pdf_new_objnum: integer; { create a new object and return its number }
begin pdf_create_obj(obj_type_others, 0); pdf_new_objnum ← obj_ptr;
end;
procedure pdf_os_switch(pdf_os : boolean); { switch between PDF stream and object stream mode }
begin if pdf_os ∧ pdf_os_enable then
begin if ¬pdf_os_mode then
begin { back up PDF stream variables }
pdf_op_ptr ← pdf_ptr; pdf_ptr ← pdf_os_ptr; pdf_buf ← pdf_os_buf; pdf_buf_size ← pdf_os_buf_size;
pdf_os_mode ← true; { switch to object stream }
end;
end
else begin if pdf_os_mode then
begin { back up object stream variables }
pdf_os_ptr ← pdf_ptr; pdf_ptr ← pdf_op_ptr; pdf_buf ← pdf_op_buf; pdf_buf_size ← pdf_op_buf_size;
pdf_os_mode ← false; { switch to PDF stream }
end;
end;
end;
procedure pdf_os_prepare_obj(i : integer; pdf_os_level : integer);
{ create new /ObjStm object if required, and set up cross reference info }
begin pdf_os_switch((pdf_os_level > 0) ∧ (fixed_pdf_objcompresslevel ≥ pdf_os_level));
if pdf_os_mode then
begin if pdf_os_cur_objnum = 0 then
begin pdf_os_cur_objnum ← pdf_new_objnum; decr(obj_ptr);
{ object stream is not accessible to user }
incr(pdf_os_cntr); { only for statistics }
pdf_os_objidx ← 0; pdf_ptr ← 0; { start fresh object stream }
end
else incr(pdf_os_objidx);
obj_os_idx(i) ← pdf_os_objidx; obj_offset(i) ← pdf_os_cur_objnum; pdf_os_objnum[pdf_os_objidx] ← i;
pdf_os_objoff[pdf_os_objidx] ← pdf_ptr;
end
else begin obj_offset(i) ← pdf_offset; obj_os_idx(i) ← -1; { mark it as not included in object stream }
end;
end;
procedure pdf_begin_obj(i : integer; pdf_os_level : integer); { begin a PDF object }
begin check_pdfversion; pdf_os_prepare_obj(i, pdf_os_level);
if ¬pdf_os_mode then
begin pdf_print_int(i); pdf_print_ln(" „0 „obj");
end
else if pdf_compress_level = 0 then
begin pdf_print("% „"); { debugging help }
pdf_print_int(i); pdf_print_ln(" „0 „obj");
end;
end;
procedure pdf_new_obj(t, i : integer; pdf_os : integer); { begin a new PDF object }
begin pdf_create_obj(t, i); pdf_begin_obj(obj_ptr, pdf_os);
end;
procedure pdf_end_obj; { end a PDF object }
begin if pdf_os_mode then
begin if pdf_os_objidx = pdf_os_max_objs - 1 then pdf_os_write_objstream;

```

```
    end
else pdf_print_ln("endobj"); { end a PDF object }
end;
procedure pdf_begin_dict(i : integer; pdf_os_level : integer); { begin a PDF dictionary object }
begin check_pdfversion; pdf_os_prepare_obj(i, pdf_os_level);
if ¬pdf_os_mode then
begin pdf_print_int(i); pdf_print_ln("0obj");
end
else if pdf_compress_level = 0 then
begin pdf.print("%");
{ debugging help }
pdf.print_int(i); pdf.print_ln("0obj");
end;
pdf.print_ln("<<");
end;
procedure pdf_new_dict(t, i : integer; pdf_os : integer); { begin a new PDF dictionary object }
begin pdf_create_obj(t, i); pdf_begin_dict(obj_ptr, pdf_os);
end;
procedure pdf_end_dict; { end a PDF dictionary object }
begin if pdf_os_mode then
begin pdf.print_ln(">>");
if pdf_os_objidx = pdf_os_max_objs - 1 then pdf_os_write_objstream;
end
else begin pdf.print_ln(">>"); pdf.print_ln("endobj");
end;
end;
```

699. Write out an accumulated object stream. First the object number and byte offset pairs are generated and appended to the ready buffered object stream. By this the value of /First can be calculated. Then a new /ObjStm object is generated, and everything is copied to the PDF output buffer, where also compression is done. When calling this procedure, *pdf_os_mode* must be *true*.

```
< Declare procedures that need to be declared forward for pdfTeX 686 > +≡
procedure pdf_os_write_objstream;
  var i, j, p, q: pointer;
begin if pdf_os_cur_objnum = 0 then { no object stream started }
  return;
  p ← pdf_ptr; i ← 0; j ← 0;
  while i ≤ pdf_os_objidx do
    begin { assemble object number and byte offset pairs }
    pdf_print_int(pdf_os_objnum[i]); pdf_print(" "); pdf_print_int(pdf_os_objoff[i]);
    if j = 9 then
      begin { print out in groups of ten for better readability }
      pdf_out(pdf_new_line_char); j ← 0;
      end
    else begin pdf_print(" "); incr(j);
    end;
    incr(i);
    end;
  pdf_buf[pdf_ptr - 1] ← pdf_new_line_char; { no risk of flush, as we are in pdf_os_mode }
  q ← pdf_ptr; pdf_begin_dict(pdf_os_cur_objnum, 0); { switch to PDF stream writing }
  pdf_print_ln("/Type /ObjStm"); pdf_print("/N "); pdf_print_int_ln(pdf_os_objidx + 1);
  pdf_print("/First "); pdf_print_int_ln(q - p); pdf_begin_stream; pdf_room(q - p);
  { should always fit into the PDF output buffer }
  i ← p;
  while i < q do
    begin { write object number and byte offset pairs }
    pdf_quick_out(pdf_os_buf[i]); incr(i);
    end;
  i ← 0;
  while i < p do
    begin q ← i + pdf_buf_size;
    if q > p then q ← p;
    pdf_room(q - i);
    while i < q do
      begin { write the buffered objects }
      pdf_quick_out(pdf_os_buf[i]); incr(i);
      end;
    end;
  pdf_end_stream; pdf_os_cur_objnum ← 0; { to force object stream generation next time }
end;
```

700. ⟨ Declare procedures that need to be declared forward for pdfTeX 686 ⟩ +≡

```

function append_ptr(p : pointer; i : integer) : pointer;
    { appends a pointer with info i to the end of linked list with head p }
    var q : pointer;
    begin append_ptr ← p; fast_get_avail(q); info(q) ← i; link(q) ← null;
    if p = null then
        begin append_ptr ← q; return;
        end;
    while link(p) ≠ null do p ← link(p);
    link(p) ← q;
    end;
function pdf_lookup_list(p : pointer; i : integer) : pointer; { looks up for pointer with info i in list p }
    begin pdf_lookup_list ← null;
    while p ≠ null do
        begin if info(p) = i then
            begin pdf_lookup_list ← p; return;
            end;
        p ← link(p);
        end;
    end;

```

701. ⟨ Global variables 13 ⟩ +≡

pdf_image_procset: integer; { collection of image types used in current page/form }
pdf_text_procset: boolean; { mask of used ProcSet's in the current page/form }

702. Subroutines to print out various PDF objects:

```

define is_hex_char(#)  $\equiv$  (((#  $\geq$  '0')  $\wedge$  (#  $\leq$  '9'))  $\vee$  ((#  $\geq$  'A')  $\wedge$  (#  $\leq$  'F')))  $\vee$  ((#  $\geq$  'a')  $\wedge$  (#  $\leq$  'f')))

procedure pdf_print_fw_int(n : longinteger; w : integer);
    { print out an integer with fixed width; used for outputting cross-reference table }
    var k: integer; { 0  $\leq$  k  $\leq$  23 }
    begin k  $\leftarrow$  0;
    repeat dig[k]  $\leftarrow$  n mod 10; n  $\leftarrow$  n div 10; incr(k);
    until k = w;
    pdf_room(k);
    while k > 0 do
        begin decr(k); pdf_quick_out("0" + dig[k]);
        end;
    end;
procedure pdf_out_bytes(n : longinteger; w : integer);
    { print out an integer as a number of bytes; used for outputting /XRef cross-reference stream }
    var k: integer; byte: array [0 .. 7] of integer; { digits in a number being output }
    begin k  $\leftarrow$  0;
    repeat byte[k]  $\leftarrow$  n mod 256; n  $\leftarrow$  n div 256; incr(k);
    until k = w;
    pdf_room(k);
    while k > 0 do
        begin decr(k); pdf_quick_out(byte[k]);
        end;
    end;
procedure pdf_int_entry(s : str_number; v : integer);
    { print out an entry in dictionary with integer value to PDF buffer }
    begin pdf_out("/"); pdf_print(s); pdf_out(" "); pdf_print_int(v);
    end;
procedure pdf_int_entry_ln(s : str_number; v : integer);
    begin pdf_int_entry(s, v); pdf_print_nl();
    end;
procedure pdf_indirect(s : str_number; o : integer); { print out an indirect entry in dictionary }
    begin pdf_out("/"); pdf_print(s); pdf_out(" "); pdf_print_int(o); pdf_print(" 0 R");
    end;
procedure pdf_indirect_ln(s : str_number; o : integer);
    begin pdf_indirect(s, o); pdf_print_nl();
    end;
procedure pdf_print_str(s : str_number); { print out s as string in PDF output }
    label done;
    var i, j: pool_pointer; is_hex_string: boolean;
    begin i  $\leftarrow$  str_start[s]; j  $\leftarrow$  i + length(s) - 1;
    if i > j then
        begin pdf_print("()"); { null string }
        return;
        end;
    if (str_pool[i] = '(')  $\wedge$  (str_pool[j] = ')') then
        begin pdf_print(s); return;
        end;
    is_hex_string  $\leftarrow$  false;
    if (str_pool[i]  $\neq$  '<')  $\vee$  (str_pool[j]  $\neq$  '>')  $\vee$  odd(length(s)) then goto done;
    incr(i); decr(j);
    while i < j do

```

```
begin if is_hex_char(str_pool[i]) ∧ is_hex_char(str_pool[i + 1]) then i ← i + 2
else goto done;
end;
is_hex_string ← true;
done: if is_hex_string then pdf_print(s)
else begin pdf_out("("); pdf_print(s); pdf_out(")");
end;
end;
procedure pdf_print_str_ln(s : str_number); { print out s as string in PDF output }
begin pdf_print_str(s); pdf_print_nl;
end;
procedure pdf_str_entry(s, v : str_number);
{ print out an entry in dictionary with string value to PDF buffer }
begin if v = 0 then return;
pdf_out("/"); pdf_print(s); pdf_out(" "); pdf_print_str(v);
end;
procedure pdf_str_entry_ln(s, v : str_number);
begin if v = 0 then return;
pdf_str_entry(s, v); pdf_print_nl;
end;
```

703. Font processing. As pdfTEX should also act as a back-end driver, it needs to support virtual fonts too. Information about virtual fonts can be found in the source of some DVI-related programs.

Whenever we want to write out a character in a font to PDF output, we should check whether the used font is a new (has not been used yet), virtual or real font. The array *pdf_font_type* holds a flag of each used font. After initialization the flag of each font is set to *new_font_type*. The first time a character of a font is written out, pdfTEX looks for the corresponding virtual font. If the corresponding virtual font exists, then the font type is set to *virtual_font_type*; otherwise it will be set to *real_font_type*. *subst_font_type* indicates fonts that have been substituted during adjusting spacing. Such fonts are linked via the *pdf_font_elink* array.

```
define new_font_type = 0 { new font (has not been used yet) }
define virtual_font_type = 1 { virtual font }
define real_font_type = 2 { real font }
define subst_font_type = 3 { substituted font }

⟨ Declare procedures that need to be declared forward for pdfTEX 686 ⟩ +≡
procedure pdf_check_vf_cur_val; forward;
procedure pdf_init_font_cur_val; forward;
procedure scan_pdf_ext_toks; forward;
```

704. ⟨ Global variables 13 ⟩ +≡

```
pdf_font_type: ↑eight_bits; { the type of font }
pdf_font_attr: ↑str_number; { pointer to additional attributes }
pdf_font_nobuiltin_tounicode: ↑boolean; { disable generating ToUnicode for this font? }
```

705. Here come some subroutines to deal with expanded fonts for HZ-algorithm.

```

define set_char_and_font (#) ≡
  if is_char_node (#) then
    begin c ← character (#); f ← font (#);
    end
  else if type (#) = ligature_node then
    begin c ← character (lig_char (#)); f ← font (lig_char (#));
    end
define non_existent_path ≡ "///..."
procedure set_tag_code (f : internal_font_number; c : eight_bits; i : integer);
  var fixedi: integer;
  begin if is_valid_char (c) then
    begin fixedi ← abs (fix_int (i, -7, 0));
    if fixedi ≥ 4 then
      begin if char_tag (char_info (f)(c)) = ext_tag then
        op_byte (char_info (f)(c)) ← (op_byte (char_info (f)(c))) - ext_tag;
        fixedi ← fixedi - 4;
      end;
    if fixedi ≥ 2 then
      begin if char_tag (char_info (f)(c)) = list_tag then
        op_byte (char_info (f)(c)) ← (op_byte (char_info (f)(c))) - list_tag;
        fixedi ← fixedi - 2;
      end;
    if fixedi ≥ 1 then
      begin if char_tag (char_info (f)(c)) = lig_tag then
        op_byte (char_info (f)(c)) ← (op_byte (char_info (f)(c))) - lig_tag;
      end;
    end;
  end;
procedure set_no_ligatures (f : internal_font_number);
  var c: integer;
  begin for c ← font_bc [f] to font_ec [f] do
    if char_exists (orig_char_info (f)(c)) then
      if char_tag (orig_char_info (f)(c)) = lig_tag then
        op_byte (orig_char_info (f)(c)) ← (op_byte (orig_char_info (f)(c))) - lig_tag;
    end;
  function init_font_base (v : integer): integer;
  var i, j: integer;
  begin i ← pdf_get_mem (256);
  for j ← 0 to 255 do pdf_mem [i + j] ← v;
  init_font_base ← i;
  end;
procedure set_lp_code (f : internal_font_number; c : eight_bits; i : integer);
  begin if pdf_font_lp_base [f] = 0 then pdf_font_lp_base [f] ← init_font_base (0);
  pdf_mem [pdf_font_lp_base [f] + c] ← fix_int (i, -1000, 1000);
  end;
procedure set_rp_code (f : internal_font_number; c : eight_bits; i : integer);
  begin if pdf_font_rp_base [f] = 0 then pdf_font_rp_base [f] ← init_font_base (0);
  pdf_mem [pdf_font_rp_base [f] + c] ← fix_int (i, -1000, 1000);
  end;
procedure set_ef_code (f : internal_font_number; c : eight_bits; i : integer);
  begin if pdf_font_ef_base [f] = 0 then pdf_font_ef_base [f] ← init_font_base (1000);

```

```

pdf_mem[pdf_font_ef_base[f] + c] ← fix_int(i, 0, 1000);
end;

procedure set_kn_bs_code(f : internal_font_number; c : eight_bits; i : integer);
begin if pdf_font_kn_bs_base[f] = 0 then pdf_font_kn_bs_base[f] ← init_font_base(0);
pdf_mem[pdf_font_kn_bs_base[f] + c] ← fix_int(i, -1000, 1000);
end;

procedure set_st_bs_code(f : internal_font_number; c : eight_bits; i : integer);
begin if pdf_font_st_bs_base[f] = 0 then pdf_font_st_bs_base[f] ← init_font_base(0);
pdf_mem[pdf_font_st_bs_base[f] + c] ← fix_int(i, -1000, 1000);
end;

procedure set_sh_bs_code(f : internal_font_number; c : eight_bits; i : integer);
begin if pdf_font_sh_bs_base[f] = 0 then pdf_font_sh_bs_base[f] ← init_font_base(0);
pdf_mem[pdf_font_sh_bs_base[f] + c] ← fix_int(i, -1000, 1000);
end;

procedure set_kn_bc_code(f : internal_font_number; c : eight_bits; i : integer);
begin if pdf_font_kn_bc_base[f] = 0 then pdf_font_kn_bc_base[f] ← init_font_base(0);
pdf_mem[pdf_font_kn_bc_base[f] + c] ← fix_int(i, -1000, 1000);
end;

procedure set_kn_ac_code(f : internal_font_number; c : eight_bits; i : integer);
begin if pdf_font_kn_ac_base[f] = 0 then pdf_font_kn_ac_base[f] ← init_font_base(0);
pdf_mem[pdf_font_kn_ac_base[f] + c] ← fix_int(i, -1000, 1000);
end;

procedure adjust_interword_glue(p, g : pointer); { adjust the interword glue g after a character p }
var kn, st, sh: scaled; q, r: pointer; c: halfword; f: internal_font_number;
begin if ¬(¬is_char_node(g) ∧ type(g) = glue_node) then
begin pdf_warning("adjust_interword_glue", "g is not a glue", true, true); return;
end;
c ← non_char; { no char before interword glue yet }
set_char_and_font(p) { set f and c if p is a char or ligature }
else if (type(p) = kern_node) ∧ (subtype(p) = auto_kern) ∧ (save_tail ≠ null) then
begin r ← save_tail;
while (link(r) ≠ null) ∧ (link(r) ≠ p) do r ← link(r);
if (link(r) = p) then set_char_and_font(r); { set f and c if r is a char or ligature }
end;
if (c = non_char) then return;
kn ← get_kn_bs_code(f, c); st ← get_st_bs_code(f, c); sh ← get_sh_bs_code(f, c);
if (kn ≠ 0) ∨ (st ≠ 0) ∨ (sh ≠ 0) then
begin q ← new_spec(glue_ptr(g)); delete_glue_ref(glue_ptr(g));
width(q) ← width(q) + round_xn_over_d(quad(f), kn, 1000);
stretch(q) ← stretch(q) + round_xn_over_d(quad(f), st, 1000);
shrink(q) ← shrink(q) + round_xn_over_d(quad(f), sh, 1000); glue_ptr(g) ← q;
end;
end;

function get_auto_kern(f : internal_font_number; l, r : halfword): pointer;
{ return a pointer to an auto kern node, or null }
var tmp_w: scaled; k: integer; p: pointer;
begin pdffassert((l ≥ 0) ∧ (r ≥ 0)); get_auto_kern ← null;
if (pdf_append_kern ≤ 0) ∧ (pdf_prepend_kern ≤ 0) then return;
tmp_w ← 0;
if (pdf_append_kern > 0) ∧ (l < non_char) then
begin k ← get_kn_ac_code(f, l);
if k ≠ 0 then tmp_w ← round_xn_over_d(quad(f), k, 1000);

```

```

end;
if (pdf_prepended_kern > 0) ∧ (r < non_char) then
begin k ← get_kn_bc_code(f, r);
if k ≠ 0 then tmp_w ← tmp_w + round_xn_over_d(quad(f), k, 1000);
end;
if tmp_w ≠ 0 then
begin p ← new_kern(tmp_w); subtype(p) ← auto_kern; get_auto_kern ← p;
end;
end;

function expand_font_name(f : internal_font_number; e : integer): str_number;
var old_setting: 0 .. max_selector; { holds selector setting }
begin old_setting ← selector; selector ← new_string; print(font_name[f]);
if e > 0 then print("+" ); { minus sign will be printed by print_int }
print_int(e); selector ← old_setting; expand_font_name ← make_string;
end;

function auto_expand_font(f : internal_font_number; e : integer): internal_font_number;
{ creates an expanded font from the base font; doesn't load expanded tfm at all }
var k: internal_font_number; nw, nk, ni, i: integer;
begin k ← font_ptr + 1; incr(font_ptr);
if (font_ptr ≥ font_max) then overflow("maximum_internal_font_number(font_max)", font_max);
font_name[k] ← expand_font_name(f, e); font_area[k] ← font_area[f]; font_id_text(k) ← font_id_text(f);
hyphen_char[k] ← hyphen_char[f]; skew_char[k] ← skew_char[f]; font_bchar[k] ← font_bchar[f];
font_false_bchar[k] ← font_false_bchar[f]; font_bc[k] ← font_bc[f]; font_ec[k] ← font_ec[f];
font_size[k] ← font_size[f]; font_dsize[k] ← font_dsize[f]; font_params[k] ← font_params[f];
font_glue[k] ← font_glue[f]; bchar_label[k] ← bchar_label[f]; char_base[k] ← char_base[f];
height_base[k] ← height_base[f]; depth_base[k] ← depth_base[f]; lig_kern_base[k] ← lig_kern_base[f];
exten_base[k] ← exten_base[f]; param_base[k] ← param_base[f]; nw ← height_base[f] - width_base[f];
ni ← lig_kern_base[f] - italic_base[f]; nk ← exten_base[f] - (kern_base[f] + kern_base_offset);
if (fmem_ptr + nw + ni + nk ≥ font_mem_size) then
overflow("number_of_words_of_font_memory(font_mem_size)", font_mem_size);
width_base[k] ← fmem_ptr; italic_base[k] ← width_base[k] + nw;
kern_base[k] ← italic_base[k] + ni - kern_base_offset; fmem_ptr ← fmem_ptr + nw + ni + nk;
for i ← 0 to nw - 1 do
font_info[width_base[k] + i].sc ← round_xn_over_d(font_info[width_base[f] + i].sc, 1000 + e, 1000);
for i ← 0 to ni - 1 do
font_info[italic_base[k] + i].sc ← round_xn_over_d(font_info[italic_base[f] + i].sc, 1000 + e, 1000);
for i ← 0 to nk - 1 do font_info[kern_base[k] + kern_base_offset + i].sc ←
round_xn_over_d(font_info[kern_base[f] + kern_base_offset + i].sc, 1000 + e, 1000);
auto_expand_font ← k;
end;

procedure copy_expand_params(k, f : internal_font_number; e : integer);
{ set expansion-related parameters for an expanded font k, based on the base font f and the
expansion amount e }
begin if pdf_font_rp_base[f] = 0 then pdf_font_rp_base[f] ← init_font_base(0);
if pdf_font_lp_base[f] = 0 then pdf_font_lp_base[f] ← init_font_base(0);
if pdf_font_ef_base[f] = 0 then pdf_font_ef_base[f] ← init_font_base(1000);
pdf_font_expand_ratio[k] ← e; pdf_font_step[k] ← pdf_font_step[f];
pdf_font_auto_expand[k] ← pdf_font_auto_expand[f]; pdf_font_blink[k] ← f;
pdf_font_lp_base[k] ← pdf_font_lp_base[f]; pdf_font_rp_base[k] ← pdf_font_rp_base[f];
pdf_font_ef_base[k] ← pdf_font_ef_base[f];
if pdf_font_kn_bs_base[f] = 0 then pdf_font_kn_bs_base[f] ← init_font_base(0);
if pdf_font_st_bs_base[f] = 0 then pdf_font_st_bs_base[f] ← init_font_base(0);

```

```

if  $pdf\_font\_sh\_bs\_base[f] = 0$  then  $pdf\_font\_sh\_bs\_base[f] \leftarrow init\_font\_base(0)$ ;
if  $pdf\_font\_kn\_bc\_base[f] = 0$  then  $pdf\_font\_kn\_bc\_base[f] \leftarrow init\_font\_base(0)$ ;
if  $pdf\_font\_kn\_ac\_base[f] = 0$  then  $pdf\_font\_kn\_ac\_base[f] \leftarrow init\_font\_base(0)$ ;
 $pdf\_font\_kn\_bs\_base[k] \leftarrow pdf\_font\_kn\_bs\_base[f]$ ;  $pdf\_font\_st\_bs\_base[k] \leftarrow pdf\_font\_st\_bs\_base[f]$ ;
 $pdf\_font\_sh\_bs\_base[k] \leftarrow pdf\_font\_sh\_bs\_base[f]$ ;  $pdf\_font\_kn\_bc\_base[k] \leftarrow pdf\_font\_kn\_bc\_base[f]$ ;
 $pdf\_font\_kn\_ac\_base[k] \leftarrow pdf\_font\_kn\_ac\_base[f]$ ;
end;
function  $tfm\_lookup(s : str\_number; fs : scaled) : internal\_font\_number$ ;
    { looks up for a TFM with name  $s$  loaded at  $fs$  size; if found then flushes  $s$  }
var  $k : internal\_font\_number$ ;
begin if  $fs \neq 0$  then
    begin for  $k \leftarrow font\_base + 1$  to  $font\_ptr$  do
        if ( $font\_area[k] \neq non\_existent\_path$ )  $\wedge str\_eq\_str(font\_name[k], s) \wedge (font\_size[k] = fs)$  then
            begin  $flush\_str(s)$ ;  $tfm\_lookup \leftarrow k$ ; return;
            end;
        end
    else begin for  $k \leftarrow font\_base + 1$  to  $font\_ptr$  do
        if ( $font\_area[k] \neq non\_existent\_path$ )  $\wedge str\_eq\_str(font\_name[k], s)$  then
            begin  $flush\_str(s)$ ;  $tfm\_lookup \leftarrow k$ ; return;
            end;
        end;
    end;
 $tfm\_lookup \leftarrow null\_font$ ;
end;
function  $load\_expand\_font(f : internal\_font\_number; e : integer) : internal\_font\_number$ ; { loads font  $f$ 
    expanded by  $e$  thousandths into font memory;  $e$  is nonzero and is a multiple of  $pdf\_font\_step[f]$ }
label found;
var  $s : str\_number$ ; { font name }
 $k : internal\_font\_number$ ;
begin  $s \leftarrow expand\_font\_name(f, e)$ ;  $k \leftarrow tfm\_lookup(s, font\_size[f])$ ;
if  $k = null\_font$  then
    begin if  $pdf\_font\_auto\_expand[f]$  then  $k \leftarrow auto\_expand\_font(f, e)$ 
    else  $k \leftarrow read\_font\_info(null\_cs, s, "", font\_size[f])$ ;
    end;
if  $k \neq null\_font$  then  $copy\_expand\_params(k, f, e)$ ;
 $load\_expand\_font \leftarrow k$ ;
end;
function  $fix\_expand\_value(f : internal\_font\_number; e : integer) : integer$ ;
    { return the multiple of  $pdf\_font\_step[f]$  that is nearest to  $e$  }
var  $step : integer$ ;  $max\_expand : integer$ ;  $neg : boolean$ ;
begin  $fix\_expand\_value \leftarrow 0$ ;
if  $e = 0$  then return;
if  $e < 0$  then
    begin  $e \leftarrow -e$ ;  $neg \leftarrow true$ ;  $max\_expand \leftarrow -pdf\_font\_expand\_ratio[pdf\_font\_shrink[f]]$ ;
    end
else begin  $neg \leftarrow false$ ;  $max\_expand \leftarrow pdf\_font\_expand\_ratio[pdf\_font\_stretch[f]]$ ;
    end;
if  $e > max\_expand$  then  $e \leftarrow max\_expand$ 
else begin  $step \leftarrow pdf\_font\_step[f]$ ;
    if  $e \bmod step > 0$  then  $e \leftarrow step * round\_xn\_over\_d(e, 1, step)$ ;
    end;
if  $neg$  then  $e \leftarrow -e$ ;
 $fix\_expand\_value \leftarrow e$ ;

```

```

end;
function get_expand_font(f : internal_font_number; e : integer) : internal_font_number;
  { look up and create if not found an expanded version of f; f is an expandable font; e is nonzero
    and is a multiple of pdf_font_step[f] }
var k : internal_font_number;
begin k ← pdf_font_elink[f];
while k ≠ null_font do
  begin if pdf_font_expand_ratio[k] = e then
    begin get_expand_font ← k; return;
    end;
    k ← pdf_font_elink[k];
  end;
  k ← load_expand_font(f, e); pdf_font_elink[k] ← pdf_font_elink[f]; pdf_font_elink[f] ← k;
  get_expand_font ← k;
  end;
function expand_font(f : internal_font_number; e : integer) : internal_font_number;
  { looks up for font f expanded by e thousandths, e is an arbitrary value between max stretch and
    max shrink of f; if not found then creates it }
begin expand_font ← f;
if e = 0 then return;
e ← fix_expand_value(f, e);
if e = 0 then return;
if pdf_font_elink[f] = null_font then pdf_error("font_expansion", "uninitialized_pdf_font_elink");
  expand_font ← get_expand_font(f, e);
end;
procedure set_expand_params(f : internal_font_number; auto_expand : boolean;
  stretch_limit, shrink_limit, font_step, expand_ratio : integer);
  { expand a font with given parameters }
begin pdf_font_step[f] ← font_step; pdf_font_auto_expand[f] ← auto_expand;
  if stretch_limit > 0 then pdf_font_stretch[f] ← get_expand_font(f, stretch_limit);
  if shrink_limit > 0 then pdf_font_shrink[f] ← get_expand_font(f, -shrink_limit);
  if expand_ratio ≠ 0 then pdf_font_expand_ratio[f] ← expand_ratio;
end;
procedure vf_expand_local_fonts(f : internal_font_number);
var lf : internal_font_number; k : integer;
begin pdfassert(pdf_font_type[f] = virtual_font_type);
  for k ← 0 to vf_local_font_num[f] – 1 do
    begin lf ← vf_i_fnts[vf_default_font[f] + k];
    set_expand_params(lf, pdf_font_auto_expand[f], pdf_font_expand_ratio[pdf_font_stretch[f]],
      –pdf_font_expand_ratio[pdf_font_shrink[f]], pdf_font_step[f], pdf_font_expand_ratio[f]);
    if pdf_font_type[lf] = virtual_font_type then vf_expand_local_fonts(lf);
    end;
  end;
procedure read_expand_font; { read font expansion spec and load expanded font }
var shrink_limit, stretch_limit, font_step: integer; f: internal_font_number; auto_expand: boolean;
begin { read font expansion parameters }
  scan_font_ident; f ← cur_val;
  if f = null_font then pdf_error("font_expansion", "invalid_font_identifier");
  if pdf_font_blink[f] ≠ null_font then pdf_error("font_expansion",
    "\pdffontexpand cannot be used this way (the base font has been expanded)");
  scan_optional_equals; scan_int; stretch_limit ← fix_int(cur_val, 0, 1000); scan_int;
  shrink_limit ← fix_int(cur_val, 0, 500); scan_int; font_step ← fix_int(cur_val, 0, 100);

```

```

if font_step = 0 then pdf_error("font_expansion", "invalid_step");
stretch_limit ← stretch_limit - stretch_limit mod font_step;
if stretch_limit < 0 then stretch_limit ← 0;
shrink_limit ← shrink_limit - shrink_limit mod font_step;
if shrink_limit < 0 then shrink_limit ← 0;
if (stretch_limit = 0) ∧ (shrink_limit = 0) then pdf_error("font_expansion", "invalid_limit(s)");
auto_expand ← false;
if scan_keyword("autoexpand") then
begin auto_expand ← true; { Scan an optional space 469 };
end; { check if the font can be expanded }
if (pdf_font_expand_ratio[f] ≠ 0) then pdf_error("font_expansion",
"this_font_has_been_expanded_by_another_font_so_it_cannot_be_used_now");
if (pdf_font_step[f] ≠ 0) then
{ this font has been expanded, ensure the expansion parameters are identical }
begin if pdf_font_step[f] ≠ font_step then
pdf_error("font_expansion", "font_has_been_expanded_with_different_expansion_step");
if ((pdf_font_stretch[f] = null_font) ∧ (stretch_limit ≠ 0)) ∨ ((pdf_font_stretch[f] ≠
null_font) ∧ (pdf_font_expand_ratio[pdf_font_stretch[f]] ≠ stretch_limit)) then
pdf_error("font_expansion", "font_has_been_expanded_with_different_stretch_limit");
if ((pdf_font_shrink[f] = null_font) ∧ (shrink_limit ≠ 0)) ∨ ((pdf_font_shrink[f] ≠
null_font) ∧ (-pdf_font_expand_ratio[pdf_font_shrink[f]] ≠ shrink_limit)) then
pdf_error("font_expansion", "font_has_been_expanded_with_different_shrink_limit");
if pdf_font_auto_expand[f] ≠ auto_expand then pdf_error("font_expansion",
"font_has_been_expanded_with_different_auto_expansion_value");
end
else begin if (pdf_font_type[f] ≠ new_font_type) ∧ (pdf_font_type[f] ≠ virtual_font_type) then
pdf_warning("font_expansion", "font_should_be_expanded_before_its_first_use", true, true);
set_expand_params(f, auto_expand, stretch_limit, shrink_limit, font_step, 0);
if pdf_font_type[f] = virtual_font_type then vf_expand_local_fonts(f);
end;
end;

```

706. We implement robust letter spacing using virtual font.

```

define vf_replace_z ≡
  begin vf_alpha ← 16;
  while vf_z ≥ '40000000 do
    begin vf_z ← vf_z div 2; vf_alpha ← vf_alpha + vf_alpha;
    end;
  vf_beta ← 256 div vf_alpha; vf_alpha ← vf_alpha * vf_z;
  end

function letter_space_font(u : pointer; f : internal_font_number; e : integer) : internal_font_number;
  var k: internal_font_number; w,r: scaled; s: str_number; i,nw: integer; old_setting: 0 .. max_selector;
  vf_z: integer; vf_alpha: integer; vf_beta: 1 .. 16;
  begin { read a new font and expand the character widths }
  k ← read_font_info(u, font_name[f], font_size[f]);
  if scan_keyword("nolig") then set_no_ligatures(k); { disable ligatures for letter-spaced fonts }
  nw ← height_base[k] - width_base[k];
  if (quad(k) = 0) ∧ (quad(f) > 0) then quad(k) ← quad(f);
  if quad(k) = 0 then
    pdf_warning("\letterspacefont", "font_has_zero_em_size(\fontdimen6)", true, true);
  for i ← 0 to nw - 1 do
    font_info[width_base[k] + i].sc ← font_info[width_base[k] + i].sc + round_xn_over_d(quad(k), e, 1000);
    { append, e.g., '+100ls' to font name }
    str_room(length(font_name[k]) + 7); { abs(e) ≤ 1000 }
    old_setting ← selector; selector ← new_string; print(font_name[k]);
    if e > 0 then print("+"); { minus sign will be printed by print_int }
    print_int(e); print("ls"); selector ← old_setting; font_name[k] ← make_string;
    { create the corresponding virtual font }
    allocvffnts; vf_e_fnts[vf_nf] ← 0; vf_i_fnts[vf_nf] ← f; incr(vf_nf); vf_local_font_num[k] ← 1;
    vf_default_font[k] ← vf_nf - 1; pdf_font_type[k] ← virtual_font_type; vf_z ← font_size[f]; vf_replace_z;
    w ← round_xn_over_d(quad(f), e, 2000);
    if w ≥ 0 then tmp_b0 ← 0
    else begin tmp_b0 ← 255; w ← vf_alpha + w;
    end;
    r ← w * vf_beta; tmp_b1 ← r div vf_z; r ← r mod vf_z;
    if r = 0 then tmp_b2 ← 0
    else begin r ← r * 256; tmp_b2 ← r div vf_z; r ← r mod vf_z;
    end;
    if r = 0 then tmp_b3 ← 0
    else begin r ← r * 256; tmp_b3 ← r div vf_z;
    end;
    vf_packet_base[k] ← new_vf_packet(k);
    for c ← font_bc[k] to font_ec[k] do
      begin str_room(12); append_char(right1 + 3); append_char(tmp_b0); append_char(tmp_b1);
      append_char(tmp_b2); append_char(tmp_b3);
      if c < set1 then append_char(c)
      else begin append_char(set1); append_char(c);
      end;
      append_char(right1 + 3); append_char(tmp_b0); append_char(tmp_b1); append_char(tmp_b2);
      append_char(tmp_b3); s ← make_string; storepacket(k, c, s); flush_str(s);
    end;
    letter_space_font ← k;
  end;

procedure new_letterspaced_font(a : small_number); { letter-space a font by creating a virtual font }

```

```

var u: pointer; { user's font identifier }
t: str_number; { name for the frozen font identifier }
old_setting: 0 .. max_selector; { holds selector setting }
f, k: internal_font_number;
begin get_r_token; u  $\leftarrow$  cur_cs;
if u  $\geq$  hash_base then t  $\leftarrow$  text(u)
else if u  $\geq$  single_base then
  if u = null_cs then t  $\leftarrow$  "FONT" else t  $\leftarrow$  u - single_base
  else begin old_setting  $\leftarrow$  selector; selector  $\leftarrow$  new_string; print("FONT"); print(u - active_base);
    selector  $\leftarrow$  old_setting; str_room(1); t  $\leftarrow$  make_string;
  end;
  define(u, set_font, null_font); scan_optional_equals; scan_font_ident; k  $\leftarrow$  cur_val; scan_int;
  f  $\leftarrow$  letter_space_font(u, k, fix_int(cur_val, -1000, 1000)); equiv(u)  $\leftarrow$  f; eqtb[font_id_base + f]  $\leftarrow$  eqtb[u];
  font_id_text(f)  $\leftarrow$  t;
  end;
function is_letterspaced_font(f : internal_font_number): boolean;
label done;
var i, j: pool_pointer;
begin is_letterspaced_font  $\leftarrow$  false;
if pdf_font_type[f]  $\neq$  virtual_font_type then return;
i  $\leftarrow$  str_start[font_name[f] + 1] - 1; j  $\leftarrow$  str_start[font_name[f]];
if (str_pool[i - 1]  $\neq$  '1')  $\vee$  (str_pool[i]  $\neq$  's') then return;
i  $\leftarrow$  i - 2;
while i  $\geq$  j do
  begin if (str_pool[i] < '0')  $\vee$  (str_pool[i] > '9') then goto done;
  i  $\leftarrow$  i - 1;
  end;
done: if i < j then return;
if (str_pool[i]  $\neq$  '+')  $\wedge$  (str_pool[i]  $\neq$  '-') then return;
  is_letterspaced_font  $\leftarrow$  true;
end;
function copy_font_info(f : internal_font_number): internal_font_number;
  { create a copy of f in the font mem }
var lf, bc, ec, i: halfword; k: internal_font_number;
begin if (pdf_font_expand_ratio[f]  $\neq$  0)  $\vee$  (pdf_font_step[f]  $\neq$  0) then
  pdf_error("\pdfcopyfont", "cannot_copy_an_expanded_font");
if is_letterspaced_font(f) then pdf_error("\pdfcopyfont", "cannot_copy_a_letterspaced_font");
k  $\leftarrow$  font_ptr + 1; incr(font_ptr);
if (font_ptr  $\geq$  font_max) then overflow("maximum_internal_font_number(font_max)", font_max);
font_name[k]  $\leftarrow$  font_name[f]; font_area[k]  $\leftarrow$  non_existent_path;
  { to avoid interferences with new_font() and tfm_lookup() }
hyphen_char[k]  $\leftarrow$  hyphen_char[f]; skew_char[k]  $\leftarrow$  skew_char[f]; font_bchar[k]  $\leftarrow$  font_bchar[f];
font_false_bchar[k]  $\leftarrow$  font_false_bchar[f]; font_bc[k]  $\leftarrow$  font_bc[f]; font_ec[k]  $\leftarrow$  font_ec[f];
font_size[k]  $\leftarrow$  font_size[f]; font_dsize[k]  $\leftarrow$  font_dsize[f]; font_params[k]  $\leftarrow$  font_params[f];
font_glue[k]  $\leftarrow$  font_glue[f]; bchar_label[k]  $\leftarrow$  bchar_label[f]; { set base addresses }
bc  $\leftarrow$  font_bc[f]; ec  $\leftarrow$  font_ec[f]; char_base[k]  $\leftarrow$  fmem_ptr - bc; width_base[k]  $\leftarrow$  char_base[k] + ec + 1;
height_base[k]  $\leftarrow$  width_base[k] + (height_base[f] - width_base[f]);
depth_base[k]  $\leftarrow$  height_base[k] + (depth_base[f] - height_base[f]);
italic_base[k]  $\leftarrow$  depth_base[k] + (italic_base[f] - depth_base[f]);
lig_kern_base[k]  $\leftarrow$  italic_base[k] + (lig_kern_base[f] - italic_base[f]);
kern_base[k]  $\leftarrow$  lig_kern_base[k] + (kern_base[f] - lig_kern_base[f]);
exten_base[k]  $\leftarrow$  kern_base[k] + (exten_base[f] - kern_base[f]);

```

```

param_base[k] ← exten_base[k] + (param_base[f] – exten_base[f]);
{ allocate memory for the new font k and copy data from f }
lf ← (param_base[f] – char_base[f]) + font_params[f] + 1;
if (fmem_ptr + lf ≥ font_mem_size) then
    overflow("number_of_words_of_font_memory(font_mem_size)", font_mem_size);
for i ← 0 to lf – 1 do font_info[char_base[k] + bc + i] ← font_info[char_base[f] + bc + i];
fmem_ptr ← fmem_ptr + lf; copy_font_info ← k;
end;
procedure make_font_copy(a : small_number); { make a font copy for further use with font expansion }
var u: pointer; { user's font identifier }
t: str_number; { name for the frozen font identifier }
old_setting: 0 .. max_selector; { holds selector setting }
f, k: internal_font_number;
begin get_r_token; u ← cur_cs;
if u ≥ hash_base then t ← text(u)
else if u ≥ single_base then
    if u = null_cs then t ← "FONT" else t ← u – single_base
    else begin old_setting ← selector; selector ← new_string; print("FONT"); print(u – active_base);
        selector ← old_setting; str_room(1); t ← make_string;
    end;
define(u, set_font, null_font); scan_optional_equals; scan_font_ident; k ← cur_val; f ← copy_font_info(k);
equiv(u) ← f; eqtb[font_id_base + f] ← eqtb[u]; font_id_text(f) ← t;
end;

```

707. We need to hold information about used characters in each font for partial downloading.

⟨ Types in the outer block 18 ⟩ +≡
 $\text{char_used_array} = \text{array } [0..31] \text{ of eight_bits};$
 $\text{char_map_array} = \text{array } [0..32] \text{ of eight_bits}; \{ \text{move chars in range } 0..32 \}$
 $\text{fm_entry_ptr} = \uparrow \text{integer};$

708. ⟨ Global variables 13 ⟩ +≡
 $\text{pdf_char_used}: \uparrow \text{char_used_array}; \{ \text{to mark used chars} \}$
 $\text{pdf_font_size}: \uparrow \text{scaled}; \{ \text{used size of font in PDF file} \}$
 $\text{pdf_font_num}: \uparrow \text{integer};$
{ mapping between internal font number in TeX and font name defined in resources in PDF file }
 $\text{pdf_font_map}: \uparrow \text{fm_entry_ptr}; \{ \text{pointer into AVL tree of font mappings} \}$
 $\text{pdf_font_list}: \text{pointer}; \{ \text{list of used fonts in current page} \}$
 $\text{pdf_resname_prefix}: \text{str_number}; \{ \text{global prefix of resources name} \}$
 $\text{last_tokens_string}: \text{str_number}; \{ \text{the number of the most recently string created by tokens_to_string} \}$

709. ⟨ Set initial values of key variables 21 ⟩ +≡
 $\text{pdf_resname_prefix} \leftarrow 0; \text{last_tokens_string} \leftarrow 0;$

710. Here we implement reading information from VF file.

```

define vf_max_packet_length = 10000 { max length of character packet in VF file }
define do_char = 70 { label to go to typesetting a character of virtual font }
define long_char = 242 { VF command for general character packet }
define vf_id = 202 { identifies VF files }
define put1 = 133 { typeset a character }
define four_cases(#) $\equiv$ #,#+1,#+2,#+3
define tmp_b0  $\equiv$  tmp_w.qqqq.b0
define tmp_b1  $\equiv$  tmp_w.qqqq.b1
define tmp_b2  $\equiv$  tmp_w.qqqq.b2
define tmp_b3  $\equiv$  tmp_w.qqqq.b3
define tmp_int  $\equiv$  tmp_w.int
define bad_vf(#) $\equiv$  vf_error(font_name[f],#) { quit with an error message telling the vf filename }

{Global variables 13} +≡
vf_packet_base: ↑integer; { base addresses of character packets from virtual fonts }
vf_default_font: ↑internal_font_number; { default font in a VF file }
vf_local_font_num: ↑internal_font_number; { number of local fonts in a VF file }
vf_packet_length: integer; { length of the current packet }
vf_file: byte_file;
vf_nf: internal_font_number; { the local fonts counter }
vf_e_fnts: ↑integer; { external font numbers }
vf_i_fnts: ↑internal_font_number; { corresponding internal font numbers }
tmp_w: memory_word; { accumulator }

```

711. { Set initial values of key variables 21 } +≡

$vf_nf \leftarrow 0;$

712. The *do_vf* procedure attempts to read the VF file for a font, and sets *pdf.font_type* to *real_font_type* if the VF file could not be found or loaded, otherwise sets *pdf.font_type* to *virtual_font_type*. To process font definitions in virtual font we call *vf_def_font*.

```

procedure vf_error(filename, msg : str_number);
  var old_setting: 0 .. max_selector; { holds print selector }
      s: str_number;
begin str_room(length(filename) + 3); old_setting ← selector; selector ← new_string; print(filename);
  print(".vf"); s ← make_string; selector ← old_setting; pdf_error(s, msg);
end;

function vf_byte: eight_bits; { read a byte from vf_file }
  var i: integer;
begin i ← getc(vf_file);
if i < 0 then pdf_error("vf", "unexpected_EOF_or_error");
  vf_byte ← i;
end;

function vf_read_signed(k : integer): integer; { read k bytes as an signed integer from VF file }
  var i: integer;
begin pdffassert((k > 0) ∧ (k ≤ 4)); i ← vf_byte;
if i ≥ 128 then i ← i - 256;
  decr(k);
while k > 0 do
  begin i ← i * 256 + vf_byte; decr(k);
  end;
  vf_read_signed ← i;
end;

function vf_read_unsigned(k : integer): integer; { read k bytes as an unsigned integer from VF file }
  var i: integer;
begin pdffassert((k > 0) ∧ (k ≤ 4)); i ← vf_byte;
if (k = 4) ∧ (i ≥ 128) then bad_vf("number_too_big");
  decr(k);
while k > 0 do
  begin i ← i * 256 + vf_byte; decr(k);
  end;
  vf_read_unsigned ← i;
end;

procedure vf_local_font_warning(f, k : internal_font_number; s : str_number);
  { print a warning message if an error occurs during processing local fonts in VF file }
begin print_nl(s); print("in_local_font"); print(font_name[k]); print("in_virtual_font");
  print(font_name[f]); print(".vf_ignored.");
end;

function vf_def_font(f : internal_font_number): internal_font_number; { process a local font in VF file }
  var k: internal_font_number; s: str_number; ds, fs: scaled; cs: four_quarters;
begin cs.b0 ← vf_byte; cs.b1 ← vf_byte; cs.b2 ← vf_byte; cs.b3 ← vf_byte;
  fs ← store_scaled_f(vf_read_signed(4), font_size[f]); ds ← vf_read_signed(4) div '20; tmp_b0 ← vf_byte;
  tmp_b1 ← vf_byte;
while tmp_b0 > 0 do
  begin decr(tmp_b0); call_func(vf_byte); { skip the font path }
  end;
  str_room(tmp_b1);
while tmp_b1 > 0 do
  begin decr(tmp_b1); append_char(vf_byte);
  end;

```

```

 $s \leftarrow make\_string; k \leftarrow tfm\_lookup(s, fs);$ 
 $\text{if } k = null\_font \text{ then } k \leftarrow read\_font\_info(null\_cs, s, "", fs);$ 
 $\text{if } k \neq null\_font \text{ then}$ 
 $\quad \text{begin if } ((cs.b0 \neq 0) \vee (cs.b1 \neq 0) \vee (cs.b2 \neq 0) \vee (cs.b3 \neq 0)) \wedge ((font\_check[k].b0 \neq 0) \vee (font\_check[k].b1 \neq 0) \vee (font\_check[k].b2 \neq 0) \vee (font\_check[k].b3 \neq 0)) \wedge ((cs.b0 \neq font\_check[k].b0) \vee (cs.b1 \neq font\_check[k].b1) \vee (cs.b2 \neq font\_check[k].b2) \vee (cs.b3 \neq font\_check[k].b3)) \text{ then } vf.local\_font\_warning(f, k, "checksum\_mismatch");$ 
 $\quad \text{if } ds \neq font\_dsizes[k] \text{ then } vf.local\_font\_warning(f, k, "design\_size\_mismatch");$ 
 $\quad \text{if } (pdf\_font\_step[f] \neq 0) \text{ then}$ 
 $\quad \quad set\_expand\_params(k, pdf\_font\_auto\_expand[f], pdf\_font\_expand\_ratio[pdf\_font\_stretch[f]],$ 
 $\quad \quad \quad -pdf\_font\_expand\_ratio[pdf\_font\_shrink[f]], pdf\_font\_step[f], pdf\_font\_expand\_ratio[f]);$ 
 $\quad \text{end};$ 
 $\quad vf\_def\_font \leftarrow k;$ 
 $\quad \text{end};$ 
 $\text{procedure } do\_vf(f : internal\_font\_number); \quad \{ \text{process VF file with font internal number } f \}$ 
 $\quad \text{var } cmd, k, n: integer; cc, cmd\_length, packet\_length: integer; tfm\_width: scaled; s: str\_number;$ 
 $\quad stack\_level: vf\_stack\_index; save\_vf\_nf: internal\_font\_number;$ 
 $\quad \text{begin } pdf\_font\_type[f] \leftarrow real\_font\_type;$ 
 $\quad \text{if } auto\_expand\_vf(f) \text{ then return; } \{ \text{auto-expanded virtual font} \}$ 
 $\quad stack\_level \leftarrow 0; \langle \text{Open vf\_file, return if not found 713} \rangle;$ 
 $\quad \langle \text{Process the preamble 714} \rangle;$ 
 $\quad \langle \text{Process the font definitions 715} \rangle;$ 
 $\quad \langle \text{Allocate memory for the new virtual font 716} \rangle;$ 
 $\quad \text{while } cmd \leq long\_char \text{ do}$ 
 $\quad \quad \text{begin } \langle \text{Build a character packet 717} \rangle;$ 
 $\quad \quad \text{end};$ 
 $\quad \text{if } cmd \neq post \text{ then } bad\_vf("POST\_command\_expected");$ 
 $\quad b\_close(vf\_file); pdf\_font\_type[f] \leftarrow virtual\_font\_type;$ 
 $\quad \text{end};$ 

```

713. $\langle \text{Open vf_file, return if not found 713} \rangle \equiv$

```

 $pack\_file\_name(font\_name[f], "", ".vf");$ 
 $\text{if } \neg vf\_b\_open\_in(vf\_file) \text{ then return}$ 

```

This code is used in section 712.

714. $\langle \text{Process the preamble 714} \rangle \equiv$

```

 $\text{if } vf\_byte \neq pre \text{ then } bad\_vf("PRE\_command\_expected");$ 
 $\text{if } vf\_byte \neq vf.id \text{ then } bad\_vf("wrong\_id\_byte");$ 
 $cmd\_length \leftarrow vf\_byte;$ 
 $\text{for } k \leftarrow 1 \text{ to } cmd\_length \text{ do } call\_func(vf\_byte); \quad \{ \text{skip the comment} \}$ 
 $tmp\_b0 \leftarrow vf\_byte; tmp\_b1 \leftarrow vf\_byte; tmp\_b2 \leftarrow vf\_byte; tmp\_b3 \leftarrow vf\_byte;$ 
 $\text{if } ((tmp\_b0 \neq 0) \vee (tmp\_b1 \neq 0) \vee (tmp\_b2 \neq 0) \vee (tmp\_b3 \neq 0)) \wedge ((font\_check[f].b0 \neq 0) \vee (font\_check[f].b1 \neq 0) \vee (font\_check[f].b2 \neq 0) \vee (font\_check[f].b3 \neq 0)) \wedge ((tmp\_b0 \neq font\_check[f].b0) \vee (tmp\_b1 \neq font\_check[f].b1) \vee (tmp\_b2 \neq font\_check[f].b2) \vee (tmp\_b3 \neq font\_check[f].b3)) \text{ then}$ 
 $\quad \text{begin print\_nl("checksum\_mismatch\_in\_font"); print(font\_name[f]); print(".vf\_ignored");}$ 
 $\quad \text{end};$ 
 $\text{if } vf\_read\_signed(4) \text{ div } 20 \neq font\_dsizes[f] \text{ then}$ 
 $\quad \text{begin print\_nl("design\_size\_mismatch\_in\_font"); print(font\_name[f]); print(".vf\_ignored");}$ 
 $\quad \text{end};$ 
 $\quad update\_terminal$ 

```

This code is used in section 712.

715. *< Process the font definitions 715 >* ≡
 $cmd \leftarrow vf_byte; save_vf_nf \leftarrow vf_nf;$
while ($cmd \geq fnt_def1$) \wedge ($cmd \leq fnt_def1 + 3$) **do**
 begin *allocuffnts*; $vf_e_fnts[vf_nf] \leftarrow vf_read_unsigned(cmd - fnt_def1 + 1);$
 $vf_i_fnts[vf_nf] \leftarrow vf_def_font(f); incr(vf_nf); cmd \leftarrow vf_byte;$
 end;
 $vf_default_font[f] \leftarrow save_vf_nf; vf_local_font_num[f] \leftarrow vf_nf - save_vf_nf;$

This code is used in section 712.

716. *< Allocate memory for the new virtual font 716 >* ≡
 $vf_packet_base[f] \leftarrow new_vf_packet(f)$

This code is used in section 712.

717. *< Build a character packet 717 >* ≡
if $cmd = long_char$ **then**
 begin $packet_length \leftarrow vf_read_unsigned(4); cc \leftarrow vf_read_unsigned(4);$
 if $\neg is_valid_char(cc)$ **then** *bad_vf("invalid_character_code")*;
 $tfm_width \leftarrow store_scaled_f(vf_read_signed(4), font_size[f]);$
 end;
else begin $packet_length \leftarrow cmd; cc \leftarrow vf_byte;$
 if $\neg is_valid_char(cc)$ **then** *bad_vf("invalid_character_code")*;
 $tfm_width \leftarrow store_scaled_f(vf_read_unsigned(3), font_size[f]);$
 end;
if $packet_length < 0$ **then** *bad_vf("negative_packet_length")*;
if $packet_length > vf_max_packet_length$ **then** *bad_vf("packet_length_too_long")*;
if $tfm_width \neq char_width(f)(char_info(f)(cc))$ **then**
 begin *print_nl("character_width_mismatch_in_font")*; *print(font_name[f])*;
 print(".vf_ignored");
 end;
str_room(packet_length);
while $packet_length > 0$ **do**
 begin $cmd \leftarrow vf_byte; decr(packet_length);$
 < Cases of DVI commands that can appear in character packet 719 >
 if $cmd \neq nop$ **then** *append_char(cmd)*;
 $packet_length \leftarrow packet_length - cmd_length;$
 while $cmd_length > 0$ **do**
 begin *decr(cmd_length)*; *append_char(vf_byte)*;
 end;
 end;
if $stack_level \neq 0$ **then** *bad_vf("more_PUSHs_than_POPS_in_character_packet")*;
if $packet_length \neq 0$ **then** *bad_vf("invalid_packet_length_or_DVI_command_in_packet")*;
< Store the packet being built 718 >
 $cmd \leftarrow vf_byte$

This code is used in section 712.

718. *< Store the packet being built 718 >* ≡
 $s \leftarrow make_string; storepacket(f, cc, s); flush_str(s)$

This code is used in section 717.

719. \langle Cases of DVI commands that can appear in character packet 719 $\rangle \equiv$

```

if ( $cmd \geq set\_char\_0$ )  $\wedge$  ( $cmd \leq set\_char\_0 + 127$ ) then  $cmd\_length \leftarrow 0$ 
else if (( $fnt\_num\_0 \leq cmd$ )  $\wedge$  ( $cmd \leq fnt\_num\_0 + 63$ ))  $\vee$  (( $fnt1 \leq cmd$ )  $\wedge$  ( $cmd \leq fnt1 + 3$ )) then
begin if  $cmd \geq fnt1$  then
begin  $k \leftarrow vf\_read\_unsigned(cmd - fnt1 + 1)$ ;  $packet\_length \leftarrow packet\_length - (cmd - fnt1 + 1)$ ;
end
else  $k \leftarrow cmd - fnt\_num\_0$ ;
if  $k \geq 256$  then  $bad\_vf("too\_many\_local\_fonts")$ ;
 $n \leftarrow 0$ ;
while ( $n < vf\_local\_font\_num[f]$ )  $\wedge$  ( $vf\_e\_fnts[vf\_default\_font[f] + n] \neq k$ ) do  $incr(n)$ ;
if  $n = vf\_local\_font\_num[f]$  then  $bad\_vf("undefined\_local\_font")$ ;
if  $k \leq 63$  then  $append\_char(fnt\_num\_0 + k)$ 
else begin  $append\_char(fnt1)$ ;  $append\_char(k)$ ;
end;
 $cmd\_length \leftarrow 0$ ;  $cmd \leftarrow nop$ ;
end
else case  $cmd$  of
   $set\_rule, put\_rule$ :  $cmd\_length \leftarrow 8$ ;
   $four\_cases(set1)$ :  $cmd\_length \leftarrow cmd - set1 + 1$ ;
   $four\_cases(put1)$ :  $cmd\_length \leftarrow cmd - put1 + 1$ ;
   $four\_cases(right1)$ :  $cmd\_length \leftarrow cmd - right1 + 1$ ;
   $four\_cases(w1)$ :  $cmd\_length \leftarrow cmd - w1 + 1$ ;
   $four\_cases(x1)$ :  $cmd\_length \leftarrow cmd - x1 + 1$ ;
   $four\_cases(down1)$ :  $cmd\_length \leftarrow cmd - down1 + 1$ ;
   $four\_cases(y1)$ :  $cmd\_length \leftarrow cmd - y1 + 1$ ;
   $four\_cases(z1)$ :  $cmd\_length \leftarrow cmd - z1 + 1$ ;
   $four\_cases(xxx1)$ : begin  $cmd\_length \leftarrow vf\_read\_unsigned(cmd - xxx1 + 1)$ ;
     $packet\_length \leftarrow packet\_length - (cmd - xxx1 + 1)$ ;
    if  $cmd\_length > vf\_max\_packet\_length$  then  $bad\_vf("packet\_length\_too\_long")$ ;
    if  $cmd\_length < 0$  then  $bad\_vf("string\_of\_negative\_length")$ ;
     $append\_char(xxx1)$ ;  $append\_char(cmd\_length)$ ;  $cmd \leftarrow nop$ ;
    {  $cmd$  has been already stored above as  $xxx1$  }
  end;
   $w0, x0, y0, z0, nop$ :  $cmd\_length \leftarrow 0$ ;
   $push, pop$ : begin  $cmd\_length \leftarrow 0$ ;
    if  $cmd = push$  then
      if  $stack\_level = vf\_stack\_size$  then  $overflow("virtual\_font\_stack\_size", vf\_stack\_size)$ 
      else  $incr(stack\_level)$ 
    else if  $stack\_level = 0$  then  $bad\_vf("more\_POPs\_than\_PUSHs\_in\_character")$ 
      else  $decr(stack\_level)$ ;
    end;
  othercases  $bad\_vf("improper\_DVI\_command")$ ;
endcases

```

This code is used in section 717.

720.

```

procedure pdf_check_vf_cur_val;
  var f: internal_font_number;
begin f ← cur_val; do_vf(f);
if pdf_font_type[f] = virtual_font_type then
  pdf_error("font", "command cannot be used with virtual font");
end;
function auto_expand_vf(f : internal_font_number): boolean; { check for a virtual auto-expanded font }
  var bf, lf: internal_font_number; e, k: integer;
begin auto_expand_vf ← false;
if (¬pdf_font_auto_expand[f]) ∨ (pdf_font_blink[f] = null_font) then return;
  { not an auto-expanded font }
bf ← pdf_font_blink[f];
if pdf_font_type[bf] = new_font_type then { we must process the base font first }
  do_vf(bf);
if pdf_font_type[bf] ≠ virtual_font_type then return; { not a virtual font }
e ← pdf_font_expand_ratio[f];
for k ← 0 to vf_local_font_num[bf] - 1 do
  begin lf ← vf_default_font[bf] + k; allocvffnts; { copy vf local font numbers: }
  vf_e_fnts[vf_nf] ← vf_e_fnts[lf]; { definition of local vf fonts are expanded from base fonts: }
  vf_i_fnts[vf_nf] ← auto_expand_font(vf_i_fnts[lf], e);
  copy_expand_params(vf_i_fnts[vf_nf], vf_i_fnts[lf], e); incr(vf_nf);
  end;
vf_packet_base[f] ← vf_packet_base[bf]; vf_local_font_num[f] ← vf_local_font_num[bf];
vf_default_font[f] ← vf_nf - vf_local_font_num[f]; pdf_font_type[f] ← virtual_font_type;
auto_expand_vf ← true;
end;

```

721. The *do_vf_packet* procedure is called in order to interpret the character packet for a virtual character. Such a packet may contain the instruction to typeset a character from the same or an other virtual font; in such cases *do_vf_packet* calls itself recursively. The recursion level, i.e., the number of times this has happened, is kept in the global variable *vf_cur_s* and should not exceed *vf_max_recursion*.

⟨ Constants in the outer block 11 ⟩ +≡

```

vf_max_recursion = 10; { VF files shouldn't recurse beyond this level }
vf_stack_size = 100; { DVI files shouldn't push beyond this depth }

```

722. ⟨ Types in the outer block 18 ⟩ +≡

```

vf_stack_index = 0 .. vf_stack_size; { an index into the stack }
vf_stack_record = record stack_h, stack_v, stack_w, stack_x, stack_y, stack_z: scaled;
end;

```

723. ⟨ Global variables 13 ⟩ +≡

```

vf_cur_s: 0 .. vf_max_recursion; { current recursion level }
vf_stack: array [vf_stack_index] of vf_stack_record;
vf_stack_ptr: vf_stack_index; { pointer into vf_stack }

```

724. ⟨ Set initial values of key variables 21 ⟩ +≡

```

vf_cur_s ← 0; vf_stack_ptr ← 0;

```

725. Some functions for processing character packets.

```

function packet_read_signed(k : integer) : integer;
    { read k bytes as a signed integer from character packet }
    var i: integer;
    begin pdfassert((k > 0)  $\wedge$  (k  $\leq$  4)); i  $\leftarrow$  packet_byte;
        if i  $\geq$  128 then i  $\leftarrow$  i - 256;
        decr(k);
        while k > 0 do
            begin i  $\leftarrow$  i * 256 + packet_byte; decr(k);
            end;
        packet_read_signed  $\leftarrow$  i;
        end;
function packet_read_unsigned(k : integer) : integer;
    { read k bytes as an unsigned integer from character packet }
    var i: integer;
    begin pdfassert((k > 0)  $\wedge$  (k  $\leq$  4)); i  $\leftarrow$  packet_byte;
        if (k = 4)  $\wedge$  (i  $\geq$  128) then bad_vf("number_too_big");
        decr(k);
        while k > 0 do
            begin i  $\leftarrow$  i * 256 + packet_byte; decr(k);
            end;
        packet_read_unsigned  $\leftarrow$  i;
        end;
function packet_scaled(k : integer; fs : scaled) : scaled; { get k bytes from packet as scaled }
    begin packet_scaled  $\leftarrow$  store_scaled_f(packet_read_signed(k), fs);
    end;
procedure do_vf_packet(vf_f : internal_font_number; c : eight_bits);
    { typeset the DVI commands in the character packet for character c in current font f }
    label do_char, continue;
    var f, k, n: internal_font_number; save_cur_h, save_cur_v: scaled; cmd: integer; char_move: boolean;
        w, x, y, z: scaled; s: str_number;
    begin incr(vf_cur_s);
        if vf_cur_s > vf_max_recursion then
            overflow("max_level_recursion_of_virtual_fonts", vf_max_recursion);
            save_cur_v  $\leftarrow$  cur_v; save_cur_h  $\leftarrow$  cur_h; push_packet_state;
                { save pointer and length of the current packet }
            start_packet(vf_f, c); { set pointer and length of the new packet }
            f  $\leftarrow$  vf_i.fnts[vf.default_font[vf_f]]; w  $\leftarrow$  0; x  $\leftarrow$  0; y  $\leftarrow$  0; z  $\leftarrow$  0;
            while vf_packet_length > 0 do
                begin cmd  $\leftarrow$  packet_byte; { Do typesetting the DVI commands in virtual character packet 726 };
                continue: end;
                pop_packet_state; { restore pointer and length of the previous packet }
                cur_v  $\leftarrow$  save_cur_v; cur_h  $\leftarrow$  save_cur_h; decr(vf_cur_s);
            end;

```

726. The following code typesets a character to PDF output.

```

define output_one_char (#) ≡
  begin if pdf_font_type[f] = new_font_type then do_vf(f);
  if pdf_font_type[f] = virtual_font_type then do_vf_packet(f, #)
  else begin pdf_begin_string(f); pdf_print_char(f, #); adv_char_width(f, #, 4);
  end;
end

⟨ Do typesetting the DVI commands in virtual character packet 726 ⟩ ≡
  if (cmd ≥ set_char_0) ∧ (cmd ≤ set_char_0 + 127) then
    begin if ¬is_valid_char(cmd) then
      begin char_warning(f, cmd); goto continue;
      end;
    c ← cmd; char_move ← true; goto do_char;
    end
  else if ((fnt_num_0 ≤ cmd) ∧ (cmd ≤ fnt_num_0 + 63)) ∨ (cmd = fnt1) then
    begin if cmd = fnt1 then k ← packet_byte
    else k ← cmd - fnt_num_0;
    n ← 0;
    while (n < vf_local_font_num[vf_f]) ∧ (vf_e_fnts[vf_default_font[vf_f] + n] ≠ k) do incr(n);
    if (n = vf_local_font_num[vf_f]) then pdf_error("vf", "local_font_not_found")
    else f ← vf_i_fnts[vf_default_font[vf_f] + n];
    end
  else case cmd of
    push: begin vf_stack[vf_stack_ptr].stack_h ← cur_h; vf_stack[vf_stack_ptr].stack_v ← cur_v;
    vf_stack[vf_stack_ptr].stack_w ← w; vf_stack[vf_stack_ptr].stack_x ← x;
    vf_stack[vf_stack_ptr].stack_y ← y; vf_stack[vf_stack_ptr].stack_z ← z; incr(vf_stack_ptr);
    end;
    pop: begin decr(vf_stack_ptr); cur_h ← vf_stack[vf_stack_ptr].stack_h;
    cur_v ← vf_stack[vf_stack_ptr].stack_v; w ← vf_stack[vf_stack_ptr].stack_w;
    x ← vf_stack[vf_stack_ptr].stack_x; y ← vf_stack[vf_stack_ptr].stack_y;
    z ← vf_stack[vf_stack_ptr].stack_z;
    end;
    four_cases(set1), four_cases(put1): begin if (set1 ≤ cmd) ∧ (cmd ≤ set1 + 3) then
      begin tmp_int ← packet_read_unsigned(cmd - set1 + 1); char_move ← true;
      end
    else begin tmp_int ← packet_read_unsigned(cmd - put1 + 1); char_move ← false;
      end;
    if ¬is_valid_char(tmp_int) then
      begin char_warning(f, tmp_int); goto continue;
      end;
    c ← tmp_int; goto do_char;
    end;
    set_rule, put_rule: begin rule_ht ← packet_scaled(4, font_size[vf_f]);
    rule_wd ← packet_scaled(4, font_size[vf_f]);
    if (rule_wd > 0) ∧ (rule_ht > 0) then
      begin pdf_set_rule(cur_h, cur_v, rule_wd, rule_ht);
      if cmd = set_rule then cur_h ← cur_h + rule_wd;
      end;
    end;
    four_cases(right1): cur_h ← cur_h + packet_scaled(cmd - right1 + 1, font_size[vf_f]);
    w0, four_cases(w1): begin if cmd > w0 then w ← packet_scaled(cmd - w0, font_size[vf_f]);
    cur_h ← cur_h + w;
  
```

```

end;
x0,four_cases(x1): begin if cmd > x0 then  $x \leftarrow \text{packet\_scaled}(\text{cmd} - x0, \text{font\_size}[vf_f]);$ 
cur_h \leftarrow cur_h + x;
end;
four_cases(down1): cur_v \leftarrow cur_v + \text{packet\_scaled}(\text{cmd} - down1 + 1, \text{font\_size}[vf_f]);
y0,four_cases(y1): begin if cmd > y0 then  $y \leftarrow \text{packet\_scaled}(\text{cmd} - y0, \text{font\_size}[vf_f]);$ 
cur_v \leftarrow cur_v + y;
end;
z0,four_cases(z1): begin if cmd > z0 then  $z \leftarrow \text{packet\_scaled}(\text{cmd} - z0, \text{font\_size}[vf_f]);$ 
cur_v \leftarrow cur_v + z;
end;
four_cases(xxx1): begin tmp_int \leftarrow \text{packet\_read\_unsigned}(\text{cmd} - xxx1 + 1); str_room(tmp_int);
while tmp_int > 0 do
    begin decr(tmp_int); append_char(packet_byte);
    end;
s \leftarrow make_string; literal(s, scan_special, false); flush_str(s);
end;
othercases pdf_error("vf", "invalid_DVI_command");
endcases;
goto continue;
do_char: if is_valid_char(c) then output_one_char(c)
else char_warning(f, c);
if char_move then  $cur_h \leftarrow cur_h + \text{char\_width}(f)(\text{char\_info}(f)(c))$ 

```

This code is used in section 725.

727. PDF shipping out. To ship out a TeX box to PDF page description we need to implement *pdf_hlist_out*, *pdf_vlist_out* and *pdf_ship_out*, which are equivalent to the TeX' original *hlist_out*, *vlist_out* and *ship_out* resp. But first we need to declare some procedures needed in *pdf_hlist_out* and *pdf_vlist_out*.

```

< Declare procedures needed in pdf_hlist_out, pdf_vlist_out 727 > ≡
procedure pdf_out_literal(p : pointer);
  var old_setting: 0 .. max_selector; { holds print selector }
    s: str_number; h: halfword; q, r: pointer; { temporary variables for list manipulation }
    old_mode: integer; { saved mode }
  begin old_setting ← selector;
  if subtype(p) = pdf_lateliteral_node then
    begin { Expand macros in the token list and make link(def_ref) point to the result 1618 };
      h ← def_ref;
    end
  else h ← pdf_literal_data(p);
  selector ← new_string; show_token_list(link(h), null, pool_size - pool_ptr); selector ← old_setting;
  s ← make_string; literal(s, pdf_literal_mode(p), false); flush_str(s);
  if subtype(p) = pdf_lateliteral_node then flush_list(def_ref);
  end;
procedure pdf_out_colorstack(p : pointer);
  var old_setting: 0 .. max_selector; { holds print selector }
    s: str_number; cmd: integer; stack_no: integer; literal_mode: integer;
  begin cmd ← pdf_colorstack_cmd(p); stack_no ← pdf_colorstack_stack(p);
  if stack_no ≥ colorstackused then
    begin print_nl(""); print("Color_stack"); print_int(stack_no);
      print(" _is _not _initialized _for _use!"); print_nl("");
    end;
  case cmd of
    colorstack_set, colorstack_push: begin old_setting ← selector; selector ← new_string;
      show_token_list(link(pdf_colorstack_data(p)), null, pool_size - pool_ptr); selector ← old_setting;
      s ← make_string;
      if cmd = colorstack_set then literal_mode ← colorstackset(stack_no, s)
      else literal_mode ← colorstackpush(stack_no, s);
      if length(s) > 0 then literal(s, literal_mode, false);
      flush_str(s); return;
    end;
    colorstack_pop: literal_mode ← colorstackpop(stack_no);
    colorstack_current: literal_mode ← colorstackcurrent(stack_no);
    othercases confusion("pdfcolorstack")
  endcases;
  if cur_length > 0 then
    begin s ← make_string; literal(s, literal_mode, false); flush_str(s);
    end
  end;
procedure pdf_out_colorstack_startpage;
  var i: integer; max: integer; start_status: integer; literal_mode: integer; s: str_number;
  begin i ← 0; max ← colorstackused;
  while i < max do
    begin start_status ← colorstackskippagestart(i);
    if start_status = 0 then
      begin literal_mode ← colorstackcurrent(i);
      if cur_length > 0 then
        begin s ← make_string; literal(s, literal_mode, false); flush_str(s);
      end
    end
  end;

```

```

    end;
  end;
  incr(i);
end;
end;

procedure pdf_out_setmatrix(p : pointer);
var old_setting: 0 .. max_selector; { holds print selector }
  s: str_number;
begin old_setting ← selector; selector ← new_string;
show_token_list(link(pdf_setmatrix_data(p)), null, pool_size - pool_ptr); selector ← old_setting;
str_room(7); str_pool[pool_ptr] ← 0; { make C string for pdfsetmatrix }
if pdfsetmatrix(str_start[str_ptr], cur_h, cur_page_height - cur_v) = 1 then
  begin str_room(7); append_char("L"); append_char("0"); append_char("L"); append_char("0");
  append_char("L"); append_char("c"); append_char("m"); s ← make_string; literal(s, set_origin, false);
  end
else begin pdf_error("\pdfsetmatrix", "Unrecognized\format.");
  end;
flush_str(s);
end;

procedure pdf_out_save(p : pointer);
begin checkpdfsave(cur_h, cur_v); literal("q", set_origin, false);
end;

procedure pdf_out_restore(p : pointer);
begin checkpdfrestore(cur_h, cur_v); literal("Q", set_origin, false);
end;

procedure pdf_special(p : pointer);
var old_setting: 0 .. max_selector; { holds print selector }
  s: str_number; h: halfword; q, r: pointer; { temporary variables for list manipulation }
  old_mode: integer; { saved mode }
begin old_setting ← selector; selector ← selector;
if subtype(p) = latespecial_node then
  begin { Expand macros in the token list and make link(def_ref) point to the result 1618 };
  h ← def_ref;
  end
else h ← write_tokens(p);
selector ← new_string; show_token_list(link(h), null, pool_size - pool_ptr); selector ← old_setting;
s ← make_string; literal(s, scan_special, true); flush_str(s);
if subtype(p) = latespecial_node then flush_list(def_ref);
end;

procedure pdf_print_toks(p : pointer); { print tokens list p }
var s: str_number;
begin s ← tokens_to_string(p);
if length(s) > 0 then pdf_print(s);
flush_str(s);
end;

procedure pdf_print_toks_ln(p : pointer); { print tokens list p }
var s: str_number;
begin s ← tokens_to_string(p);
if length(s) > 0 then
  begin pdf_print_ln(s);
  end;
flush_str(s);

```

```
end;
```

See also sections 772, 778, 785, 1564, 1630, 1635, 1636, and 1637.

This code is used in section 729.

728. Similar to *vlist_out*, *pdf_vlist_out* needs to be declared forward.

```
procedure pdf_vlist_out; forward;
```

729. The implementation of procedure *pdf_hlist_out* is similar to *hlist_out*.

```
< Declare procedures needed in pdf_hlist_out, pdf_vlist_out 727 >
procedure pdf_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: scaled; { what cur_h 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 }
    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 }
    i: small_number; { index to scan pdf_link_stack }
  begin cur_g  $\leftarrow$  0; cur_glue  $\leftarrow$  float_constant(0); this_box  $\leftarrow$  temp_ptr; g_order  $\leftarrow$  glue_order(this_box);
    g_sign  $\leftarrow$  glue_sign(this_box); p  $\leftarrow$  list_ptr(this_box); incr(cur_s); base_line  $\leftarrow$  cur_v;
    prev_p  $\leftarrow$  this_box + list_offset; < Initialize hlist_out for mixed direction typesetting 1714 >;
    left_edge  $\leftarrow$  cur_h; < Create link annotations for the current hbox if needed 730 >;
    while p  $\neq$  null do < Output node p for pdf_hlist_out and move to the next node, maintaining the
      condition cur_v = base_line 731 >;
    < Finish hlist_out for mixed direction typesetting 1715 >;
    decr(cur_s);
  end;
```

730. < Create link annotations for the current hbox if needed 730 > \equiv

```
for i  $\leftarrow$  1 to pdf_link_stack_ptr do
  begin pdfassert(is_running(pdf_width(pdf_link_stack[i].link_node)));
  if (pdf_link_stack[i].nesting_level = cur_s)  $\wedge$  gen_running_link then
    append_link(this_box, left_edge, base_line, i);
  end
```

This code is used in section 729.

731. ⟨ Output node p for pdf_hlist_out and move to the next node, maintaining the condition
 $cur_v = base_line$ 731 ⟩ ≡

reswitch: **if** $is_char_node(p)$ **then**

begin repeat $f \leftarrow font(p); c \leftarrow character(p);$
if $is_valid_char(c)$ **then** $output_one_char(c)$
else $char_warning(f, c);$
 $cur_h \leftarrow cur_h + char_width(f)(char_info(f)(c)); prev_p \leftarrow link(prev_p);$
{ N.B.: not $prev_p \leftarrow p$, p might be *lig-trick* }
 $p \leftarrow link(p);$
until $\neg is_char_node(p);$
end

else ⟨ Output the non-*char_node* p for pdf_hlist_out and move to the next node 732 ⟩

This code is used in section 729.

732. ⟨ Output the non-*char_node* p for pdf_hlist_out and move to the next node 732 ⟩ ≡

begin case $type(p)$ **of**

hlist_node, vlist_node: ⟨ (pdfTEX) Output a box in an hlist 733 ⟩;
rule_node: **begin** $rule_ht \leftarrow height(p); rule_dp \leftarrow depth(p); rule_wd \leftarrow width(p); goto fin_rule;$
end;

whatsit_node: ⟨ Output the whatsit node p in pdf_hlist_out 1645 ⟩;
glue_node: ⟨ (pdfTEX) Move right or output leaders 735 ⟩;
margin_kern_node, kern_node: $cur_h \leftarrow cur_h + width(p);$
math_node: ⟨ Handle a math node in $hlist_out$ 1716 ⟩;
ligature_node: ⟨ Make node p look like a *char_node* and **goto** *reswitch* 826 ⟩;
{ Cases of *hlist_out* that arise in mixed direction text only 1720 }
othercases do_nothing
endcases;
goto $next_p;$

fin_rule: ⟨ (pdfTEX) Output a rule in an hlist 734 ⟩;
move_past: $cur_h \leftarrow cur_h + rule_wd;$
next_p: $prev_p \leftarrow p; p \leftarrow link(p);$
end

This code is used in section 731.

733. ⟨ (pdfTEX) Output a box in an hlist 733 ⟩ ≡

if $list_ptr(p) = null$ **then** $cur_h \leftarrow cur_h + width(p)$
else begin $cur_v \leftarrow base_line + shift_amount(p);$ { shift the box down }
 $temp_ptr \leftarrow p; edge \leftarrow cur_h + width(p);$
if $cur_dir = right_to_left$ **then** $cur_h \leftarrow edge;$
if $type(p) = vlist_node$ **then** pdf_vlist_out **else** $pdf_hlist_out;$
 $cur_h \leftarrow edge; cur_v \leftarrow base_line;$
end

This code is used in section 732.

734. ⟨ (pdfTEX) Output a rule in an hlist 734 ⟩ ≡

if $is_running(rule_ht)$ **then** $rule_ht \leftarrow height(this_box);$
if $is_running(rule_dp)$ **then** $rule_dp \leftarrow depth(this_box);$
 $rule_ht \leftarrow rule_ht + rule_dp;$ { this is the rule thickness }
if $(rule_ht > 0) \wedge (rule_wd > 0)$ **then** { we don't output empty rules }
begin $cur_v \leftarrow base_line + rule_dp; pdf_set_rule(cur_h, cur_v, rule_wd, rule_ht); cur_v \leftarrow base_line;$
end

This code is used in section 732.

735. $\langle (\text{pdfTeX}) \text{ Move right or output leaders } 735 \rangle \equiv$

```

begin  $g \leftarrow \text{glue\_ptr}(p); rule\_wd \leftarrow \text{width}(g) - cur\_g;$ 
if  $g.\text{sign} \neq \text{normal}$  then
  begin if  $g.\text{sign} = \text{stretching}$  then
    begin if  $\text{stretch\_order}(g) = g.\text{order}$  then
      begin  $cur\_glue \leftarrow cur\_glue + \text{stretch}(g); vet\_glue(\text{float}(\text{glue\_set}(\text{this\_box})) * cur\_glue);$ 
       $cur\_g \leftarrow \text{round}(\text{glue\_temp});$ 
      end;
    end
  else if  $\text{shrink\_order}(g) = g.\text{order}$  then
    begin  $cur\_glue \leftarrow cur\_glue - \text{shrink}(g); vet\_glue(\text{float}(\text{glue\_set}(\text{this\_box})) * cur\_glue);$ 
     $cur\_g \leftarrow \text{round}(\text{glue\_temp});$ 
    end;
  end;
rule_wd  $\leftarrow rule\_wd + cur\_g;$ 
if  $eTeX\_ex$  then ⟨Handle a glue node for mixed direction typesetting 1699⟩;
if  $\text{subtype}(p) \geq a.\text{leaders}$  then
  ⟨(pdfTeX) Output leaders in an hlist, goto  $\text{fin\_rule}$  if a rule or to  $\text{next\_p}$  if done 736⟩;
  goto  $\text{move\_past};$ 
end

```

This code is used in section 732.

736. $\langle (\text{pdfTeX}) \text{ Output leaders in an hlist, goto } \text{fin_rule} \text{ if a rule or to } \text{next_p} \text{ if done } 736 \rangle \equiv$

```

begin  $leader\_box \leftarrow \text{leader\_ptr}(p);$ 
if  $\text{type}(leader\_box) = \text{rule\_node}$  then
  begin  $rule\_ht \leftarrow \text{height}(leader\_box); rule\_dp \leftarrow \text{depth}(leader\_box); \text{goto } \text{fin\_rule};$ 
  end;
 $leader\_wd \leftarrow \text{width}(leader\_box);$ 
if  $(leader\_wd > 0) \wedge (rule\_wd > 0)$  then
  begin  $rule\_wd \leftarrow rule\_wd + 10; \{ \text{compensate for floating-point rounding} \}$ 
  if  $cur\_dir = \text{right\_to\_left}$  then  $cur\_h \leftarrow cur\_h - 10;$ 
  edge  $\leftarrow cur\_h + rule\_wd; lx \leftarrow 0; \{ \text{Let } cur\_h \text{ be the position of the first box, and set } leader\_wd + lx \text{ to}$ 
  the spacing between corresponding parts of boxes 655;
  while  $cur\_h + leader\_wd \leq edge$  do
    ⟨(pdfTeX) Output a leader box at  $cur\_h$ , then advance  $cur\_h$  by  $leader\_wd + lx$  737⟩;
  if  $cur\_dir = \text{right\_to\_left}$  then  $cur\_h \leftarrow edge$ 
  else  $cur\_h \leftarrow edge - 10;$ 
  goto  $\text{next\_p};$ 
  end;
end

```

This code is used in section 735.

737. $\langle (\text{pdfTeX}) \text{ Output a leader box at } cur_h, \text{ then advance } cur_h \text{ by } leader_wd + lx \text{ 737} \rangle \equiv$

```

begin  $cur\_v \leftarrow base\_line + shift\_amount(leader\_box);$ 
 $save\_h \leftarrow cur\_h; temp\_ptr \leftarrow leader\_box;$ 
if  $cur\_dir = \text{right\_to\_left}$  then  $cur\_h \leftarrow cur\_h + leader\_wd;$ 
outer_doing_leaders  $\leftarrow doing\_leaders; doing\_leaders \leftarrow true;$ 
if  $\text{type}(leader\_box) = vlist\_node$  then  $\text{pdf\_vlist\_out}$  else  $\text{pdf\_hlist\_out};$ 
doing_leaders  $\leftarrow outer\_doing\_leaders; cur\_v \leftarrow base\_line; cur\_h \leftarrow save\_h + leader\_wd + lx;$ 
end

```

This code is used in section 736.

738. The *pdf_vlist_out* routine is similar to *pdf_hlist_out*, but a bit simpler.

```

procedure pdf_vlist_out; { output a pdf_vlist_node box }
  label move_past, fin_rule, next_p;
  var left_edge: scaled; { the left coordinate for this box }
    top_edge: scaled; { the top coordinate for this box }
    save_v: scaled; { what cur_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 vlist }
    leader_box: pointer; { the leader box being replicated }
    leader_ht: scaled; { height of leader box being replicated }
    lx: scaled; { extra space between leader boxes }
    outer_doing_leaders: boolean; { were we doing leaders? }
    edge: scaled; { bottom boundary of leader space }
    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); left_edge ← cur_h;
    cur_v ← cur_v − height(this_box); top_edge ← cur_v; { Create thread for the current vbox if needed 739 };
    while p ≠ null do { Output node p for pdf_vlist_out and move to the next node, maintaining the
      condition cur_h = left_edge 740 };
      deer(cur_s);
    end;

```

739. { Create thread for the current vbox if needed 739 } ≡

```

if (last_thread ≠ null) ∧ is_running(pdf_thread_dp) ∧ (pdf_thread_level = cur_s) then
  append_thread(this_box, left_edge, top_edge + height(this_box))

```

This code is used in section 738.

740. { Output node *p* for *pdf_vlist_out* and move to the next node, maintaining the condition

```

cur_h = left_edge 740 } ≡
begin if is_char_node(p) then confusion("pdfvlistout")
  else { Output the non-char_node p for pdf_vlist_out 741 };
next_p: p ← link(p);
end

```

This code is used in section 738.

741. \langle Output the non-*char_node* *p* for *pdf_vlist_out* 741 $\rangle \equiv$

```

begin case type(p) of
  hlist_node, vlist_node: ⟨(pdfTeX) Output a box in a vlist 742⟩;
  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 pdf_vlist_out 1639⟩;
  glue_node: ⟨(pdfTeX) Move down or output leaders 744⟩;
  kern_node: cur_v ← cur_v + width(p);
  othercases do_nothing
endcases;
goto next_p;
fin_rule: ⟨(pdfTeX) Output a rule in a vlist, goto next_p 743⟩;
move_past: cur_v ← cur_v + rule_ht;
end

```

This code is used in section 740.

742. \langle (pdfTeX) Output a box in a vlist 742 $\rangle \equiv$

```

if list_ptr(p) = null then cur_v ← cur_v + height(p) + depth(p)
else begin cur_v ← cur_v + height(p); save_v ← cur_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 pdf_vlist_out else pdf_hlist_out;
  cur_v ← save_v + depth(p); cur_h ← left_edge;
end

```

This code is used in section 741.

743. \langle (pdfTeX) Output a rule in a vlist, goto next_p 743 $\rangle \equiv$

```

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;
  pdf_set_rule(cur_h, cur_v, rule_wd, rule_ht); cur_h ← left_edge;
  end;
goto next_p

```

This code is used in section 741.

```

744. ⟨ (pdfTeX) Move down or output leaders 744 ⟩ ≡
begin  $g \leftarrow glue\_ptr(p)$ ;  $rule\_ht \leftarrow width(g) - cur\_g$ ;
if  $g\_sign \neq normal$  then
begin if  $g\_sign = stretching$  then
begin if  $stretch\_order(g) = g\_order$  then
begin  $cur\_glue \leftarrow cur\_glue + stretch(g)$ ;  $vet\_glue(float(glue\_set(this\_box)) * cur\_glue)$ ;
 $cur\_g \leftarrow round(glue\_temp)$ ;
end;
end
else if  $shrink\_order(g) = g\_order$  then
begin  $cur\_glue \leftarrow cur\_glue - shrink(g)$ ;  $vet\_glue(float(glue\_set(this\_box)) * cur\_glue)$ ;
 $cur\_g \leftarrow round(glue\_temp)$ ;
end;
end;
rule_ht  $\leftarrow rule\_ht + cur\_g$ ;
if  $subtype(p) \geq a\_leaders$  then
⟨ (pdfTeX) Output leaders in a vlist, goto fin_rule if a rule or to next_p if done 745 ⟩;
goto move_past;
end

```

This code is used in section 741.

```

745. ⟨ (pdfTeX) Output leaders in a vlist, goto fin_rule if a rule or to next_p if done 745 ⟩ ≡
begin  $leader\_box \leftarrow leader\_ptr(p)$ ;
if  $type(leader\_box) = rule\_node$  then
begin  $rule\_wd \leftarrow width(leader\_box)$ ;  $rule\_dp \leftarrow 0$ ; goto fin_rule;
end;
 $leader\_ht \leftarrow height(leader\_box) + depth(leader\_box)$ ;
if ( $leader\_ht > 0$ )  $\wedge$  ( $rule\_ht > 0$ ) then
begin  $rule\_ht \leftarrow rule\_ht + 10$ ; { compensate for floating-point rounding }
 $edge \leftarrow cur\_v + rule\_ht$ ;  $lx \leftarrow 0$ ; { Let  $cur\_v$  be the position of the first box, and set  $leader\_ht + lx$  to
the spacing between corresponding parts of boxes 664 };
while  $cur\_v + leader\_ht \leq edge$  do
⟨ (pdfTeX) Output a leader box at  $cur\_v$ , then advance  $cur\_v$  by  $leader\_ht + lx$  746 ⟩;
 $cur\_v \leftarrow edge - 10$ ; goto next_p;
end;
end

```

This code is used in section 744.

```

746. ⟨ (pdfTeX) Output a leader box at  $cur\_v$ , then advance  $cur\_v$  by  $leader\_ht + lx$  746 ⟩ ≡
begin if  $cur\_dir = right\_to\_left$  then  $cur\_h \leftarrow left\_edge - shift\_amount(leader\_box)$ 
else  $cur\_h \leftarrow left\_edge + shift\_amount(leader\_box)$ ;
 $cur\_v \leftarrow cur\_v + height(leader\_box)$ ;  $save\_v \leftarrow cur\_v$ ;  $temp\_ptr \leftarrow leader\_box$ ;
 $outer\_doing\_leaders \leftarrow doing\_leaders$ ;  $doing\_leaders \leftarrow true$ ;
if  $type(leader\_box) = vlist\_node$  then  $pdf\_vlist\_out$  else  $pdf\_hlist\_out$ ;
 $doing\_leaders \leftarrow outer\_doing\_leaders$ ;  $cur\_h \leftarrow left\_edge$ ;
 $cur\_v \leftarrow save\_v - height(leader\_box) + leader\_ht + lx$ ;
end

```

This code is used in section 745.

747. *fix_pdfoutput* freezes *pdfoutput* when something has been written to the output.

```
procedure fix_pdfoutput;
begin if ¬fixed_pdfoutput_set then
  begin fixed_pdfoutput ← pdf_output; fixed_pdfoutput_set ← true;
  end
else if fixed_pdfoutput ≠ pdf_output then pdf_error("setup",
  "\pdfoutput can only be changed before anything is written to the output");
if fixed_pdfoutput_set then fix_pdf_draftmode;
end;
```

748. *fix_pdf_draftmode* freezes *pdfdraftmode* when something has been written to the output and also switches some things off when draftmode is on.

```
procedure fix_pdf_draftmode;
begin if ¬fixed_pdf_draftmode_set then
  begin fixed_pdf_draftmode ← pdf_draftmode; fixed_pdf_draftmode_set ← true;
  end
else if fixed_pdf_draftmode ≠ pdf_draftmode then pdf_error("setup",
  "\pdfdraftmode can only be changed before anything is written to the output");
if fixed_pdf_draftmode_set ∧ fixed_pdf_draftmode > 0 then
  begin fixed_pdf_draftmode_set ← true; pdf_compress_level ← 0; fixed_pdf_objcompresslevel ← 0;
  end;
end;
```

749. *substr_of_str* is used in *pdf_ship_out* and *pdf_print_info*.

```
function substr_of_str(s, t : str_number): boolean;
label continue, exit;
var j, k, kk: pool_pointer; { running indices }
begin k ← str_start[t];
while (k < str_start[t + 1] – length(s)) do
  begin j ← str_start[s]; kk ← k;
  while (j < str_start[s + 1]) do
    begin if str_pool[j] ≠ str_pool[kk] then goto continue;
    incr(j); incr(kk);
    end;
  substr_of_str ← true; return;
  continue: incr(k);
  end;
substr_of_str ← false;
end;
```

750. *pdf_ship_out* is used instead of *ship_out* to shipout a box to PDF output. If *shipping_page* is not set then the output will be a Form object, otherwise it will be a Page object.

```

procedure pdf_ship_out(p : pointer; shipping_page : boolean); { output the box p }
  label done, done1;
  var i, j, k: integer; { general purpose accumulators }
    s: pool_pointer; { index into str_pool }
  mediabox_given: boolean; save_font_list: pointer;
    { to save pdf_font_list during flushing pending forms }
  save_obj_list: pointer; { to save pdf_obj_list }
  save_ximage_list: pointer; { to save pdf_ximage_list }
  save_xform_list: pointer; { to save pdf_xform_list }
  save_image_procset: integer; { to save pdf_image_procset }
  save_text_procset: integer; { to save pdf_text_procset }
  pdf_last_resources: integer; { pointer to most recently generated Resources object }
begin if tracing_output > 0 then
  begin print_nl(""); print_ln; print("Completed_box_being_shipped_out");
  end;
if  $\neg$ init_pdf_output then
  begin {Initialize variables for PDF output 792};
  init_pdf_output  $\leftarrow$  true;
  end;
is_shipping_page  $\leftarrow$  shipping_page;
if shipping_page then
  begin if term_offset > max_print_line - 9 then print_ln
  else if (term_offset > 0)  $\vee$  (file_offset > 0) then print_char("[");
  print_char("["); j  $\leftarrow$  9;
  while (count(j) = 0)  $\wedge$  (j > 0) do decr(j);
  for k  $\leftarrow$  0 to j do
    begin print_int(count(k));
    if k < j then print_char(".");
    end;
  update_terminal;
  end;
if tracing_output > 0 then
  begin if shipping_page then print_char("]");
  begin_diagnostic; show_box(p); end_diagnostic(true);
  end;
{ (pdfTeX) Ship box p out 751 };
if eTeX_ex then { Check for LR anomalies at the end of ship_out 1730 };
if (tracing_output  $\leq$  0)  $\wedge$  shipping_page then print_char("]");
dead_cycles  $\leftarrow$  0; update_terminal; { progress report }
{ Flush the box from memory, showing statistics if requested 667 };
end;

```

751. { (pdfTeX) Ship box *p* out 751 } =

```

{ Update the values of max_h and max_v; but if the page is too large, goto done 669 };
{ Initialize variables as pdf_ship_out begins 752 };
if type(p) = vlist_node then pdf_vlist_out else pdf_hlist_out;
if shipping_page then incr(total_pages);
cur_s  $\leftarrow$  -1; { Finish shipping 759 };
done:

```

This code is used in section 750.

752. \langle Initialize variables as pdf_ship_out begins 752 $\rangle \equiv$

```

fix_pdfoutput; temp_ptr  $\leftarrow p$ ; prepare_mag; pdf_last_resources  $\leftarrow pdf\_new\_objnum$ ;
pdf_page_group_val  $\leftarrow 0$ ;  $\langle$  Reset resource lists 753  $\rangle$ ;
if  $\neg shipping\_page$  then
  begin pdf_xform_width  $\leftarrow width(p)$ ; pdf_xform_height  $\leftarrow height(p)$ ; pdf_xform_depth  $\leftarrow depth(p)$ ;
  pdf_begin_dict(pdf_cur_form, 0); pdf_last_stream  $\leftarrow pdf\_cur\_form$ ; cur_v  $\leftarrow height(p)$ ; cur_h  $\leftarrow 0$ ;
  pdf_origin_h  $\leftarrow 0$ ; pdf_origin_v  $\leftarrow pdf\_xform\_height + pdf\_xform\_depth$ ;
  end
else begin  $\langle$  Calculate page dimensions and margins 755  $\rangle$ ;
  pdf_last_page  $\leftarrow get\_obj(obj\_type\_page, total\_pages + 1, 0)$ ; obj_aux(pdf_last_page)  $\leftarrow 1$ ;
  { mark that this page has been created }
  pdf_new_dict(obj_type_others, 0, 0); pdf_last_stream  $\leftarrow obj\_ptr$ ; cur_h  $\leftarrow cur\_h\_offset$ ;
  cur_v  $\leftarrow height(p) + cur\_v\_offset$ ; pdf_origin_h  $\leftarrow 0$ ; pdf_origin_v  $\leftarrow cur\_page\_height$ ;
   $\langle$  Reset PDF mark lists 754  $\rangle$ ;
  end;
if  $\neg shipping\_page$  then
  begin  $\langle$  Write out Form stream header 756  $\rangle$ ;
  end;
 $\langle$  Start stream of page/form contents 757  $\rangle$ 

```

This code is used in section 751.

753. \langle Reset resource lists 753 $\rangle \equiv$

```

pdf_font_list  $\leftarrow null$ ; pdf_obj_list  $\leftarrow null$ ; pdf_xform_list  $\leftarrow null$ ; pdf_ximage_list  $\leftarrow null$ ;
pdf_text_procset  $\leftarrow false$ ; pdf_image_procset  $\leftarrow 0$ 

```

This code is used in sections 752 and 775.

754. \langle Reset PDF mark lists 754 $\rangle \equiv$

```

pdf_annot_list  $\leftarrow null$ ; pdf_link_list  $\leftarrow null$ ; pdf_dest_list  $\leftarrow null$ ; pdf_bead_list  $\leftarrow null$ ; last_thread  $\leftarrow null$ 

```

This code is used in section 752.

755. \langle Calculate page dimensions and margins 755 $\rangle \equiv$

```

cur_h_offset  $\leftarrow pdf\_h\_origin + h\_offset$ ; cur_v_offset  $\leftarrow pdf\_v\_origin + v\_offset$ ;
if pdf_page_width  $\neq 0$  then cur_page_width  $\leftarrow pdf\_page\_width$ 
else cur_page_width  $\leftarrow width(p) + 2 * cur\_h\_offset$ ;
if pdf_page_height  $\neq 0$  then cur_page_height  $\leftarrow pdf\_page\_height$ 
else cur_page_height  $\leftarrow height(p) + depth(p) + 2 * cur\_v\_offset$ 

```

This code is used in section 752.

756. Here we write out the header for Form.

\langle Write out Form stream header 756 $\rangle \equiv$

```

pdf_print_ln("/Type\u2225/XObject"); pdf_print_ln("/Subtype\u2225/Form");
if obj_xform_attr(pdf_cur_form)  $\neq null$  then
  begin pdf_print_toks_ln(obj_xform_attr(pdf_cur_form)); delete_toks(obj_xform_attr(pdf_cur_form));
  end;
pdf_print("/BBox\u2225["); pdf_print("0\u22250\u2225"); pdf_print_bp(pdf_xform_width); pdf_out("]\u2225");
pdf_print_bp(pdf_xform_height + pdf_xform_depth); pdf_print_ln("] "); pdf_print_ln("/FormType\u22251");
pdf_print_ln("/Matrix\u2225[1\u22250\u22250\u22251\u22250\u22250\u2225]"); pdf_indirect_ln("Resources", pdf_last_resources)

```

This code is used in section 752.

757. \langle Start stream of page/form contents 757 $\rangle \equiv$
 $\text{pdf_begin_stream};$
 $\text{if } \textit{shipping_page} \text{ then}$
 $\quad \text{begin} \langle$ Adjust transformation matrix for the magnification ratio 758 $\rangle;$
 $\quad \text{end};$
 $\quad \text{pdfshipoutbegin}(\textit{shipping_page});$
 $\quad \text{if } \textit{shipping_page} \text{ then } \text{pdf_out_colorstack_startpage};$

This code is used in section 752.

758. \langle Adjust transformation matrix for the magnification ratio 758 $\rangle \equiv$
 $\text{prepare_mag};$
 $\text{if } \textit{mag} \neq 1000 \text{ then}$
 $\quad \text{begin } \text{pdf_print_real}(\textit{mag}, 3); \text{ pdf_print}(" \u2225 0 \u2225 0 \u2225 "); \text{ pdf_print_real}(\textit{mag}, 3); \text{ pdf_print_ln}(" \u2225 0 \u2225 0 \u2225 cm");$
 $\quad \text{end}$

This code is used in section 757.

759. \langle Finish shipping 759 $\rangle \equiv$
 \langle Finish stream of page/form contents 760 $\rangle;$
 $\text{if } \textit{shipping_page} \text{ then}$
 $\quad \text{begin} \langle$ Write out page object 769 $\rangle;$
 $\quad \text{end};$
 \langle Write out resource lists 761 $\rangle;$
 $\text{if } \textit{shipping_page} \text{ then}$
 $\quad \text{begin} \langle$ Write out pending PDF marks 780 $\rangle;$
 $\quad \text{end};$
 \langle Write out resources dictionary 762 $\rangle;$
 \langle Flush resource lists 764 $\rangle;$
 $\text{if } \textit{shipping_page} \text{ then}$
 $\quad \text{begin} \langle$ Flush PDF mark lists 765 $\rangle;$
 $\quad \text{end}$

This code is used in section 751.

760. \langle Finish stream of page/form contents 760 $\rangle \equiv$
 $\text{pdf_end_text}; \text{ pdfshipoutend}(\textit{shipping_page}); \text{ pdf_end_stream}$

This code is used in section 759.

761. We need to write forms last, since the recursive call to *ship_out* would reset global state such as *pdfpagegroupval*, which is needed while writing images.

\langle Write out resource lists 761 $\rangle \equiv$
 \langle Write out pending raw objects 773 $\rangle;$
 \langle Write out pending images 779 $\rangle;$
 \langle Write out pending forms 775 \rangle

This code is used in section 759.

762. \langle Write out resources dictionary 762 $\rangle \equiv$
 $\text{pdf_begin_dict}(\text{pdf_last_resources}, 1); \langle$ Print additional resources 763 $\rangle;$
 \langle Generate font resources 766 $\rangle;$
 \langle Generate XObject resources 767 $\rangle;$
 \langle Generate ProcSet if desired 768 $\rangle;$
 pdf_end_dict

This code is used in section 759.

763. \langle Print additional resources [763](#) $\rangle \equiv$

```

if shipping_page then
  begin if pdf_page_resources  $\neq$  null then pdf_print_toks_ln(pdf_page_resources);
  end
else begin if obj_xform_resources(pdf_cur_form)  $\neq$  null then
  begin pdf_print_toks_ln(obj_xform_resources(pdf_cur_form));
  delete_toks(obj_xform_resources(pdf_cur_form));
  end;
end

```

This code is used in section [762](#).

764. In the end of shipping out a page we reset all the lists holding objects have been created during the page shipping.

```

define delete_toks(#)  $\equiv$ 
  begin delete_token_ref(#); #  $\leftarrow$  null;
  end

 $\langle$  Flush resource lists 764  $\rangle \equiv$ 
  flush_list(pdf_font_list); flush_list(pdf_obj_list); flush_list(pdf_xform_list); flush_list(pdf_ximage_list)

```

This code is used in section [759](#).

765. \langle Flush PDF mark lists [765](#) $\rangle \equiv$

```

flush_list(pdf_annot_list); flush_list(pdf_link_list); flush_list(pdf_dest_list); flush_list(pdf_bead_list)

```

This code is used in section [759](#).

766. \langle Generate font resources [766](#) $\rangle \equiv$

```

if pdf_font_list  $\neq$  null then
  begin pdf_print("/Font<<"); k  $\leftarrow$  pdf_font_list;
  while k  $\neq$  null do
    begin pdf_print("/Fn"); set_ff(info(k)); pdf_print_int(ff); pdf_print_resname_prefix; pdf_out(""");
    pdf_print_int(pdf_font_num[ff]); pdf_print("0R"); k  $\leftarrow$  link(k);
    end;
  pdf_print_ln(">>"); pdf_text_procset  $\leftarrow$  true;
  end

```

This code is used in section [762](#).

767. \langle Generate XObject resources [767](#) $\rangle \equiv$

```

if (pdf_xform_list  $\neq$  null)  $\vee$  (pdf_ximage_list  $\neq$  null) then
  begin pdf_print("/XObject<<"); k  $\leftarrow$  pdf_xform_list;
  while k  $\neq$  null do
    begin pdf_print("/Fmn"); pdf_print_int(obj_info(info(k))); pdf_print_resname_prefix; pdf_out(""");
    pdf_print_int(info(k)); pdf_print("0R"); k  $\leftarrow$  link(k);
    end;
  k  $\leftarrow$  pdf_ximage_list;
  while k  $\neq$  null do
    begin pdf_print("/Imn"); pdf_print_int(obj_info(info(k))); pdf_print_resname_prefix; pdf_out(""");
    pdf_print_int(info(k)); pdf_print("0R"); update_image_procset(obj_ximage_data(info(k)));
    k  $\leftarrow$  link(k);
    end;
  pdf_print_ln(">>");
```

This code is used in section [762](#).

768. ⟨Generate ProcSet if desired 768⟩ ≡
if (*pdf OMIT procset* < 0) ∨ ((*pdf OMIT procset* = 0) ∧ (*pdf major version* < 2)) **then**
begin *pdf print* (" /ProcSet [/PDF ");
if *pdf text procset* **then** *pdf print* (" /Text ");
if *check image b* (*pdf image procset*) **then** *pdf print* (" /ImageB ");
if *check image c* (*pdf image procset*) **then** *pdf print* (" /ImageC ");
if *check image i* (*pdf image procset*) **then** *pdf print* (" /ImageI ");
pdf print ln ("] ")
end

This code is used in section 762.

769. ⟨Write out page object 769⟩ ≡
pdf begin dict (*pdf last page*, 1); *pdf print ln* (" /Type /Page ");
pdf indirect ln (" Contents ", *pdf last stream*); *pdf indirect ln* (" Resources ", *pdf last resources*);
mediabox given ← false;
if *pdf page attr* ≠ null **then**
begin *s* ← *tokens to string* (*pdf page attr*); *mediabox given* ← *substr of str* (" /MediaBox ", *s*);
flush str (*s*);
end;
if ¬*mediabox given* **then**
begin *pdf print* (" /MediaBox [0 0] "); *pdf print mag bp* (*cur page width*); *pdf out* (" ");
pdf print mag bp (*cur page height*); *pdf print ln* ("] ");
end;
if *pdf page attr* ≠ null **then** *pdf print toks ln* (*pdf page attr*);
⟨Generate parent pages object 770⟩;
if *pdf page group val* > 0 **then**
begin *pdf print* (" /Group "); *pdf print int* (*pdf page group val*); *pdf print ln* (" 0 R ");
end;
⟨ Generate array of annotations or beads in page 771 ⟩;
pdf end dict

This code is used in section 759.

770. ⟨Generate parent pages object 770⟩ ≡
if *total pages mod pages tree kids max* = 1 **then**
begin *pdf create obj* (*obj type pages*, *pages tree kids max*); *pdf last pages* ← *obj ptr*;
end;
pdf indirect ln (" Parent ", *pdf last pages*)

This code is used in section 769.

771. ⟨ Generate array of annotations or beads in page 771 ⟩ ≡

```
if (pdf_annot_list ≠ null) ∨ (pdf_link_list ≠ null) then
    begin pdf_print("/Annots\u201c"); k ← pdf_annot_list;
    while k ≠ null do
        begin pdf_print_int(info(k)); pdf_print("0R"); k ← link(k);
        end;
    k ← pdf_link_list;
    while k ≠ null do
        begin pdf_print_int(info(k)); pdf_print("0R"); k ← link(k);
        end;
    pdf_print_ln("] ");
    end;
if pdf_bead_list ≠ null then
begin k ← pdf_bead_list; pdf_print("/B\u201c");
while k ≠ null do
    begin pdf_print_int(info(k)); pdf_print("0R"); k ← link(k);
    end;
pdf_print_ln("] ");
end
```

This code is used in section 769.

772. { Declare procedures needed in *pdf_hlist_out*, *pdf_vlist_out* 727 } +≡

```

procedure pdf_write_obj(n : integer); { write a raw PDF object }
  var s: str_number; f: byte_file;
  begin s ← tokens_to_string(obj_obj_data(n)); delete_toks(obj_obj_data(n));
  if obj_obj_is_stream(n) > 0 then
    begin pdf_begin_dict(n, 0);
    if obj_obj_stream_attr(n) ≠ null then
      begin pdf_print_toks_ln(obj_obj_stream_attr(n)); delete_toks(obj_obj_stream_attr(n));
      end;
    pdf_begin_stream;
    end
  else pdf_begin_obj(n, 1);
  if obj_obj_is_file(n) > 0 then
    begin cur_name ← s; cur_area ← ""; cur_ext ← ""; pack_cur_name;
    if ¬tex_b_openin(f) then
      begin print_nl("! "); print(s); print(" not found.");
      pdf_error("ext5", "cannot open file for embedding");
      end;
    print("<<"); print(s);
    if ¬eof(f) then
      begin { at least one byte available }
      while ¬eof(f) do pdf_out(getc(f));
      if (¬obj_obj_is_stream(n)) ∧ (pdf_ptr > 0) ∧ (pdf_buf[pdf_ptr - 1] ≠ 10) then pdf_out(10);
      end;
    print(">>"); b_close(f);
    end
  else if obj_obj_is_stream(n) > 0 then pdf_print(s)
  else pdf_print_ln(s);
  if obj_obj_is_stream(n) > 0 then pdf_end_stream
  else pdf_end_obj;
  flush_str(s);
  end;
procedure flush_whatsit_node(p : pointer; s : small_number);
  begin type(p) ← whatsit_node; subtype(p) ← s;
  if link(p) ≠ null then pdf_error("flush_whatsit_node", "link(p) is not null");
  flush_node_list(p);
  end;

```

773. { Write out pending raw objects 773 } ≡

```

if pdf_obj_list ≠ null then
  begin k ← pdf_obj_list;
  while k ≠ null do
    begin if ¬is_obj_written(info(k)) then pdf_write_obj(info(k));
    k ← link(k);
    end;
  end

```

This code is used in section 761.

774. { Global variables 13 } +≡

```

saved_pdf_cur_form: integer;

```

775. When flushing pending forms we need to save and restore resource lists (*pdf_font_list*, *pdf_obj_list*, *pdf_xform_list* and *pdf_ximage_list*), which are also used by page shipping.

```
< Write out pending forms 775 > ≡
  if pdf_xform_list ≠ null then
    begin k ← pdf_xform_list;
    while k ≠ null do
      begin if ¬is_obj_written(info(k)) then
        begin saved_pdf_cur_form ← pdf_cur_form; pdf_cur_form ← info(k); < Save resource lists 776 >;
        < Reset resource lists 753 >;
        pdf_ship_out(obj_xform_box(pdf_cur_form), false); pdf_cur_form ← saved_pdf_cur_form;
        < Restore resource lists 777 >;
        end;
      k ← link(k);
      end;
    end
```

This code is used in section 761.

776. < Save resource lists 776 > ≡

```
  save_font_list ← pdf_font_list; save_obj_list ← pdf_obj_list; save_xform_list ← pdf_xform_list;
  save_ximage_list ← pdf_ximage_list; save_text_procset ← pdf_text_procset;
  save_image_procset ← pdf_image_procset
```

This code is used in section 775.

777. < Restore resource lists 777 > ≡

```
  pdf_font_list ← save_font_list; pdf_obj_list ← save_obj_list; pdf_xform_list ← save_xform_list;
  pdf_ximage_list ← save_ximage_list; pdf_text_procset ← save_text_procset;
  pdf_image_procset ← save_image_procset
```

This code is used in section 775.

778. < Declare procedures needed in *pdf_hlist_out*, *pdf_vlist_out* 727 > +≡

```
procedure pdf_write_image(n : integer); { write an image }
  begin pdf_begin_dict(n, 0);
  if obj_ximage_attr(n) ≠ null then
    begin pdf_print_toks_ln(obj_ximage_attr(n)); delete_toks(obj_ximage_attr(n));
    end;
  if fixed_pdf_draftmode = 0 then write_image(obj_ximage_data(n));
  delete_image(obj_ximage_data(n));
  end;
```

779. < Write out pending images 779 > ≡

```
  if pdf_ximage_list ≠ null then
    begin k ← pdf_ximage_list;
    while k ≠ null do
      begin if ¬is_obj_written(info(k)) then pdf_write_image(info(k));
      k ← link(k);
      end;
    end
```

This code is used in section 761.

780. \langle Write out pending PDF marks [780](#) $\rangle \equiv$
 $\text{pdf_origin_h} \leftarrow 0; \text{pdf_origin_v} \leftarrow \text{cur_page_height}; \langle$ Write out PDF annotations [781](#) $\rangle;$
 \langle Write out PDF link annotations [782](#) $\rangle;$
 \langle Write out PDF mark destinations [784](#) $\rangle;$
 \langle Write out PDF bead rectangle specifications [786](#) \rangle

This code is used in section [759](#).

781. \langle Write out PDF annotations [781](#) $\rangle \equiv$
if $\text{pdf_annot_list} \neq \text{null}$ **then**
begin $k \leftarrow \text{pdf_annot_list};$
while $k \neq \text{null}$ **do**
begin $i \leftarrow \text{obj_annot_ptr}(\text{info}(k)); \{ i \text{ points to } \text{pdf_annot_node} \}$
 $\text{pdf_begin_dict}(\text{info}(k), 1); \text{pdf_print_ln}("/\text{Type}_{\square}/\text{Annot}"); \text{pdf_print_toks_ln}(\text{pdf_annot_data}(i));$
 $\text{pdf_rectangle}(\text{pdf_left}(i), \text{pdf_top}(i), \text{pdf_right}(i), \text{pdf_bottom}(i)); \text{pdf_end_dict}; k \leftarrow \text{link}(k);$
end;
end

This code is used in section [780](#).

782. \langle Write out PDF link annotations [782](#) $\rangle \equiv$
if $\text{pdf_link_list} \neq \text{null}$ **then**
begin $k \leftarrow \text{pdf_link_list};$
while $k \neq \text{null}$ **do**
begin $i \leftarrow \text{obj_annot_ptr}(\text{info}(k)); \text{pdf_begin_dict}(\text{info}(k), 1); \text{pdf_print_ln}("/\text{Type}_{\square}/\text{Annot}");$
if $\text{pdf_action_type}(\text{pdf_link_action}(i)) \neq \text{pdf_action_user}$ **then** $\text{pdf_print_ln}("/\text{Subtype}_{\square}/\text{Link}");$
if $\text{pdf_link_attr}(i) \neq \text{null}$ **then** $\text{pdf_print_toks_ln}(\text{pdf_link_attr}(i));$
 $\text{pdf_rectangle}(\text{pdf_left}(i), \text{pdf_top}(i), \text{pdf_right}(i), \text{pdf_bottom}(i));$
if $\text{pdf_action_type}(\text{pdf_link_action}(i)) \neq \text{pdf_action_user}$ **then** $\text{pdf_print}("/\text{A}_{\square}");$
 $\text{write_action}(\text{pdf_link_action}(i)); \text{pdf_end_dict}; k \leftarrow \text{link}(k);$
end;
 \langle Flush $\text{pdf_start_link_node}'s$ created by append_link [783](#) $\rangle;$
end

This code is used in section [780](#).

783. \langle Flush $\text{pdf_start_link_node}'s$ created by append_link [783](#) $\rangle \equiv$
 $k \leftarrow \text{pdf_link_list};$
while $k \neq \text{null}$ **do**
begin $i \leftarrow \text{obj_annot_ptr}(\text{info}(k)); \{ \text{nodes with } \text{info} = \text{max_halfword} \text{ were created by } \text{append_link}$
 $\text{and must be flushed here, as they are not linked in any list} \}$
if $\text{info}(i) = \text{max_halfword}$ **then** $\text{flush_whatsit_node}(i, \text{pdf_start_link_node});$
 $k \leftarrow \text{link}(k);$
end

This code is used in section [782](#).

784. ⟨ Write out PDF mark destinations 784 ⟩ ≡

```

if pdf_dest_list ≠ null then
begin k ← pdf_dest_list;
while k ≠ null do
begin if is_obj_written(info(k)) then
    pdf_error("ext5", "destination_has_been_written_(this_shouldn't_happen)");
else begin i ← obj_dest_ptr(info(k));
if (pdf_dest_named_id(i) > 0) ∧ (pdf_dest_objnum(i) = null) then
    begin pdf_begin_dict(info(k), 1); pdf_print("/D");
    end
else pdf_begin_obj(info(k), 1);
pdf_out("[");

if pdf_dest_objnum(i) = null then pdf_print_int(pdf_last_page)
else pdf_print_int(pdf_dest_objnum(i));
pdf_print("O[R");

case pdf_dest_type(i) of
pdf_dest_xyz: begin pdf_print("/XYZ"); pdf_print_mag_bp(pdf_x(pdf_left(i))); pdf_out("u");
pdf_print_mag_bp(pdf_y(pdf_top(i))); pdf_out("u");
if pdf_dest_xyZoom(i) = null then pdf_print("null")
else begin pdf_print_int((pdf_dest_xyZoom(i) div 1000); pdf_out(".");
pdf_print_int((pdf_dest_xyZoom(i) mod 1000));
end;
end;
pdf_dest_fit: pdf_print("/Fit");
pdf_dest_fith: begin pdf_print("/FitH"); pdf_print_mag_bp(pdf_y(pdf_top(i)));
end;
pdf_dest_fitv: begin pdf_print("/FitV"); pdf_print_mag_bp(pdf_x(pdf_left(i)));
end;
pdf_dest_fitb: pdf_print("/FitB");
pdf_dest_fitbh: begin pdf_print("/FitBH"); pdf_print_mag_bp(pdf_y(pdf_top(i)));
end;
pdf_dest_fitbv: begin pdf_print("/FitBV"); pdf_print_mag_bp(pdf_x(pdf_left(i)));
end;
pdf_dest_fitr: begin pdf_print("/FitR"); pdf_print_rect_spec(i);
end;
othercases pdf_error("ext5", "unknown_dest_type");
endcases; pdf_print_ln("]");
if (pdf_dest_named_id(i) > 0) ∧ (pdf_dest_objnum(i) = null) then pdf_end_dict
else pdf_end_obj;
end;
k ← link(k);
end;
end

```

This code is used in section 780.

785. ⟨ Declare procedures needed in pdf_hlist_out, pdf_vlist_out 727 ⟩ +≡

```

procedure pdf_print_rect_spec(r : pointer); { prints a rect spec }
begin pdf_print_mag_bp(pdf_x(pdf_left(r))); pdf_out("u"); pdf_print_mag_bp(pdf_y(pdf_bottom(r)));
pdf_out("u"); pdf_print_mag_bp(pdf_x(pdf_right(r))); pdf_out("u"); pdf_print_mag_bp(pdf_y(pdf_top(r)));
end;

```

786. ⟨ Write out PDF bead rectangle specifications 786 ⟩ ≡

```

if pdf_bead_list ≠ null then
  begin k ← pdf_bead_list;
  while k ≠ null do
    begin pdf_new_obj(obj_type_others, 0, 1); pdf_out("["); i ← obj_bead_data(info(k));
      { pointer to a whatsit or whatsit-like node }
      pdf_print_rect_spec(i);
      if info(i) = max_halfword then { not a whatsit node, so must be destroyed here }
        flush_whatsit_node(i, pdf_start_thread_node);
      pdf_print_ln("]"); obj_bead_rect(info(k)) ← obj_ptr; { rewrite obj_bead_data }
      pdf_end_obj; k ← link(k);
      end;
    end;
  end

```

This code is used in section 780.

787. In the end we must flush PDF objects that cannot be written out immediately after shipping out pages.

788. ⟨ Output outlines 788 ⟩ ≡

```

if pdf_first_outline ≠ 0 then
  begin pdf_new_dict(obj_type_others, 0, 1); outlines ← obj_ptr; l ← pdf_first_outline; k ← 0;
  repeat incr(k); a ← open_subentries(l);
    if obj_outline_count(l) > 0 then k ← k + a;
    obj_outline_parent(l) ← obj_ptr; l ← obj_outline_next(l);
  until l = 0;
  pdf_print_ln("/Type\U00d7/Outlines"); pdf_indirect_ln("First", pdf_first_outline);
  pdf_indirect_ln("Last", pdf_last_outline); pdf_int_entry_ln("Count", k); pdf_end_dict;
  { Output PDF outline entries 789 };
  end
else outlines ← 0

```

This code is used in section 794.

```

789.  ⟨Output PDF outline entries 789⟩ ≡
  k ← head_tab[obj_type_outline];
  while k ≠ 0 do
    begin if obj_outline_parent(k) = pdf_parent_outline then
      begin if obj_outline_prev(k) = 0 then pdf_first_outline ← k;
      if obj_outline_next(k) = 0 then pdf_last_outline ← k;
      end;
      pdf_begin_dict(k, 1); pdf_indirect_ln("Title", obj_outline_title(k));
      pdf_indirect_ln("A", obj_outline_action_objnum(k));
      if obj_outline_parent(k) ≠ 0 then pdf_indirect_ln("Parent", obj_outline_parent(k));
      if obj_outline_prev(k) ≠ 0 then pdf_indirect_ln("Prev", obj_outline_prev(k));
      if obj_outline_next(k) ≠ 0 then pdf_indirect_ln("Next", obj_outline_next(k));
      if obj_outline_first(k) ≠ 0 then pdf_indirect_ln("First", obj_outline_first(k));
      if obj_outline_last(k) ≠ 0 then pdf_indirect_ln("Last", obj_outline_last(k));
      if obj_outline_count(k) ≠ 0 then pdf_int_entry_ln("Count", obj_outline_count(k));
      if obj_outline_attr(k) ≠ 0 then
        begin pdf_print_toks_ln(obj_outline_attr(k)); delete_toks(obj_outline_attr(k));
        end;
      pdf_end_dict; k ← obj_link(k);
    end

```

This code is used in section 788.

```

790.  ⟨Output article threads 790⟩ ≡
  if head_tab[obj_type_thread] ≠ 0 then
    begin pdf_new_obj(obj_type_others, 0, 1); threads ← obj_ptr; pdf_out("[");
    k ← head_tab[obj_type_thread];
    while k ≠ 0 do
      begin pdf_print_int(k); pdf_print("□0□R□"); k ← obj_link(k);
      end;
    remove_last_space; pdf_print_ln("]"); pdf_end_obj; k ← head_tab[obj_type_thread];
    while k ≠ 0 do
      begin out_thread(k); k ← obj_link(k);
      end;
    end
  else threads ← 0

```

This code is used in section 794.

791. Now we are ready to declare our new procedure *ship_out*. It will call *pdf_ship_out* if the integer parameter *pdf_output* is positive; otherwise it will call *dvi_ship_out*, which is the TeX original *ship_out*.

```

procedure ship_out(p : pointer); { output the box p }
  begin fix_pdfoutput;
  if pdf_output > 0 then pdf_ship_out(p, true)
  else dvi_ship_out(p);
  end;

```

792. \langle Initialize variables for PDF output 792 $\rangle \equiv$

```
check_pdfversion; prepare_mag; fixed_decimal_digits ← fix_int(pdf_decimal_digits, 0, 4);
min_bp_val ← divide_scaled(one_hundred_bp, ten_pow[fixed_decimal_digits + 2], 0);
if pdf_pk_resolution = 0 then { if not set from format file or by user }
  pdf_pk_resolution ← pk_dpi; { take it from texmf.cnf }
  fixed_pk_resolution ← fix_int(pdf_pk_resolution, 72, 8000);
  pk_scale_factor ← divide_scaled(72, fixed_pk_resolution, 5 + fixed_decimal_digits);
if pdf_pk_mode ≠ null then
  begin kpse_init_prog(`PDFTEX', fixed_pk_resolution, make_cstring(tokens_to_string(pdf_pk_mode)), nil);
  flush_string;
  end
else kpse_init_prog(`PDFTEX', fixed_pk_resolution, nil, nil);
kpse_set_program_enabled(kpse_pk_format, 1, kpse_src_compile); set_job_id(year, month, day, time);
if (pdf_unique_resname > 0) ∧ (pdf_resname_prefix = 0) then pdf_resname_prefix ← get_resname_prefix
```

This code is used in section 750.

793. Finishing the PDF output file.

The following procedures sort the table of destination names.

```

define get_next_char(#) ≡ c ← str_pool[j] incr(j);
  if (c = 92) ∧ (j < e) then
    begin c ← str_pool[j]; incr(j);
    if (c ≥ 48) ∧ (c ≤ 55) then
      begin c ← c - 48;
      if (j < e) ∧ (str_pool[j] ≥ 48) ∧ (str_pool[j] ≤ 55) then
        begin c ← 8 * c + str_pool[j] - 48; incr(j);
        if (j < e) ∧ (str_pool[j] ≥ 48) ∧ (str_pool[j] ≤ 55) ∧ (c < 32) then
          begin c ← 8 * c + str_pool[j] - 48; incr(j);
          end;
        end;
      end;
    else begin case c of
      98: c ← 8; { 'b': backspace }
      102: c ← 12; { 'f': form feed }
      110: c ← 10; { 'n': line feed }
      114: c ← 13; { 'r': carriage return }
      116: c ← 9; { 't': horizontal tab }
      { nothing to do for '\', '(', ')' }
    othercases do_nothing
    endcases;
  end;
end

function str_less_str(s1, s2 : str_number): boolean; { compare two pdf strings }
  var j1, j2, e1, e2: pool_pointer; c1, c2: packed_ASCII_code;
  begin { Minimal requirement: output of \pdfescapestring must be supported. }
    { This implementation also supports all escape sequences }
    { listed in the table 'Escape sequences in literal strings' }
    { of the pdf specification. }
    { End-of-line markers are not detected: }
    { The marker is not replaced by '\n' or removed if it is escaped. }
    j1 ← str_start[s1]; j2 ← str_start[s2]; e1 ← j1 + length(s1); e2 ← j2 + length(s2);
    while (j1 < e1) ∧ (j2 < e2) do
      begin { get next character of first string }
        get_next_char(1); { get next character of second string }
        get_next_char(2); { compare characters }
        if c1 < c2 then
          begin str_less_str ← true; return;
          end
        else if c1 > c2 then
          begin str_less_str ← false; return;
          end;
        end; { compare string lengths }
        if (j1 ≥ e1) ∧ (j2 < e2) then str_less_str ← true
        else str_less_str ← false;
    exit: end;
  procedure sort_dest_names(l, r : integer); { sorts dest_names by names }
    var i, j: integer; s: str_number; e: dest_name_entry;
    begin i ← l; j ← r; s ← dest_names[l + r] div 2].objname;
    repeat while str_less_str(dest_names[i].objname, s) do incr(i);

```

```

while str_less_str(s, dest_names[j].objname) do decr(j);
if i ≤ j then
  begin e ← dest_names[i]; dest_names[i] ← dest_names[j]; dest_names[j] ← e; incr(i); decr(j);
  end;
until i > j;
if l < j then sort_dest_names(l, j);
if i < r then sort_dest_names(i, r);
end;

```

794. Now the finish of PDF output file. At this moment all Page objects are already written completely to PDF output file.

```

⟨Finish the PDF file 794⟩ ≡
  if total_pages = 0 then
    begin print_nl("No\u00a9pages\u00a9of\u00a9output .");
    if pdf_gone > 0 then garbage_warning;
    end
  else begin if fixed_pdf_draftmode = 0 then
    begin pdf_flush; { to make sure that the output file name has been already created }
    if total_pages mod pages_tree.kids_max ≠ 0 then
      obj_info(pdf_last_pages) ← total_pages mod pages_tree.kids_max;
      { last pages object may have less than pages_tree.kids_max children }
    flush_jbig2_page0_objects; { flush page 0 objects from JBIG2 images, if any }
    ⟨Check for non-existing pages 799⟩;
    ⟨Reverse the linked list of Page and Pages objects 800⟩;
    ⟨Check for non-existing destinations 796⟩;
    ⟨Check for non-existing structure destinations 798⟩;
    ⟨Output fonts definition 801⟩;
    ⟨Output pages tree 802⟩;
    ⟨Output outlines 788⟩;
    ⟨Output name tree 804⟩;
    ⟨Output article threads 790⟩;
    ⟨Output the catalog object 806⟩;
    if pdf OMIT_INFO_DICT = 0 then pdf.print_info; { last candidate for object stream }
    if pdf.OS_ENABLE then
      begin pdf.OS_SWITCH(true); pdf.OS_WRITE_OBJSTREAM; pdf_FLUSH; pdf.OS_SWITCH(false);
      ⟨Output the cross-reference stream dictionary 814⟩;
      pdf_FLUSH;
      end
    else begin ⟨Output the obj-tab 813⟩;
    end;
    ⟨Output the trailer 815⟩;
    pdf_FLUSH; print_nl("Output\u00a9written\u00a9on\u00a9"); print_file_name(0, output_file_name, 0); print(" \u2225(");
    print_int(total_pages); print(" \u2225page");
    if total_pages ≠ 1 then print_char("s");
    print(" ,\u2225"); print_int(pdf_offset); print(" \u2225bytes) . ");
    end;
  libpdffinish;
  if fixed_pdf_draftmode = 0 then b_CLOSE(pdf_file)
  else pdf_warning(0, "\pdfdraftmode\u00a9enabled,\u00a9not\u00a9changing\u00a9output\u00a9pdf", true, true)
end

```

This code is used in section 1513.

795. Destinations that have been referenced but don't exists have `obj_dest_ptr = null`. Leaving them undefined might cause troubles for PDF browsers, so we need to fix them.

```
procedure pdf_fix_dest(k : integer);
begin if obj_dest_ptr(k) ≠ null then return;
pdf_warning("dest", "", true, false);
if obj_info(k) < 0 then
begin print("name{"); print(-obj_info(k)); print("}");
end
else begin print("num"); print_int(obj_info(k));
end;
print("has been referenced but does not exist, replaced by a fixed one");
print_ln; pdf_begin_obj(k, 1); pdf_out("["); pdf_print_int(head_tab[obj_type_page]);
pdf_print_ln("O|R/Fit]"); pdf_end_obj;
end;
```

796. ⟨Check for non-existing destinations 796⟩ ≡

```
k ← head_tab[obj_type_dest];
while k ≠ 0 do
begin pdf_fix_dest(k); k ← obj_link(k);
end
```

This code is used in section 794.

797. The same for structure destinations, except that there is no sensible default object to point to.

```
procedure pdf_fix_struct_dest(k : integer);
begin if obj_dest_ptr(k) ≠ null then return;
pdf_warning("structure dest", "", false, false);
if obj_info(k) < 0 then
begin print("name{"); print(-obj_info(k)); print("}");
end
else begin print("num"); print_int(obj_info(k));
end;
print("has been referenced but does not exist");
print_ln; print_ln; pdf_begin_obj(k, 1);
pdf_out("["); pdf_print_int(head_tab[obj_type_page]); pdf_print_ln("O|R/Fit]");
pdf_end_obj;
end;
```

798. ⟨Check for non-existing structure destinations 798⟩ ≡

```
k ← head_tab[obj_type_struct_dest];
while k ≠ 0 do
begin pdf_fix_struct_dest(k); k ← obj_link(k);
end
```

This code is used in section 794.

799. ⟨Check for non-existing pages 799⟩ ≡

```
k ← head_tab[obj_type_page];
while obj_aux(k) = 0 do
begin pdf_warning("dest", "Page", true, false); print_int(obj_info(k));
print("has been referenced but does not exist!");
print_ln; print_ln; k ← obj_link(k);
end;
head_tab[obj_type_page] ← k
```

This code is used in section 794.

800. \langle Reverse the linked list of Page and Pages objects 800 $\rangle \equiv$
 $k \leftarrow head_tab[obj_type_page]; l \leftarrow 0;$
repeat $i \leftarrow obj_link(k); obj_link(k) \leftarrow l; l \leftarrow k; k \leftarrow i;$
until $k = 0;$
 $head_tab[obj_type_page] \leftarrow l; k \leftarrow head_tab[obj_type_pages]; pages_tail \leftarrow k; l \leftarrow 0;$
repeat $i \leftarrow obj_link(k); obj_link(k) \leftarrow l; l \leftarrow k; k \leftarrow i;$
until $k = 0;$
 $head_tab[obj_type_pages] \leftarrow l$

This code is used in section 794.

801. \langle Output fonts definition 801 $\rangle \equiv$
for $k \leftarrow font_base + 1$ **to** $font_ptr$ **do**
if $font_used[k] \wedge hasfmentry(k) \wedge (pdf_font_num[k] < 0)$ **then**
begin $i \leftarrow -pdf_font_num[k]$; $pdfassert(pdf_font_num[i] > 0);$
for $j \leftarrow 0$ **to** 255 **do**
if $pdf_char_marked(k, j)$ **then** $pdf_mark_char(i, j);$
if $(length(pdf_font_attr[i]) = 0) \wedge (length(pdf_font_attr[k]) \neq 0)$ **then**
 $pdf_font_attr[i] \leftarrow pdf_font_attr[k]$
else if $(length(pdf_font_attr[k]) = 0) \wedge (length(pdf_font_attr[i]) \neq 0)$ **then**
 $pdf_font_attr[k] \leftarrow pdf_font_attr[i]$
else if $(length(pdf_font_attr[i]) \neq 0) \wedge (length(pdf_font_attr[k]) \neq 0) \wedge \neg str_eq_str(pdf_font_attr[i], pdf_font_attr[k])$ **then**
begin $pdf_warning("\backslash pdffontattr", "fonts_\wedge", true, false); print_font_identifier(i);$
 $print(" \wedge and \wedge"); print_font_identifier(k);$
 $print(" \wedge have \wedge conflicting \wedge attributes; I \wedge will \wedge ignore \wedge the \wedge attributes \wedge assigned \wedge to \wedge ");$
 $print_font_identifier(i); print_ln; print_ln;$
end;
end;
 $fixed_gen_tounicode \leftarrow pdf_gen_tounicode; k \leftarrow head_tab[obj_type_font];$
while $k \neq 0$ **do**
begin $f \leftarrow obj_info(k); pdfassert(pdf_font_num[f] > 0); do_pdf_font(k, f); k \leftarrow obj_link(k);$
end;
write_fontstuff

This code is used in section 794.

802. We will generate in each single step the parents of all Pages/Page objects in the previous level. These new generated Pages object will create a new level of the Pages tree. We will repeat this until we have only one Pages object. This one will be the Root object.

$\langle \text{Output pages tree } 802 \rangle \equiv$

```

 $a \leftarrow \text{sys\_obj\_ptr} + 1;$ 
    { all Pages objects whose children are not Page objects should have index greater than  $a$  }
 $l \leftarrow \text{head\_tab}[\text{obj\_type\_pages}]$ ; {  $l$  is the index of current Pages object which is being output }
 $k \leftarrow \text{head\_tab}[\text{obj\_type\_page}]$ ; {  $k$  is the index of current child of  $l$  }
 $b \leftarrow 0;$ 
repeat  $i \leftarrow 0$ ; { counter of Pages object in current level }
 $c \leftarrow 0$ ; { first Pages object in previous level }
if  $\text{obj\_link}(l) = 0$  then  $\text{is\_root} \leftarrow \text{true}$ 
    { only Pages object; total pages is not greater than  $\text{pages\_tree\_kids\_max}$  }
else  $\text{is\_root} \leftarrow \text{false}$ ;
repeat if  $\neg \text{is\_root}$  then
begin if  $i \bmod \text{pages\_tree\_kids\_max} = 0$  then
begin { create a new Pages object for next level }
 $\text{pdf\_last\_pages} \leftarrow \text{pdf\_new\_objnum};$ 
if  $c = 0$  then  $c \leftarrow \text{pdf\_last\_pages};$ 
 $\text{obj\_link}(\text{pages\_tail}) \leftarrow \text{pdf\_last\_pages}; \text{pages\_tail} \leftarrow \text{pdf\_last\_pages}; \text{obj\_link}(\text{pdf\_last\_pages}) \leftarrow 0;$ 
 $\text{obj\_info}(\text{pdf\_last\_pages}) \leftarrow \text{obj\_info}(l);$ 
end
else  $\text{obj\_info}(\text{pdf\_last\_pages}) \leftarrow \text{obj\_info}(\text{pdf\_last\_pages}) + \text{obj\_info}(l);$ 
end;
{ Output the current Pages object in this level 803 };
 $\text{incr}(i); l \leftarrow \text{obj\_link}(l);$ 
until ( $l = c$ );
 $b \leftarrow c;$ 
if  $l = 0$  then goto done;
until false;
done:
```

This code is used in section 794.

803. $\langle \text{Output the current Pages object in this level } 803 \rangle \equiv$

```

 $\text{pdf\_begin\_dict}(l, 1); \text{pdf\_print\_ln}("/\text{Type}\_\text{Pages}"); \text{pdf\_int\_entry\_ln}("Count", \text{obj\_info}(l));$ 
if  $\neg \text{is\_root}$  then  $\text{pdf\_indirect\_ln}("Parent", \text{pdf\_last\_pages});$ 
 $\text{pdf\_print}("/\text{Kids}\_\text{["}); j \leftarrow 0;$ 
repeat  $\text{pdf\_print\_int}(k); \text{pdf\_print}("\text{\_0\_\R}\text{["}); k \leftarrow \text{obj\_link}(k); \text{incr}(j);$ 
until  $((l < a) \wedge (j = \text{obj\_info}(l))) \vee (k = 0) \vee ((k = b) \wedge (b \neq 0)) \vee (j = \text{pages\_tree\_kids\_max});$ 
remove\_last\_space;  $\text{pdf\_print\_ln}("]\text{ "});$ 
if  $k = 0$  then
begin  $k \leftarrow \text{head\_tab}[\text{obj\_type\_pages}]$ ;  $\text{head\_tab}[\text{obj\_type\_pages}] \leftarrow 0;$ 
end;
if  $\text{is\_root} \wedge (\text{pdf\_pages\_attr} \neq \text{null})$  then  $\text{pdf\_print\_toks\_ln}(\text{pdf\_pages\_attr});$ 
 $\text{pdf\_end\_dict};$ 
```

This code is used in section 802.

804. The name tree is very similar to Pages tree so its construction should be certain from Pages tree construction. For intermediate node *obj_info* will be the first name and *obj_link* will be the last name in \limits array. Note that *pdf_dest_names_ptr* will be less than *obj_ptr*, so we test if $k < \text{pdf_dest_names_ptr}$ then *k* is index of leaf in *dest_names*; else *k* will be index in *obj_tab* of some intermediate node.

```

⟨Output name tree 804⟩ ≡
  if pdf_dest_names_ptr = 0 then
    begin dests ← 0; goto done1;
    end;
  sort_dest_names(0, pdf_dest_names_ptr − 1); names_head ← 0; names_tail ← 0; k ← 0;
  { index of current child of l; if k < pdf_dest_names_ptr then this is pointer to dest_names array;
  otherwise it is the pointer to obj_tab (object number)}
  is_names ← true; { flag whether Names or Kids }
  b ← 0;
  repeat repeat pdf_create_obj(obj_type_others, 0); { create a new node }
    l ← obj_ptr;
    if b = 0 then b ← l; { first in this level }
    if names_head = 0 then
      begin names_head ← l; names_tail ← l;
      end
    else begin obj_link(names_tail) ← l; names_tail ← l;
      end;
    obj_link(names_tail) ← 0; ⟨Output the current node in this level 805⟩;
  until b = 0;
  if k = l then
    begin dests ← l; goto done1;
    end;
  until false;
done1: if (dests ≠ 0) ∨ (pdf_names_toks ≠ null) then
  begin pdf_new_dict(obj_type_others, 0, 1);
  if (dests ≠ 0) then pdf_indirect_ln("Dest", dests);
  if pdf_names_toks ≠ null then
    begin pdf_print_toks_ln(pdf_names_toks); delete_toks(pdf_names_toks);
    end;
  pdf_end_dict; names_tree ← obj_ptr;
  end
  else names_tree ← 0

```

This code is used in section 794.

805. \langle Output the current node in this level 805 $\rangle \equiv$

```

pdf_begin_dict(l, 1); j ← 0;
if is_names then
begin obj_info(l) ← dest_names[k].objname; pdf_print("/Names[");
repeat pdf_print_str(dest_names[k].objname); pdf_out("]"); pdf_print_int(dest_names[k].objnum);
    pdf_print("L0LR"); incr(j); incr(k);
until (j = name_tree_kids_max) ∨ (k = pdf_dest_names_ptr);
remove_last_space; pdf_print_ln("]");
obj_aux(l) ← dest_names[k - 1].objname;
if k = pdf_dest_names_ptr then
begin is_names ← false; k ← names_head; b ← 0;
end;
end
else begin obj_info(l) ← obj_info(k); pdf_print("/Kids[");
repeat pdf_print_int(k); pdf_print("L0LR"); incr(j); obj_aux(l) ← obj_aux(k); k ← obj_link(k);
until (j = name_tree_kids_max) ∨ (k = b) ∨ (obj_link(k) = 0);
remove_last_space; pdf_print_ln("]");
if k = b then b ← 0;
end;
pdf_print("/Limits["); pdf_print_str(obj_info(l)); pdf_out("]"); pdf_print_str(obj_aux(l));
pdf_print_ln("]"); pdf_end_dict;
```

This code is used in section 804.

806. \langle Output the catalog object 806 $\rangle \equiv$

```

pdf_new_dict(obj_type_others, 0, 1); root ← obj_ptr; pdf_print_ln("/Type/Catalog");
pdf_indirect_ln("Pages", pdf_last_pages);
if threads ≠ 0 then pdf_indirect_ln("Threads", threads);
if outlines ≠ 0 then pdf_indirect_ln("Outlines", outlines);
if names_tree ≠ 0 then pdf_indirect_ln("Names", names_tree);
if pdf_catalog_toks ≠ null then
begin pdf_print_toks_ln(pdf_catalog_toks); delete_toks(pdf_catalog_toks);
end;
if pdf_catalog_openaction ≠ 0 then pdf_indirect_ln("OpenAction", pdf_catalog_openaction);
pdf_end_dict
```

This code is used in section 794.

807. If the same keys in a dictionary are given several times, then it is not defined which value is chosen by an application. Therefore the keys */Producer* and */Creator* are only set if the token list *pdf_info_toks* converted to a string does not contain these key strings.

```

procedure pdf_print_info; { print info object }
  var s: str_number;
    creator_given, producer_given, creationdate_given, moddate_given, trapped_given: boolean;
  begin pdf_new_dict(obj_type_others, 0, 3); { keep Info readable unless explicitly forced }
    creator_given ← false; producer_given ← false; creationdate_given ← false; moddate_given ← false;
    trapped_given ← false;
    if pdf_info_toks ≠ null then
      begin s ← tokens_to_string(pdf_info_toks); creator_given ← substr_of_str("/Creator", s);
      producer_given ← substr_of_str("/Producer", s);
      creationdate_given ← substr_of_str("/CreationDate", s);
      moddate_given ← substr_of_str("/ModDate", s); trapped_given ← substr_of_str("/Trapped", s);
      end;
    if ¬producer_given then
      begin ⟨Print the Producer key 808⟩;
      end;
    if pdf_info_toks ≠ null then
      begin if length(s) > 0 then
        begin pdf_print_ln(s);
        end;
      flush_str(s); delete_toks(pdf_info_toks);
      end;
    if ¬creator_given then pdf_str_entry_ln("Creator", "TeX");
    if pdf_info OMIT_DATE = 0 then
      begin if ¬creationdate_given then
        begin ⟨Print the CreationDate key 809⟩;
        end;
      if ¬moddate_given then
        begin ⟨Print the ModDate key 810⟩;
        end;
      end;
    if ¬trapped_given then
      begin pdf_print_ln("/Trapped ↴/False");
      end;
    if pdf_suppress_ptex_info mod 2 = 0 then
      begin if get_ptex_use_underscore then pdf_str_entry_ln("PTEX_Fullbanner", pdftex_banner)
      else pdf_str_entry_ln("PTEX.Fullbanner", pdftex_banner);
      end;
  pdf_end_dict;
end;
```

808. ⟨Print the Producer key 808⟩ ≡

```

pdf_print("/Producer ↴(pdfTeX-"); pdf_print_int(pdftex_version div 100); pdf_out(".");
pdf_print_int(pdftex_version mod 100); pdf_out("."); pdf_print(pdftex_revision); pdf_print_ln(")")
```

This code is used in section 807.

809. ⟨Print the CreationDate key 809⟩ ≡

```
print_creation_date;
```

This code is used in section 807.

810. ⟨ Print the ModDate key 810 ⟩ ≡
 $print_mod_date;$

This code is used in section 807.

811. ⟨ Global variables 13 ⟩ +≡
 $pdftex_banner: str_number; \{ the complete banner \}$

812. ⟨ Build a linked list of free objects 812 ⟩ ≡
 $l \leftarrow 0; set_obj_fresh(l); \{ null object at begin of list of free objects \}$
for $k \leftarrow 1$ **to** sys_obj_ptr **do**
 if $\neg is_obj_written(k)$ **then**
 begin $obj_link(l) \leftarrow k; l \leftarrow k;$
 end;
 $obj_link(l) \leftarrow 0$

This code is used in sections 813 and 814.

813. ⟨ Output the obj_tab 813 ⟩ ≡
⟨ Build a linked list of free objects 812 ⟩;
 $pdf_save_offset \leftarrow pdf_offset; pdf_print_ln("xref"); pdf_print("0\u20a9"); pdf_print_int_ln(obj_ptr + 1);$
 $pdf_print_fw_int(obj_link(0), 10); pdf_print_ln("\u20a965535\u20a9f\u20a9");$
for $k \leftarrow 1$ **to** obj_ptr **do**
 begin if $\neg is_obj_written(k)$ **then**
 begin $pdf_print_fw_int(obj_link(k), 10); pdf_print_ln("\u20a900000\u20a9f\u20a9");$
 end
 else begin $pdf_print_fw_int(obj_offset(k), 10); pdf_print_ln("\u20a900000\u20a9n\u20a9");$
 end;
 end

This code is used in section 794.

814. \langle Output the cross-reference stream dictionary [814](#) $\rangle \equiv$

```

pdf_new_dict(obj_type_others, 0, 0);
if ((obj_offset(sys_obj_ptr)/256) > 16777215) then xref_offset_width ← 5
else if obj_offset(sys_obj_ptr) > 16777215 then xref_offset_width ← 4
else if obj_offset(sys_obj_ptr) > 65535 then xref_offset_width ← 3
else xref_offset_width ← 2;
⟨ Build a linked list of free objects 812 ⟩;
pdf_print_ln("/TypeU/XRef"); pdf_print("/IndexU[0U"); pdf_print_int(obj_ptr + 1); pdf_print_ln("] ");
pdf_int_entry_ln("Size", obj_ptr + 1); pdf_print("/WU[1U"); pdf_print_int(xref_offset_width);
pdf_print_ln("]1]"); pdf_indirect_ln("Root", root);
if pdf OMIT info dict = 0 then pdf_indirect_ln("Info", obj_ptr - 1);
if pdf_trailer_toks ≠ null then
begin pdf_print_toks_ln(pdf_trailer_toks); delete_toks(pdf_trailer_toks);
end;
if pdf_trailer_id_toks ≠ null then print_ID_alt(pdf_trailer_id_toks)
else print_ID(output_file_name);
pdf_print_nl; pdf_begin_stream;
for k ← 0 to sys_obj_ptr do
begin if ¬is_obj_written(k) then
begin { a free object }
pdf_out(0); pdf_out_bytes(obj_link(k), xref_offset_width); pdf_out(255);
end
else begin if obj_os_idx(k) = -1 then
begin { object not in object stream }
pdf_out(1); pdf_out_bytes(obj_offset(k), xref_offset_width); pdf_out(0);
end
else begin { object in object stream }
pdf_out(2); pdf_out_bytes(obj_offset(k), xref_offset_width); pdf_out(obj_os_idx(k));
end;
end;
end;
pdf_end_stream;

```

This code is used in section [794](#).

815. \langle Output the trailer [815](#) $\rangle \equiv$

```

if ¬pdf_os_enable then
begin pdf_print_ln("trailer"); pdf_print("<<U"); pdf_int_entry_ln("Size", sys_obj_ptr + 1);
pdf_indirect_ln("Root", root);
if pdf OMIT info dict = 0 then pdf_indirect_ln("Info", sys_obj_ptr);
if pdf_trailer_toks ≠ null then
begin pdf_print_toks_ln(pdf_trailer_toks); delete_toks(pdf_trailer_toks);
end;
if pdf_trailer_id_toks ≠ null then print_ID_alt(pdf_trailer_id_toks)
else print_ID(output_file_name);
pdf_print_ln("]>>");
end;
pdf_print_ln("startxref");
if pdf_os_enable then pdf_print_int_ln(obj_offset(sys_obj_ptr))
else pdf_print_int_ln(pdf_save_offset);
pdf_print_ln("%EOF")

```

This code is used in section [794](#).

816. Packaging. We're essentially done with the parts of TeX that are concerned with the input (*get_next*) and the output (*ship_out*). So it's time to get heavily into the remaining part, which does the real work of typesetting.

After lists are constructed, TeX wraps them up and puts them into boxes. Two major subroutines are given the responsibility for this task: *hpack* applies to horizontal lists (hlists) and *vpack* applies to vertical lists (vlists). The main duty of *hpack* and *vpack* is to compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified. The computed sizes normally enclose all of the material inside the new box; but some items may stick out if negative glue is used, if the box is overfull, or if a \vbox includes other boxes that have been shifted left.

The subroutine call *hpack*(*p, w, m*) returns a pointer to an *hlist_node* for a box containing the hlist that starts at *p*. Parameter *w* specifies a width; and parameter *m* is either '*exactly*' or '*additional*'. Thus, *hpack*(*p, w, exactly*) produces a box whose width is exactly *w*, while *hpack*(*p, w, additional*) yields a box whose width is the natural width plus *w*. It is convenient to define a macro called '*natural*' to cover the most common case, so that we can say *hpack*(*p, natural*) to get a box that has the natural width of list *p*.

Similarly, *vpack*(*p, w, m*) returns a pointer to a *vlist_node* for a box containing the vlist that starts at *p*. In this case *w* represents a height instead of a width; the parameter *m* is interpreted as in *hpack*.

```
define exactly = 0 { a box dimension is pre-specified }
define additional = 1 { a box dimension is increased from the natural one }
define natural ≡ 0, additional { shorthand for parameters to hpack and vpack }
```

817. The parameters to *hpack* and *vpack* correspond to TeX's primitives like '\hbox to 300pt', '\hbox spread 10pt'; note that '\hbox' with no dimension following it is equivalent to '\hbox spread 0pt'. The *scan_spec* subroutine scans such constructions in the user's input, including the mandatory left brace that follows them, and it puts the specification onto *save_stack* so that the desired box can later be obtained by executing the following code:

```
save_ptr ← save_ptr - 2;
hpack(p, saved(1), saved(0)).
```

Special care is necessary to ensure that the special *save_stack* codes are placed just below the new group code, because scanning can change *save_stack* when \csname appears.

```
procedure scan_spec(c : group_code; three_codes : boolean); { scans a box specification and left brace }
label found;
var s: integer; { temporarily saved value }
spec_code: exactly .. additional;
begin if three_codes then s ← saved(0);
if scan_keyword("to") then spec_code ← exactly
else if scan_keyword("spread") then spec_code ← additional
else begin spec_code ← additional; cur_val ← 0; goto found;
end;
scan_normal_dimen;
found: if three_codes then
begin saved(0) ← s; incr(save_ptr);
end;
saved(0) ← spec_code; saved(1) ← cur_val; save_ptr ← save_ptr + 2; new_save_level(c); scan_left_brace;
end;
```

818. To figure out the glue setting, *hpack* and *vpack* determine how much stretchability and shrinkability are present, considering all four orders of infinity. The highest order of infinity that has a nonzero coefficient is then used as if no other orders were present.

For example, suppose that the given list contains six glue nodes with the respective stretchabilities 3pt, 8fill, 5fil, 6pt, -3fil, -8fill. Then the total is essentially 2fil; and if a total additional space of 6pt is to be achieved by stretching, the actual amounts of stretch will be 0pt, 0pt, 15pt, 0pt, -9pt, and 0pt, since only 'fil' glue will be considered. (The 'fill' glue is therefore not really stretching infinitely with respect to 'fil'; nobody would actually want that to happen.)

The arrays *total_stretch* and *total_shrink* are used to determine how much glue of each kind is present. A global variable *last_badness* is used to implement \badness.

```
<Global variables 13> +≡
total_stretch, total_shrink: array [glue_ord] of scaled; { glue found by hpack or vpack }
last_badness: integer; { badness of the most recently packaged box }
```

819. If the global variable *adjust_tail* is non-null, the *hpack* routine also removes all occurrences of *ins_node*, *mark_node*, and *adjust_node* items and appends the resulting material onto the list that ends at location *adjust_tail*.

```
<Global variables 13> +≡
adjust_tail: pointer; { tail of adjustment list }
```

820. < Set initial values of key variables 21 > +≡
 $adjust_tail \leftarrow null; last_badness \leftarrow 0;$

821. < Global variables 13 > +≡

```
pdf_font_blink: ↑internal_font_number; { link to base font (used for expanded fonts only) }
pdf_font_elink: ↑internal_font_number; { link to expanded fonts (used for base fonts only) }
pdf_font_has_space_char: ↑boolean; { has font a real space char? }
pdf_font_stretch: ↑integer; { link to font expanded by stretch limi }
pdf_font_shrink: ↑integer; { link to font expanded by shrink limit }
pdf_font_step: ↑integer; { amount of one step of expansion }
pdf_font_expand_ratio: ↑integer; { expansion ratio of a particular font }
pdf_font_auto_expand: ↑boolean; { this font is auto-expanded? }
pdf_font_lp_base: ↑integer; { base of left-protruding factor }
pdf_font_rp_base: ↑integer; { base of right-protruding factor }
pdf_font_ef_base: ↑integer; { base of font expansion factor }
pdf_font_kn_bs_base: ↑integer; { base of kern before space }
pdf_font_st_bs_base: ↑integer; { base of stretch before space }
pdf_font_sh_bs_base: ↑integer; { base of shrink before space }
pdf_font_kn_bc_base: ↑integer; { base of kern before character }
pdf_font_kn_ac_base: ↑integer; { base of kern after character }
font_expand_ratio: integer; { current expansion ratio }
last_leftmost_char: pointer;
last_rightmost_char: pointer;
hlist_stack: array [0 .. max_hlist_stack] of pointer;
{ stack for find_protchar_left() and find_protchar_right() }
hlist_stack_level: 0 .. max_hlist_stack; { fill level for hlist_stack }
```

```

822. define cal_margin_kern_var(#) ≡
  begin character(cp) ← character(#); font(cp) ← font(#); do_subst_font(cp, 1000);
  if font(cp) ≠ font(#) then
    margin_kern_stretch ← margin_kern_stretch + left_pw(#) – left_pw(cp);
    font(cp) ← font(#); do_subst_font(cp, -1000);
  if font(cp) ≠ font(#) then
    margin_kern_shrink ← margin_kern_shrink + left_pw(cp) – left_pw(#);
  end

⟨ Calculate variations of marginal kerns 822 ⟩ ≡
  begin lp ← last_leftmost_char; rp ← last_rightmost_char; fast_get_avail(cp);
  if lp ≠ null then cal_margin_kern_var(lp);
  if rp ≠ null then cal_margin_kern_var(rp);
  free_avail(cp);
  end

```

This code is used in section 1027.

823. Here is *hpack*, which is place where we do font substituting when font expansion is being used. We define some constants used when calling *hpack* to deal with font expansion.

```

define cal_expand_ratio ≡ 2 { calculate amount for font expansion after breaking paragraph into lines }
define subst_ex_font ≡ 3 { substitute fonts }
define substituted = 3 { subtype of kern nodes that should be substituted }
define left_pw(#) ≡ char_pw(#, left_side)
define right_pw(#) ≡ char_pw(#, right_side)

function check_expand_pars(f : internal_font_number): boolean;
  var k: internal_font_number;
  begin check_expand_pars ← false;
  if (pdf_font_step[f] = 0) ∨ ((pdf_font_stretch[f] = null_font) ∧ (pdf_font_shrink[f] = null_font)) then
    return;
  if cur_font_step < 0 then cur_font_step ← pdf_font_step[f]
  else if cur_font_step ≠ pdf_font_step[f] then pdf_error("font_expansion",
    "using_fonts_with_different_step_of_expansion_in_one_paragraph_is_not_allowed");
  k ← pdf_font_stretch[f];
  if k ≠ null_font then
    begin if max_stretch_ratio < 0 then max_stretch_ratio ← pdf_font_expand_ratio[k]
    else if max_stretch_ratio ≠ pdf_font_expand_ratio[k] then pdf_error("font_expansion",
      "using_fonts_with_different_limit_of_expansion_in_one_paragraph_is_not_allowed");
    end;
  k ← pdf_font_shrink[f];
  if k ≠ null_font then
    begin if max_shrink_ratio < 0 then max_shrink_ratio ← -pdf_font_expand_ratio[k]
    else if max_shrink_ratio ≠ -pdf_font_expand_ratio[k] then pdf_error("font_expansion",
      "using_fonts_with_different_limit_of_expansion_in_one_paragraph_is_not_allowed");
    end;
  check_expand_pars ← true;
  end;

function char_stretch(f : internal_font_number; c : eight_bits): scaled;
  var k: internal_font_number; dw: scaled; ef: integer;
  begin char_stretch ← 0; k ← pdf_font_stretch[f]; ef ← get_ef_code(f, c);
  if (k ≠ null_font) ∧ (ef > 0) then
    begin dw ← char_width(k)(char_info(k)(c)) – char_width(f)(char_info(f)(c));
    if dw > 0 then char_stretch ← round_xn_over_d(dw, ef, 1000);
    end;
  end;

function char_shrink(f : internal_font_number; c : eight_bits): scaled;
  var k: internal_font_number; dw: scaled; ef: integer;
  begin char_shrink ← 0; k ← pdf_font_shrink[f]; ef ← get_ef_code(f, c);
  if (k ≠ null_font) ∧ (ef > 0) then
    begin dw ← char_width(f)(char_info(f)(c)) – char_width(k)(char_info(k)(c));
    if dw > 0 then char_shrink ← round_xn_over_d(dw, ef, 1000);
    end;
  end;

function get_kern(f : internal_font_number; lc, rc : eight_bits): scaled;
  label continue;
  var i: four_quarters; j: four_quarters; k: font_index;
  begin get_kern ← 0; i ← char_info(f)(lc);
  if char_tag(i) ≠ lig_tag then return;
  k ← lig_kern_start(f)(i); j ← font_info[k].qqqq;
  if skip_byte(j) ≤ stop_flag then goto continue + 1;

```

```

 $k \leftarrow \text{lig\_kern\_restart}(f)(j);$ 
 $\text{continue: } j \leftarrow \text{font\_info}[k].qqqq;$ 
 $\text{continue + 1: if } (\text{next\_char}(j) = rc) \wedge (\text{skip\_byte}(j) \leq \text{stop\_flag}) \wedge (\text{op\_byte}(j) \geq \text{kern\_flag}) \text{ then}$ 
 $\quad \text{begin } \text{get\_kern} \leftarrow \text{char\_kern}(f)(j); \text{return;}$ 
 $\quad \text{end;}$ 
 $\quad \text{if } \text{skip\_byte}(j) = q(0) \text{ then } \text{incr}(k)$ 
 $\quad \text{else begin if } \text{skip\_byte}(j) \geq \text{stop\_flag} \text{ then return;}$ 
 $\quad \quad k \leftarrow k + qo(\text{skip\_byte}(j)) + 1;$ 
 $\quad \quad \text{end;}$ 
 $\quad \text{goto continue;}$ 
 $\quad \text{end;}$ 
function  $\text{kern\_stretch}(p : \text{pointer}): \text{scaled};$ 
 $\quad \text{var } l, r : \text{pointer}; d : \text{scaled};$ 
 $\quad \text{begin } \text{kern\_stretch} \leftarrow 0;$ 
 $\quad \text{if } (\text{prev\_char\_p} = \text{null}) \vee (\text{link}(\text{prev\_char\_p}) \neq p) \vee (\text{link}(p) = \text{null}) \text{ then return;}$ 
 $\quad l \leftarrow \text{prev\_char\_p}; r \leftarrow \text{link}(p);$ 
 $\quad \text{if } \neg \text{is\_char\_node}(l) \text{ then}$ 
 $\quad \quad \text{if } \text{type}(l) = \text{ligature\_node} \text{ then } l \leftarrow \text{lig\_char}(l)$ 
 $\quad \quad \text{else return;}$ 
 $\quad \text{if } \neg \text{is\_char\_node}(r) \text{ then}$ 
 $\quad \quad \text{if } \text{type}(r) = \text{ligature\_node} \text{ then } r \leftarrow \text{lig\_char}(r)$ 
 $\quad \quad \text{else return;}$ 
 $\quad \text{if } \neg((\text{font}(l) = \text{font}(r)) \wedge (\text{pdf\_font\_stretch}[\text{font}(l)] \neq \text{null\_font})) \text{ then return;}$ 
 $\quad d \leftarrow \text{get\_kern}(\text{pdf\_font\_stretch}[\text{font}(l)], \text{character}(l), \text{character}(r));$ 
 $\quad \text{kern\_stretch} \leftarrow \text{round\_xn\_over\_d}(d - \text{width}(p), \text{get\_ef\_code}(\text{font}(l), \text{character}(l)), 1000);$ 
 $\quad \text{end;}$ 
function  $\text{kern\_shrink}(p : \text{pointer}): \text{scaled};$ 
 $\quad \text{var } l, r : \text{pointer}; d : \text{scaled};$ 
 $\quad \text{begin } \text{kern\_shrink} \leftarrow 0;$ 
 $\quad \text{if } (\text{prev\_char\_p} = \text{null}) \vee (\text{link}(\text{prev\_char\_p}) \neq p) \vee (\text{link}(p) = \text{null}) \text{ then return;}$ 
 $\quad l \leftarrow \text{prev\_char\_p}; r \leftarrow \text{link}(p);$ 
 $\quad \text{if } \neg \text{is\_char\_node}(l) \text{ then}$ 
 $\quad \quad \text{if } \text{type}(l) = \text{ligature\_node} \text{ then } l \leftarrow \text{lig\_char}(l)$ 
 $\quad \quad \text{else return;}$ 
 $\quad \text{if } \neg \text{is\_char\_node}(r) \text{ then}$ 
 $\quad \quad \text{if } \text{type}(r) = \text{ligature\_node} \text{ then } r \leftarrow \text{lig\_char}(r)$ 
 $\quad \quad \text{else return;}$ 
 $\quad \text{if } \neg((\text{font}(l) = \text{font}(r)) \wedge (\text{pdf\_font\_shrink}[\text{font}(l)] \neq \text{null\_font})) \text{ then return;}$ 
 $\quad d \leftarrow \text{get\_kern}(\text{pdf\_font\_shrink}[\text{font}(l)], \text{character}(l), \text{character}(r));$ 
 $\quad \text{kern\_shrink} \leftarrow \text{round\_xn\_over\_d}(\text{width}(p) - d, \text{get\_ef\_code}(\text{font}(l), \text{character}(l)), 1000);$ 
 $\quad \text{end;}$ 
procedure  $\text{do\_subst\_font}(p : \text{pointer}; ex\_ratio : \text{integer});$ 
 $\quad \text{var } f, k : \text{internal\_font\_number}; r : \text{pointer}; ef : \text{integer};$ 
 $\quad \text{begin if } \neg \text{is\_char\_node}(p) \wedge (\text{type}(p) = \text{disc\_node}) \text{ then}$ 
 $\quad \quad \text{begin } r \leftarrow \text{pre\_break}(p);$ 
 $\quad \quad \text{while } r \neq \text{null} \text{ do}$ 
 $\quad \quad \quad \text{begin if } \text{is\_char\_node}(r) \vee (\text{type}(r) = \text{ligature\_node}) \text{ then } \text{do\_subst\_font}(r, ex\_ratio);$ 
 $\quad \quad \quad r \leftarrow \text{link}(r);$ 
 $\quad \quad \quad \text{end;}$ 
 $\quad \quad r \leftarrow \text{post\_break}(p);$ 
 $\quad \quad \text{while } r \neq \text{null} \text{ do}$ 
 $\quad \quad \quad \text{begin if } \text{is\_char\_node}(r) \vee (\text{type}(r) = \text{ligature\_node}) \text{ then } \text{do\_subst\_font}(r, ex\_ratio);$ 

```

```

    r ← link(r);
    end;
  return;
end;

if is_char_node(p) then r ← p
else if type(p) = ligature_node then r ← lig_char(p)
else begin { short_display_n(p, 5); }
  pdf_error("font_expansion", "invalid_node_type");
end;

f ← font(r); ef ← get_ef_code(f, character(r));
if ef = 0 then return;
if (pdf_font_stretch[f] ≠ null_font) ∧ (ex_ratio > 0) then
  k ← expand_font(f, ext_xn_over_d(ex_ratio * ef, pdf_font_expand_ratio[pdf_font_stretch[f]], 1000000))
else if (pdf_font_shrink[f] ≠ null_font) ∧ (ex_ratio < 0) then
  k ← expand_font(f, ext_xn_over_d(ex_ratio * ef, -pdf_font_shrink[pdf_font_shrink[f]], 1000000))
else k ← f;
if k ≠ f then
  begin font(r) ← k;
  if ¬is_char_node(p) then
    begin r ← lig_ptr(p);
    while r ≠ null do
      begin font(r) ← k; r ← link(r);
      end;
    end;
  end;
end;

function char_pw(p : pointer; side : small_number): scaled;
var f: internal_font_number; c: integer;
begin char_pw ← 0;
if side = left_side then last_leftmost_char ← null
else last_rightmost_char ← null;
if p = null then return;
if ¬is_char_node(p) then
  begin if type(p) = ligature_node then p ← lig_char(p)
  else return;
  end;
f ← font(p);
if side = left_side then
  begin c ← get_lp_code(f, character(p)); last_leftmost_char ← p;
  end
else begin c ← get_rp_code(f, character(p)); last_rightmost_char ← p;
  end;
if c = 0 then return;
char_pw ← round_xn_over_d(quad(f), c, 1000);
end;

function new_margin_kern(w : scaled; p : pointer; side : small_number): pointer;
var k: pointer;
begin k ← get_node(margin_kern_node_size); type(k) ← margin_kern_node; subtype(k) ← side;
width(k) ← w;
if p = null then pdf_error("margin_kerning", "invalid_pointer_to_marginal_char_node");
fast_get_avail(margin_char(k)); character(margin_char(k)) ← character(p);
font(margin_char(k)) ← font(p); new_margin_kern ← k;

```

```

end;
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 }
  font_stretch: scaled; font_shrink: scaled; k: scaled;
begin last_badness  $\leftarrow$  0; r  $\leftarrow$  get_node(box_node_size); type(r)  $\leftarrow$  hlist_node;
  subtype(r)  $\leftarrow$  min_quarterword; shift_amount(r)  $\leftarrow$  0; q  $\leftarrow$  r + list_offset; link(q)  $\leftarrow$  p;
  if m = cal_expand_ratio then
    begin prev_char_p  $\leftarrow$  null; font_stretch  $\leftarrow$  0; font_shrink  $\leftarrow$  0; font_expand_ratio  $\leftarrow$  0;
    end;
  h  $\leftarrow$  0; { Clear dimensions to zero 824 };
  if TeXXeT_en then { Initialize the LR stack 1710 };
  while p  $\neq$  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 825 };
  if adjust_tail  $\neq$  null then link(adjust_tail)  $\leftarrow$  null;
  if pre_adjust_tail  $\neq$  null then link(pre_adjust_tail)  $\leftarrow$  null;
  height(r)  $\leftarrow$  h; depth(r)  $\leftarrow$  d;
  { Determine the value of width(r) and the appropriate glue setting; then return or goto
    common-ending 833 };
  common-ending: { Finish issuing a diagnostic message for an overfull or underfull hbox 839 };
  exit: if TeXXeT_en then { Check for LR anomalies at the end of hpack 1712 };
  if (m = cal_expand_ratio)  $\wedge$  (font_expand_ratio  $\neq$  0) then
    begin font_expand_ratio  $\leftarrow$  fix_int(font_expand_ratio, -1000, 1000); q  $\leftarrow$  list_ptr(r);
    free_node(r, box_node_size); r  $\leftarrow$  hpack(q, w, subst_ex_font);
    end;
  hpack  $\leftarrow$  r;
end;

```

824. { Clear dimensions to zero 824 } \equiv

d \leftarrow 0; *x* \leftarrow 0; total_stretch[normal] \leftarrow 0; total_shrink[normal] \leftarrow 0; total_stretch[fil] \leftarrow 0;
 total_shrink[fil] \leftarrow 0; total_stretch[fill] \leftarrow 0; total_shrink[fill] \leftarrow 0; total_stretch[filll] \leftarrow 0;
 total_shrink[filll] \leftarrow 0

This code is used in sections 823 and 844.

825. ⟨ 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 825 ⟩ ≡

```

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 828 ⟩;
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 827 ⟩;
    ins_node, mark_node, adjust_node: if (adjust_tail ≠ null) ∨ (pre_adjust_tail ≠ null) then
      ⟨ Transfer node  $p$  to the adjustment list 831 ⟩;
    whatsit_node: ⟨ Incorporate a whatsit node into an hbox 1607 ⟩;
    glue_node: ⟨ Incorporate glue into the horizontal totals 832 ⟩;
    margin_kern_node: begin if  $m = \text{cal\_expand\_ratio}$  then
      begin  $f \leftarrow \text{font}(\text{margin\_char}(p))$ ;  $\text{do\_subst\_font}(\text{margin\_char}(p), 1000)$ ;
      if  $f \neq \text{font}(\text{margin\_char}(p))$  then
         $\text{font\_stretch} \leftarrow \text{font\_stretch} - \text{width}(p) - \text{char\_pw}(\text{margin\_char}(p), \text{subtype}(p))$ ;
         $\text{font}(\text{margin\_char}(p)) \leftarrow f$ ;  $\text{do\_subst\_font}(\text{margin\_char}(p), -1000)$ ;
      if  $f \neq \text{font}(\text{margin\_char}(p))$  then
         $\text{font\_shrink} \leftarrow \text{font\_shrink} - \text{width}(p) - \text{char\_pw}(\text{margin\_char}(p), \text{subtype}(p))$ ;
         $\text{font}(\text{margin\_char}(p)) \leftarrow f$ ;
      end
    else if  $m = \text{subst\_ex\_font}$  then
      begin  $\text{do\_subst\_font}(\text{margin\_char}(p), \text{font\_expand\_ratio})$ ;
       $\text{width}(p) \leftarrow -\text{char\_pw}(\text{margin\_char}(p), \text{subtype}(p))$ ;
      end;
     $x \leftarrow x + \text{width}(p)$ ;
  end;
  kern_node: begin if subtype(p) = normal then
    begin if  $m = \text{cal\_expand\_ratio}$  then
      begin  $\text{font\_stretch} \leftarrow \text{font\_stretch} + \text{kern\_stretch}(p)$ ;  $\text{font\_shrink} \leftarrow \text{font\_shrink} + \text{kern\_shrink}(p)$ ;
      end
    else if  $m = \text{subst\_ex\_font}$  then
      begin if  $\text{font\_expand\_ratio} > 0$  then  $k \leftarrow \text{kern\_stretch}(p)$ 
      else if  $\text{font\_expand\_ratio} < 0$  then  $k \leftarrow \text{kern\_shrink}(p)$ 
      else  $\text{pdfassert}(0)$ ;
      if  $k \neq 0$  then
        begin if is_char_node(link(p)) then
           $\text{width}(p) \leftarrow \text{get\_kern}(\text{font}(\text{prev\_char\_p}), \text{character}(\text{prev\_char\_p}), \text{character}(\text{link}(p)))$ 
        else if type(link(p)) = ligature_node then  $\text{width}(p) \leftarrow \text{get\_kern}(\text{font}(\text{prev\_char\_p}),$ 
           $\text{character}(\text{prev\_char\_p}), \text{character}(\text{lig\_char}(\text{link}(p))))$ ;
        end;
      end;
    end;
     $x \leftarrow x + \text{width}(p)$ ;
  end;
  math_node: begin  $x \leftarrow x + \text{width}(p)$ ;
  if TeXXeT_en then ⟨ Adjust the LR stack for the hpack routine 1711 ⟩;
  end;
  ligature_node: begin if  $m = \text{subst\_ex\_font}$  then  $\text{do\_subst\_font}(p, \text{font\_expand\_ratio})$ ;
  ⟨ Make node  $p$  look like a char_node and goto reswitch 826 ⟩;
  end;
  disc_node: if  $m = \text{subst\_ex\_font}$  then  $\text{do\_subst\_font}(p, \text{font\_expand\_ratio})$ ;

```

```

othercases do_nothing
endcases;
p ← link(p);
end;
end

```

This code is used in section 823.

826. ⟨ Make node p look like a *char_node* and **goto** *reswitch* 826 ⟩ ≡
begin *mem*[*lig_trick*] ← *mem*[*lig_char*(p)]; *link*(*lig_trick*) ← *link*(p); $p \leftarrow \text{lig_trick}$; **goto** *reswitch*;
end

This code is used in sections 650, 732, 825, and 1325.

827. The code here implicitly uses the fact that running dimensions are indicated by *null_flag*, which will be ignored in the calculations because it is a highly negative number.

⟨ Incorporate box dimensions into the dimensions of the hbox that will contain it 827 ⟩ ≡
begin $x \leftarrow x + \text{width}(p)$;
if *type*(p) ≥ *rule_node* **then** $s \leftarrow 0$ **else** $s \leftarrow \text{shift_amount}(p)$;
if *height*(p) − $s > h$ **then** $h \leftarrow \text{height}(p) - s$;
if *depth*(p) + $s > d$ **then** $d \leftarrow \text{depth}(p) + s$;
end

This code is used in section 825.

828. The following code is part of TeX's inner loop; i.e., adding another character of text to the user's input will cause each of these instructions to be exercised one more time.

⟨ Incorporate character dimensions into the dimensions of the hbox that will contain it, then move to the next node 828 ⟩ ≡
begin if $m \geq \text{cal_expand_ratio}$ **then**
begin *prev_char_p* ← p ;
case m **of**
cal_expand_ratio: **begin** $f \leftarrow \text{font}(p)$; *add_char_stretch*(*font_stretch*)(*character*(p));
add_char_shrink(*font_shrink*)(*character*(p));
end;
subst_ex_font: *do_subst_font*(p , *font_expand_ratio*);
endcases;
end;
 $f \leftarrow \text{font}(p)$; $i \leftarrow \text{char_info}(f)(\text{character}(p))$; $hd \leftarrow \text{height_depth}(i)$; $x \leftarrow x + \text{char_width}(f)(i)$;
 $s \leftarrow \text{char_height}(f)(hd)$; **if** $s > h$ **then** $h \leftarrow s$;
 $s \leftarrow \text{char_depth}(f)(hd)$; **if** $s > d$ **then** $d \leftarrow s$;
 $p \leftarrow \text{link}(p)$;
end

This code is used in section 825.

829. Although node q is not necessarily the immediate predecessor of node p , it always points to some node in the list preceding p . Thus, we can delete nodes by moving q when necessary. The algorithm takes linear time, and the extra computation does not intrude on the inner loop unless it is necessary to make a deletion.

⟨ Global variables 13 ⟩ +≡
pre_adjust_tail: pointer;

830. ⟨ Set initial values of key variables 21 ⟩ +≡
pre_adjust_tail ← *null*;

831. Materials in `\vadjust` used with `pre` keyword will be appended to `pre-adjust-tail` instead of `adjust-tail`. ▀

```
define update-adjust-list (#) ≡
  begin if # = null then confusion("pre\vadjust");
  link(#) ← adjust_ptr(p);
  while link(#) ≠ null do # ← link(#);
  end
⟨ Transfer node p to the adjustment list 831 ⟩ ≡
  begin while link(q) ≠ p do q ← link(q);
  if type(p) = adjust_node then
    begin if adjust-pre(p) ≠ 0 then update-adjust-list(pre-adjust-tail)
    else update-adjust-list(adjust-tail);
    p ← link(p); free-node(link(q), small_node_size);
    end
  else begin link(adjust-tail) ← p; adjust-tail ← p; p ← link(p);
  end;
  link(q) ← p; p ← q;
  end
```

This code is used in section 825.

832. ⟨ Incorporate glue into the horizontal totals 832 ⟩ ≡

```
begin g ← glue_ptr(p); x ← x + width(g);
o ← stretch_order(g); total_stretch[o] ← total_stretch[o] + stretch(g); o ← shrink_order(g);
total_shrink[o] ← total_shrink[o] + shrink(g);
if subtype(p) ≥ a_leaders then
  begin g ← leader_ptr(p);
  if height(g) > h then h ← height(g);
  if depth(g) > d then d ← depth(g);
  end;
end
```

This code is used in section 825.

833. When we get to the present part of the program, x is the natural width of the box being packaged.

⟨ Determine the value of `width(r)` and the appropriate glue setting; then `return` or `goto common-ending 833` ⟩ ≡

```
if m = additional then w ← x + w;
width(r) ← w; x ← w - x; { now x is the excess to be made up }
if x = 0 then
  begin glue-sign(r) ← normal; glue-order(r) ← normal; set_glue_ratio_zero(glue_set(r)); return;
  end
else if x > 0 then ⟨ Determine horizontal glue stretch setting, then return or goto common-ending 834 ⟩
  else ⟨ Determine horizontal glue shrink setting, then return or goto common-ending 840 ⟩
```

This code is used in section 823.

834. If *hpack* is called with $m = \text{cal_expand_ratio}$ we calculate *font_expand_ratio* and return without checking for overfull or underfull box.

```
< Determine horizontal glue stretch setting, then return or goto common-ending 834 > ≡
begin (Determine the stretch order 835);
if (m = cal_expand_ratio) ∧ (o = normal) ∧ (font_stretch > 0) then
begin font_expand_ratio ← divide_scaled(x, font_stretch, 3); return;
end;
glue_order(r) ← o; glue_sign(r) ← stretching;
if total_stretch[o] ≠ 0 then glue_set(r) ← unfloat(x / total_stretch[o])
else begin glue_sign(r) ← normal; set_glue_ratio_zero(glue_set(r)); { there's nothing to stretch }
end;
if o = normal then
if list_ptr(r) ≠ null then
    (Report an underfull hbox and goto common-ending, if this box is sufficiently bad 836);
return;
end
```

This code is used in section 833.

835. < Determine the stretch order 835 > ≡
if total_stretch[filll] ≠ 0 then o ← filll
else if total_stretch[fill] ≠ 0 then o ← fill
else if total_stretch[fil] ≠ 0 then o ← fil
else o ← normal

This code is used in sections 834, 849, and 972.

836. < Report an underfull hbox and goto common-ending, if this box is sufficiently bad 836 > ≡
begin last_badness ← badness(x, total_stretch[normal]);
if last_badness > hbadness then
begin print_ln;
if last_badness > 100 then print_nl("Underfull") else print_nl("Loose");
print(" \hbox{badness} "); print_int(last_badness); goto common-ending;
end;
end

This code is used in section 834.

837. In order to provide a decent indication of where an overfull or underfull box originated, we use a global variable *pack_begin_line* that is set nonzero only when *hpack* is being called by the paragraph builder or the alignment finishing routine.

```
< Global variables 13 > +≡
pack_begin_line: integer; { source file line where the current paragraph or alignment began; a negative
value denotes alignment }
```

838. < Set initial values of key variables 21 > +≡
pack_begin_line ← 0;

839. ⟨Finish issuing a diagnostic message for an overfull or underfull hbox 839⟩ ≡

```

if output_active then print(")\u202ahas\u202aoccurred\u202awhile\u202a\output\u202ais\u202aactive")
else begin if pack_begin_line ≠ 0 then
    begin if pack_begin_line > 0 then print(")\u202ain\u202aparagraph\u202aat\u202alines\u202a")
    else print(")\u202ain\u202alignment\u202aat\u202alines\u202a");
    print_int(abs(pack_begin_line)); print("--");
    end
else print(")\u202adetected\u202at\u202aline\u202a");
print_int(line);
end;
print_ln;
font_in_short_display ← null_font; short_display(list_ptr(r)); print_ln;
begin_diagnostic; show_box(r); end_diagnostic(true)

```

This code is used in section 823.

840. ⟨Determine horizontal glue shrink setting, then return or goto common-ending 840⟩ ≡

```

begin ⟨Determine the shrink order 841⟩;
if (m = cal_expand_ratio) ∧ (o = normal) ∧ (font_shrink > 0) then
    begin font_expand_ratio ← divide_scaled(x, font_shrink, 3); return;
    end;
glue_order(r) ← o; glue_sign(r) ← shrinking;
if total_shrink[o] ≠ 0 then glue_set(r) ← unfloat((-x)/total_shrink[o])
else begin glue_sign(r) ← normal; set_glue_ratio_zero(glue_set(r)); { there's nothing to shrink }
    end;
if (total_shrink[o] < -x) ∧ (o = normal) ∧ (list_ptr(r) ≠ null) then
    begin last_badness ← 1000000; set_glue_ratio_one(glue_set(r)); { use the maximum shrinkage }
    ⟨Report an overfull hbox and goto common-ending, if this box is sufficiently bad 842⟩;
    end
else if o = normal then
    if list_ptr(r) ≠ null then
        ⟨Report a tight hbox and goto common-ending, if this box is sufficiently bad 843⟩;
return;
end

```

This code is used in section 833.

841. ⟨Determine the shrink order 841⟩ ≡

```

if total_shrink[filll] ≠ 0 then o ← filll
else if total_shrink[fill] ≠ 0 then o ← fill
else if total_shrink[fil] ≠ 0 then o ← fil
else o ← normal

```

This code is used in sections 840, 852, and 972.

842. ⟨Report an overfull hbox and goto common-ending, if this box is sufficiently bad 842⟩ ≡

```

if (-x - total_shrink[normal] > hfuzz) ∨ (hbadness < 100) then
    begin if (overfull_rule > 0) ∧ (-x - total_shrink[normal] > hfuzz) then
        begin while link(q) ≠ null do q ← link(q);
        link(q) ← new_rule; width(link(q)) ← overfull_rule;
        end;
        print_ln; print_nl("Overfull\u202ahbox\u202a("); print_scaled(-x - total_shrink[normal]);
        print("pt\u202ato\u202awide"); goto common_heading;
    end

```

This code is used in section 840.

843. ⟨ Report a tight hbox and **goto** *common-ending*, if this box is sufficiently bad 843 ⟩ ≡

```

begin last_badness ← badness(−x, total_shrink[normal]);
if last_badness > hbadness then
  begin print_ln; print_nl("Tight\ hbox\ (badness\ ");
  print_int(last_badness); goto common-ending;
end;
end

```

This code is used in section 840.

844. The *vpack* subroutine is actually a special case of a slightly more general routine called *vpackage*, which has four parameters. The fourth parameter, which is *max_dimen* in the case of *vpack*, specifies the maximum depth of the page box that is constructed. The depth is first computed by the normal rules; if it exceeds this limit, the reference point is simply moved down until the limiting depth is attained.

```

define vpack(#) ≡ vpackage(#, max_dimen) { special case of unconstrained depth }
function vpackage(p : pointer; h : scaled; m : small_number; l : scaled): pointer;
label common-ending, exit;
var r: pointer; { the box node that will be returned }
w, d, x: scaled; { width, depth, and natural height }
s: scaled; { shift amount }
g: pointer; { points to a glue specification }
o: glue_ord; { order of infinity }
begin last_badness ← 0; r ← get_node(box_node_size); type(r) ← vlist_node;
subtype(r) ← min_quarterword; shift_amount(r) ← 0; list_ptr(r) ← p;
w ← 0; {Clear dimensions to zero 824};
while p ≠ null do {Examine node p in the vlist, taking account of its effect on the dimensions of the
new box; then advance p to the next node 845};
width(r) ← w;
if d > l then
  begin x ← x + d - l; depth(r) ← l;
  end
else depth(r) ← d;
{Determine the value of height(r) and the appropriate glue setting; then return or goto
common-ending 848};
common-ending: {Finish issuing a diagnostic message for an overfull or underfull vbox 851};
exit: vpackage ← r;
end;

```

845. {Examine node *p* in the vlist, taking account of its effect on the dimensions of the new box; then advance *p* to the next node 845} ≡

```

begin if is_char_node(p) then confusion("vpack")
else case type(p) of
  hlist_node, vlist_node, rule_node, unset_node: {Incorporate box dimensions into the dimensions of the
vbox that will contain it 846};
  whatsit_node: {Incorporate a whatsit node into a vbox 1606};
  glue_node: {Incorporate glue into the vertical totals 847};
  kern_node: begin x ← x + d + width(p); d ← 0;
  end;
  othercases do_nothing
endcases;
p ← link(p);
end

```

This code is used in section 844.

846. \langle Incorporate box dimensions into the dimensions of the vbox that will contain it 846 $\rangle \equiv$

```

begin  $x \leftarrow x + d + height(p); d \leftarrow depth(p);$ 
if  $type(p) \geq rule\_node$  then  $s \leftarrow 0$  else  $s \leftarrow shift\_amount(p);$ 
if  $width(p) + s > w$  then  $w \leftarrow width(p) + s;$ 
end

```

This code is used in section 845.

847. \langle Incorporate glue into the vertical totals 847 $\rangle \equiv$

```

begin  $x \leftarrow x + d; d \leftarrow 0;$ 
 $g \leftarrow glue\_ptr(p); x \leftarrow x + width(g);$ 
 $o \leftarrow stretch\_order(g); total\_stretch[o] \leftarrow total\_stretch[o] + stretch(g); o \leftarrow shrink\_order(g);$ 
 $total\_shrink[o] \leftarrow total\_shrink[o] + shrink(g);$ 
if  $subtype(p) \geq a\_leaders$  then
begin  $g \leftarrow leader\_ptr(p);$ 
if  $width(g) > w$  then  $w \leftarrow width(g);$ 
end;
end

```

This code is used in section 845.

848. When we get to the present part of the program, x is the natural height of the box being packaged.

\langle Determine the value of $height(r)$ and the appropriate glue setting; then return or goto common_ending 848 $\rangle \equiv$

```

if  $m = additional$  then  $h \leftarrow x + h;$ 
 $height(r) \leftarrow h; x \leftarrow h - x;$  { now  $x$  is the excess to be made up }
if  $x = 0$  then
begin  $glue\_sign(r) \leftarrow normal; glue\_order(r) \leftarrow normal; set\_glue\_ratio\_zero(glue\_set(r)); return;$ 
end
else if  $x > 0$  then  $\langle$  Determine vertical glue stretch setting, then return or goto common_ending 849  $\rangle$ 
else  $\langle$  Determine vertical glue shrink setting, then return or goto common_ending 852  $\rangle$ 

```

This code is used in section 844.

849. \langle Determine vertical glue stretch setting, then return or goto common_ending 849 $\rangle \equiv$

```

begin  $\langle$  Determine the stretch order 835  $\rangle;$ 
 $glue\_order(r) \leftarrow o; glue\_sign(r) \leftarrow stretching;$ 
if  $total\_stretch[o] \neq 0$  then  $glue\_set(r) \leftarrow unfloat(x/total\_stretch[o])$ 
else begin  $glue\_sign(r) \leftarrow normal; set\_glue\_ratio\_zero(glue\_set(r));$  { there's nothing to stretch }
end;
if  $o = normal$  then
if  $list\_ptr(r) \neq null$  then
 $\langle$  Report an underfull vbox and goto common_ending, if this box is sufficiently bad 850  $\rangle;$ 
return;
end

```

This code is used in section 848.

850. ⟨ Report an underfull vbox and goto *common-ending*, if this box is sufficiently bad 850 ⟩ ≡

```

begin last_badness ← badness(x, total_stretch[normal]);
if last_badness > vbadness then
  begin print_ln;
  if last_badness > 100 then print_nl("Underfull") else print_nl("Loose");
  print(" \vbox\bgroup(badness\egroup"); print_int(last_badness); goto commonEnding;
  end;
end

```

This code is used in section 849.

851. ⟨ Finish issuing a diagnostic message for an overfull or underfull vbox 851 ⟩ ≡

```

if output_active then print(") has occurred while \output is active")
else begin if pack_begin_line ≠ 0 then { it's actually negative }
  begin print(") in alignment at lines "); print_int(abs(pack_begin_line)); print("--");
  end
else print(") detected at line ");
print_int(line); print_ln;
end;
begin_diagnostic; show_box(r); end_diagnostic(true)

```

This code is used in section 844.

852. ⟨ Determine vertical glue shrink setting, then return or goto *common-ending* 852 ⟩ ≡

```

begin ⟨ Determine the shrink order 841 ⟩;
glue_order(r) ← o; glue_sign(r) ← shrinking;
if total_shrink[o] ≠ 0 then glue_set(r) ← unfloat((-x)/total_shrink[o])
else begin glue_sign(r) ← normal; set_glue_ratio_zero(glue_set(r)); { there's nothing to shrink }
  end;
if (total_shrink[o] < -x) ∧ (o = normal) ∧ (list_ptr(r) ≠ null) then
  begin last_badness ← 1000000; set_glue_ratio_one(glue_set(r)); { use the maximum shrinkage }
  ⟨ Report an overfull vbox and goto common-ending, if this box is sufficiently bad 853 ⟩;
  end
else if o = normal then
  if list_ptr(r) ≠ null then
    ⟨ Report a tight vbox and goto common-ending, if this box is sufficiently bad 854 ⟩;
  return;
end

```

This code is used in section 848.

853. ⟨ Report an overfull vbox and goto *common-ending*, if this box is sufficiently bad 853 ⟩ ≡

```

if (-x - total_shrink[normal] > vfuzz) ∨ (vbadness < 100) then
  begin print_ln; print_nl("Overfull \vbox("); print_scaled(-x - total_shrink[normal]);
  print("pt too high"); goto commonEnding;
end

```

This code is used in section 852.

854. ⟨ Report a tight vbox and goto *common-ending*, if this box is sufficiently bad 854 ⟩ ≡

```

begin last_badness ← badness(-x, total_shrink[normal]);
if last_badness > vbadness then
  begin print_ln; print_nl("Tight \vbox(badness "); print_int(last_badness); goto commonEnding;
  end;
end

```

This code is used in section 852.

855. When a box is being appended to the current vertical list, the baselineskip calculation is handled by the *append_to_vlist* routine.

```
procedure append_to_vlist(b : pointer);
  var d: scaled;  { deficiency of space between baselines }
  p: pointer;  { a new glue node }
begin if prev_depth > pdf_ignored_dimen then
  begin d ← width(baseline_skip) – prev_depth – height(b);
  if d < line_skip_limit then p ← new_param_glue(line_skip_code)
  else begin p ← new_skip_param(baseline_skip_code); width(temp_ptr) ← d; { temp_ptr = glue_ptr(p) }
    end;
  link(tail) ← p; tail ← p;
  end;
link(tail) ← b; tail ← b; prev_depth ← depth(b);
end;
```

856. Data structures for math mode. When TeX reads a formula that is enclosed between \$'s, it constructs an *mlist*, which is essentially a tree structure representing that formula. An *mlist* is a linear sequence of items, but we can regard it as a tree structure because *mlists* can appear within *mlists*. For example, many of the entries can be subscripted or superscripted, and such "scripts" are *mlists* in their own right.

An entire formula is parsed into such a tree before any of the actual typesetting is done, because the current style of type is usually not known until the formula has been fully scanned. For example, when the formula '\$a+b \over c+d\$' is being read, there is no way to tell that 'a+b' will be in script size until '\over' has appeared.

During the scanning process, each element of the *mlist* being built is classified as a relation, a binary operator, an open parenthesis, etc., or as a construct like '\sqrt' that must be built up. This classification appears in the *mlist* data structure.

After a formula has been fully scanned, the *mlist* is converted to an *hlist* so that it can be incorporated into the surrounding text. This conversion is controlled by a recursive procedure that decides all of the appropriate styles by a "top-down" process starting at the outermost level and working in towards the subformulas. The formula is ultimately pasted together using combinations of horizontal and vertical boxes, with glue and penalty nodes inserted as necessary.

An *mlist* is represented internally as a linked list consisting chiefly of "noads" (pronounced "no-adds"), to distinguish them from the somewhat similar "nodes" in *hlists* and *vlists*. Certain kinds of ordinary nodes are allowed to appear in *mlists* together with the noads; TeX tells the difference by means of the *type* field, since a noad's *type* is always greater than that of a node. An *mlist* does not contain character nodes, *hlist* nodes, *vlist* nodes, math nodes, ligature nodes, or unset nodes; in particular, each *mlist* item appears in the variable-size part of *mem*, so the *type* field is always present.

857. Each noad is four or more words long. The first word contains the *type* and *subtype* and *link* fields that are already so familiar to us; the second, third, and fourth words are called the noad's *nucleus*, *subscr*, and *supscr* fields.

Consider, for example, the simple formula '\$x^2\$', which would be parsed into an mlist containing a single element called an *ord_noad*. The *nucleus* of this noad is a representation of 'x', the *subscr* is empty, and the *supscr* is a representation of '2'.

The *nucleus*, *subscr*, and *supscr* fields are further broken into subfields. If *p* points to a noad, and if *q* is one of its principal fields (e.g., *q* = *subscr(p)*), there are several possibilities for the subfields, depending on the *math_type* of *q*.

math_type(q) = math_char means that *fam(q)* refers to one of the sixteen font families, and *character(q)* is the number of a character within a font of that family, as in a character node.

math_type(q) = math_text_char is similar, but the character is unsubscripted and unsuperscripted and it is followed immediately by another character from the same font. (This *math.type* setting appears only briefly during the processing; it is used to suppress unwanted italic corrections.)

math_type(q) = empty indicates a field with no value (the corresponding attribute of noad *p* is not present).

math_type(q) = sub_box means that *info(q)* points to a box node (either an *hlist_node* or a *vlist_node*) that should be used as the value of the field. The *shift_amount* in the subsidiary box node is the amount by which that box will be shifted downward.

math_type(q) = sub_mlist means that *info(q)* points to an mlist; the mlist must be converted to an hlist in order to obtain the value of this field.

In the latter case, we might have *info(q) = null*. This is not the same as *math_type(q) = empty*; for example, '\$P_{\{ \} }\$' and '\$P\$' produce different results (the former will not have the "italic correction" added to the width of *P*, but the "script skip" will be added).

The definitions of subfields given here are evidently wasteful of space, since a halfword is being used for the *math_type* although only three bits would be needed. However, there are hardly ever many noads present at once, since they are soon converted to nodes that take up even more space, so we can afford to represent them in whatever way simplifies the programming.

```
define noad_size = 4 { number of words in a normal noad }
define nucleus(#) ≡ # + 1 { the nucleus field of a noad }
define supscr(#) ≡ # + 2 { the supscr field of a noad }
define subscr(#) ≡ # + 3 { the subscr field of a noad }
define math_type ≡ link { a halfword in mem }
define fam ≡ font { a quarterword in mem }
define math_char = 1 { math_type when the attribute is simple }
define sub_box = 2 { math_type when the attribute is a box }
define sub_mlist = 3 { math_type when the attribute is a formula }
define math_text_char = 4 { math_type when italic correction is dubious }
```

858. Each portion of a formula is classified as Ord, Op, Bin, Rel, Open, Close, Punct, or Inner, for purposes of spacing and line breaking. An *ord_noad*, *op_noad*, *bin_noad*, *rel_noad*, *open_noad*, *close_noad*, *punct_noad*, or *inner_noad* is used to represent portions of the various types. For example, an '=' sign in a formula leads to the creation of a *rel_noad* whose *nucleus* field is a representation of an equals sign (usually *fam* = 0, *character* = '75). A formula preceded by \mathrel1 also results in a *rel_noad*. When a *rel_noad* is followed by an *op_noad*, say, and possibly separated by one or more ordinary nodes (not noads), TeX will insert a penalty node (with the current *rel_penalty*) just after the formula that corresponds to the *rel_noad*, unless there already was a penalty immediately following; and a "thick space" will be inserted just before the formula that corresponds to the *op_noad*.

A noad of type *ord_noad*, *op_noad*, ..., *inner_noad* usually has a *subtype* = *normal*. The only exception is that an *op_noad* might have *subtype* = *limits* or *no_limits*, if the normal positioning of limits has been overridden for this operator.

```
define ord_noad = unset_node + 3 { type of a noad classified Ord }
define op_noad = ord_noad + 1 { type of a noad classified Op }
define bin_noad = ord_noad + 2 { type of a noad classified Bin }
define rel_noad = ord_noad + 3 { type of a noad classified Rel }
define open_noad = ord_noad + 4 { type of a noad classified Open }
define close_noad = ord_noad + 5 { type of a noad classified Close }
define punct_noad = ord_noad + 6 { type of a noad classified Punct }
define inner_noad = ord_noad + 7 { type of a noad classified Inner }
define limits = 1 { subtype of op_noad whose scripts are to be above, below }
define no_limits = 2 { subtype of op_noad whose scripts are to be normal }
```

859. A *radical_noad* is five words long; the fifth word is the *left_delimiter* field, which usually represents a square root sign.

A *fraction_noad* is six words long; it has a *right_delimiter* field as well as a *left_delimiter*.

Delimiter fields are of type *four_quarters*, and they have four subfields called *small_fam*, *small_char*, *large_fam*, *large_char*. These subfields represent variable-size delimiters by giving the “small” and “large” starting characters, as explained in Chapter 17 of *The TeXbook*.

A *fraction_noad* is actually quite different from all other noads. Not only does it have six words, it has *thickness*, *denominator*, and *numerator* fields instead of *nucleus*, *subscr*, and *supscr*. The *thickness* is a scaled value that tells how thick to make a fraction rule; however, the special value *default_code* is used to stand for the *default_rule_thickness* of the current size. The *numerator* and *denominator* point to mlists that define a fraction; we always have

$$\text{math_type}(\text{numerator}) = \text{math_type}(\text{denominator}) = \text{sub_mlist}.$$

The *left_delimiter* and *right_delimiter* fields specify delimiters that will be placed at the left and right of the fraction. In this way, a *fraction_noad* is able to represent all of TeX’s operators *\over*, *\atop*, *\above*, *\overwithdelims*, *\atopwithdelims*, and *\abovewithdelims*.

```
define left_delimiter(#) ≡ # + 4 { first delimiter field of a noad }
define right_delimiter(#) ≡ # + 5 { second delimiter field of a fraction noad }
define radical_noad = inner_noad + 1 { type of a noad for square roots }
define radical_noad_size = 5 { number of mem words in a radical noad }
define fraction_noad = radical_noad + 1 { type of a noad for generalized fractions }
define fraction_noad_size = 6 { number of mem words in a fraction noad }
define small_fam(#) ≡ mem[#].qqqq.b0 { fam for “small” delimiter }
define small_char(#) ≡ mem[#].qqqq.b1 { character for “small” delimiter }
define large_fam(#) ≡ mem[#].qqqq.b2 { fam for “large” delimiter }
define large_char(#) ≡ mem[#].qqqq.b3 { character for “large” delimiter }
define thickness ≡ width { thickness field in a fraction noad }
define default_code ≡ '10000000000 { denotes default_rule_thickness }
define numerator ≡ supscr { numerator field in a fraction noad }
define denominator ≡ subscr { denominator field in a fraction noad }
```

860. The global variable *empty_field* is set up for initialization of empty fields in new noads. Similarly, *null_delimiter* is for the initialization of delimiter fields.

```
⟨ Global variables 13 ⟩ +≡
empty_field: two_halves;
null_delimiter: four_quarters;
```

861. ⟨ Set initial values of key variables 21 ⟩ +≡

```
empty_field.rh ← empty; empty_field.lh ← null;
null_delimiter.b0 ← 0; null_delimiter.b1 ← min_quarterword;
null_delimiter.b2 ← 0; null_delimiter.b3 ← min_quarterword;
```

862. The *new_noad* function creates an *ord_noad* that is completely null.

```
function new_noad: pointer;
  var p: pointer;
begin p ← get_node(noad_size); type(p) ← ord_noad; subtype(p) ← normal;
mem[nucleus(p)].hh ← empty_field; mem[subscr(p)].hh ← empty_field;
mem[supscr(p)].hh ← empty_field; new_noad ← p;
end;
```

863. 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)
```

864. Math formulas can also contain instructions like *\textstyle* that override TeX's normal style rules. A *style_node* is inserted into the data structure to record such instructions; it is three words long, so it is considered a node instead of a noad. The *subtype* is either *display_style* or *text_style* or *script_style* or *script_script_style*. The second and third words of a *style_node* are not used, but they are present because a *choice_node* is converted to a *style_node*.

TeX uses even numbers 0, 2, 4, 6 to encode the basic styles *display_style*, ..., *script_script_style*, and adds 1 to get the "cramped" versions of these styles. This gives a numerical order that is backwards from the convention of Appendix G in *The TeXbook*; i.e., a smaller style has a larger numerical value.

```
define style_node = unset_node + 1 { type of a style node }
define style_node_size = 3 { number of words in a style node }
define display_style = 0 { subtype for \displaystyle }
define text_style = 2 { subtype for \textstyle }
define script_style = 4 { subtype for \scriptstyle }
define script_script_style = 6 { subtype for \scriptscriptstyle }
define cramped = 1 { add this to an uncramped style if you want to cramp it }

function new_style(s : small_number): pointer; { create a style node }
  var p: pointer; { the new node }
  begin p ← get_node(style_node_size); type(p) ← style_node; subtype(p) ← s; width(p) ← 0;
    depth(p) ← 0; { the width and depth are not used }
    new_style ← p;
  end;
```

865. Finally, the `\mathchoice` primitive creates a *choice_node*, which has special subfields *display_mlist*, *text_mlist*, *script_mlist*, and *script_script_mlist* pointing to the mlists for each style.

```
define choice_node = unset_node + 2 { type of a choice node }
define display_mlist(#) ≡ info(# + 1) { mlist to be used in display style }
define text_mlist(#) ≡ link(# + 1) { mlist to be used in text style }
define script_mlist(#) ≡ info(# + 2) { mlist to be used in script style }
define script_script_mlist(#) ≡ link(# + 2) { mlist to be used in scriptscript style }

function new_choice: pointer; { create a choice node }
  var p: pointer; { the new node }
  begin p ← get_node(style_node_size); type(p) ← choice_node; subtype(p) ← 0;
    { the subtype is not used }
  display_mlist(p) ← null; text_mlist(p) ← null; script_mlist(p) ← null; script_script_mlist(p) ← null;
  new_choice ← p;
  end;
```

866. Let's consider now the previously unwritten part of *show_node_list* that displays the things that can only be present in mlists; this program illustrates how to access the data structures just defined.

In the context of the following program, *p* points to a node or noad that should be displayed, and the current string contains the “recursion history” that leads to this point. The recursion history consists of a dot for each outer level in which *p* is subsidiary to some node, or in which *p* is subsidiary to the *nucleus* field of some noad; the dot is replaced by ‘_’ or ‘^’ or ‘/’ or ‘\’ if *p* is descended from the *subscr* or *supscr* or *denominator* or *numerator* fields of noads. For example, the current string would be ‘.^._/’ if *p* points to the *ord_noad* for *x* in the (ridiculous) formula ‘\$sqrt{a^{mathinner{b_{c\over x+y}}}}\$’.

```
{ Cases of show_node.list that arise in mlists only 866 } ≡
style_node: print_style(subtype(p));
choice_node: ⟨ Display choice node p 871 ⟩;
ord_noad, op_noad, bin_noad, rel_noad, open_noad, close_noad, punct_noad,
inner_noad, radical_noad, over_noad, under_noad, vcenter_noad, accent_noad, left_noad, right_noad:
⟨ Display normal noad p 872 ⟩;
fraction_noad: ⟨ Display fraction noad p 873 ⟩;
```

This code is used in section 201.

867. Here are some simple routines used in the display of noads.

```
{ Declare procedures needed for displaying the elements of mlists 867 } ≡
procedure print_fam_and_char(p: pointer); { prints family and character }
  begin print_esc("fam"); print_int(fam(p)); print_char(" "); print_ASCII(qo(character(p)));
  end;

procedure print_delimiter(p: pointer); { prints a delimiter as 24-bit hex value }
  var a: integer; { accumulator }
  begin a ← small_fam(p) * 256 + qo(small_char(p));
  a ← a * "1000 + large_fam(p) * 256 + qo(large_char(p));
  if a < 0 then print_int(a) { this should never happen }
  else print_hex(a);
  end;
```

See also sections 868 and 870.

This code is used in section 197.

868. The next subroutine will descend to another level of recursion when a subsidiary mlist needs to be displayed. The parameter c indicates what character is to become part of the recursion history. An empty mlist is distinguished from a field with $\text{math_type}(p) = \text{empty}$, because these are not equivalent (as explained above).

```
< Declare procedures needed for displaying the elements of mlists 867 > +≡
procedure show_info; forward; { show_node_list(info(temp_ptr)) }
procedure print_subsidary_data(p: pointer; c: ASCII_code); { display a noad field }
begin if cur_length ≥ depth_threshold then
  begin if math_type(p) ≠ empty then print("□[]");
  end
else begin append_char(c); { include c in the recursion history }
  temp_ptr ← p; { prepare for show_info if recursion is needed }
  case math_type(p) of
    math_char: begin print_ln; print_current_string; print_fam_and_char(p);
    end;
  sub_box: show_info; { recursive call }
  sub_mlist: if info(p) = null then
    begin print_ln; print_current_string; print("{}");
    end
    else show_info; { recursive call }
  othercases do_nothing { empty }
  endcases;
  flush_char; { remove c from the recursion history }
  end;
end;
```

869. The inelegant introduction of `show_info` in the code above seems better than the alternative of using Pascal's strange `forward` declaration for a procedure with parameters. The Pascal convention about dropping parameters from a post-`forward` procedure is, frankly, so intolerable to the author of TeX that he would rather stoop to communication via a global temporary variable. (A similar stupidity occurred with respect to `hlist_out` and `vlist_out` above, and it will occur with respect to `mlist_to_hlist` below.)

```
procedure show_info; { the reader will kindly forgive this }
begin show_node_list(info(temp_ptr));
end;
```

870. < Declare procedures needed for displaying the elements of mlists 867 > +≡
procedure print_style(c: integer);
begin case c div 2 of

```
0: print_esc("displaystyle"); { display_style = 0 }
1: print_esc("textstyle"); { text_style = 2 }
2: print_esc("scriptstyle"); { script_style = 4 }
3: print_esc("scriptscriptstyle"); { script_script_style = 6 }
othercases print("Unknown_\u00f6style!")
endcases;
end;
```

871. $\langle \text{Display choice node } p \rangle \equiv$

```
begin print_esc("mathchoice"); append_char("D"); show_node_list(display_mlist(p)); flush_char;
append_char("T"); show_node_list(text_mlist(p)); flush_char; append_char("S");
show_node_list(script_mlist(p)); flush_char; append_char("s"); show_node_list(script_script_mlist(p));
flush_char;
end
```

This code is used in section 866.

872. $\langle \text{Display normal noad } p \rangle \equiv$

```
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_subsidary_data(nucleus(p), ". ");
end;
print_subsidary_data(supscr(p), "^"); print_subsidary_data(subscr(p), "_");
end
```

This code is used in section 866.

873. $\langle \text{Display fraction noad } p \rangle \equiv$

```

begin print_esc("fraction_\u2022thickness\u2022");
if thickness(p) = default_code then print("=\u2022default")
else print_scaled(thickness(p));
if (small_fam(left_delimiter(p)) ≠ 0) ∨ (small_char(left_delimiter(p)) ≠ min_quarterword) ∨
    (large_fam(left_delimiter(p)) ≠ 0) ∨ (large_char(left_delimiter(p)) ≠ min_quarterword) then
begin print(",\u2022left-delimiter\u2022"); print_delimiter(left_delimiter(p));
end;
if (small_fam(right_delimiter(p)) ≠ 0) ∨ (small_char(right_delimiter(p)) ≠ min_quarterword) ∨
    (large_fam(right_delimiter(p)) ≠ 0) ∨ (large_char(right_delimiter(p)) ≠ min_quarterword) then
begin print(",\u2022right-delimiter\u2022"); print_delimiter(right_delimiter(p));
end;
print_subsidary_data(numerator(p), "\\"); print_subsidary_data(denominator(p), "/");
end

```

This code is used in section 866.

874. That which can be displayed can also be destroyed.

$\langle \text{Cases of } flush_node_list \text{ that arise in mlists only } \rangle \equiv$

```

style_node: begin free_node(p, style_node_size); goto done;
end;
choice_node: begin flush_node_list(display_mlist(p)); flush_node_list(text_mlist(p));
    flush_node_list(script_mlist(p)); flush_node_list(script_script_mlist(p)); free_node(p, style_node_size);
    goto done;
end;
ord_noad, op_noad, bin_noad, rel_noad, open_noad, close_noad, punct_noad, inner_noad, radical_noad,
over_noad, under_noad, vcenter_noad, accent_noad:
begin if math_type(nucleus(p)) ≥ sub_box then flush_node_list(info(nucleus(p)));
if math_type(supscr(p)) ≥ sub_box then flush_node_list(info(supscr(p)));
if math_type(subscr(p)) ≥ sub_box then flush_node_list(info(subscr(p)));
if type(p) = radical_noad then free_node(p, radical_noad_size)
else if type(p) = accent_noad then free_node(p, accent_noad_size)
    else free_node(p, noad_size);
goto done;
end;
left_noad, right_noad: begin free_node(p, noad_size); goto done;
end;
fraction_noad: begin flush_node_list(info(numerator(p))); flush_node_list(info(denominator(p)));
    free_node(p, fraction_noad_size); goto done;
end;

```

This code is used in section 220.

875. Subroutines for math mode. In order to convert mlists to hlists, i.e., noads to nodes, we need several subroutines that are conveniently dealt with now.

Let us first introduce the macros that make it easy to get at the parameters and other font information. A size code, which is a multiple of 16, is added to a family number to get an index into the table of internal font numbers for each combination of family and size. (Be alert: Size codes get larger as the type gets smaller.)

```
define text_size = 0 { size code for the largest size in a family }
define script_size = 16 { size code for the medium size in a family }
define scriptscript_size = 32 { size code for the smallest size in a family }

< Basic printing procedures 57 > +≡
procedure print_size(s : integer);
begin if s = text_size then print_esc("textfont")
else if s = script_size then print_esc("scriptfont")
else print_esc("scriptscriptfont");
end;
```

876. Before an mlist is converted to an hlist, TeX makes sure that the fonts in family 2 have enough parameters to be math-symbol fonts, and that the fonts in family 3 have enough parameters to be math-extension fonts. The math-symbol parameters are referred to by using the following macros, which take a size code as their parameter; for example, *num1*(*cur_size*) gives the value of the *num1* parameter for the current size.

```
define mathsy_end(#) ≡ fam_fnt(2 + #) ].sc
define mathsy(#) ≡ font_info [ # + param_base [ mathsy_end
define math_x.height ≡ mathsy(5) { height of 'x' }
define math_quad ≡ mathsy(6) { 18mu }
define num1 ≡ mathsy(8) { numerator shift-up in display styles }
define num2 ≡ mathsy(9) { numerator shift-up in non-display, non-\atop }
define num3 ≡ mathsy(10) { numerator shift-up in non-display \atop }
define denom1 ≡ mathsy(11) { denominator shift-down in display styles }
define denom2 ≡ mathsy(12) { denominator shift-down in non-display styles }
define sup1 ≡ mathsy(13) { superscript shift-up in uncramped display style }
define sup2 ≡ mathsy(14) { superscript shift-up in uncramped non-display }
define sup3 ≡ mathsy(15) { superscript shift-up in cramped styles }
define sub1 ≡ mathsy(16) { subscript shift-down if superscript is absent }
define sub2 ≡ mathsy(17) { subscript shift-down if superscript is present }
define sup_drop ≡ mathsy(18) { superscript baseline below top of large box }
define sub_drop ≡ mathsy(19) { subscript baseline below bottom of large box }
define delim1 ≡ mathsy(20) { size of \atopwithdelims delimiters in display styles }
define delim2 ≡ mathsy(21) { size of \atopwithdelims delimiters in non-displays }
define axis_height ≡ mathsy(22) { height of fraction lines above the baseline }
define total_mathsy_params = 22
```

877. The math-extension parameters have similar macros, but the size code is omitted (since it is always *cur_size* when we refer to such parameters).

```
define mathex(#) ≡ font_info[# + param_base[fam_fnt(3 + cur_size)]].sc
define default_rule_thickness ≡ mathex(8) { thickness of \over bars }
define big_op_spacing1 ≡ mathex(9) { minimum clearance above a displayed op }
define big_op_spacing2 ≡ mathex(10) { minimum clearance below a displayed op }
define big_op_spacing3 ≡ mathex(11) { minimum baselineskip above displayed op }
define big_op_spacing4 ≡ mathex(12) { minimum baselineskip below displayed op }
define big_op_spacing5 ≡ mathex(13) { padding above and below displayed limits }
define total_mathex_params = 13
```

878. We also need to compute the change in style between mlists and their subsidiaries. The following macros define the subsidiary style for an overlined nucleus (*cramped_style*), for a subscript or a superscript (*sub_style* or *sup_style*), or for a numerator or denominator (*num_style* or *denom_style*).

```
define cramped_style(#) ≡ 2 * (# div 2) + cramped { cramp the style }
define sub_style(#) ≡ 2 * (# div 4) + script_style + cramped { smaller and cramped }
define sup_style(#) ≡ 2 * (# div 4) + script_style + (# mod 2) { smaller }
define num_style(#) ≡ # + 2 - 2 * (# div 6) { smaller unless already script-script }
define denom_style(#) ≡ 2 * (# div 2) + cramped + 2 - 2 * (# div 6) { smaller, cramped }
```

879. When the style changes, the following piece of program computes associated information:

```
<Set up the values of cur_size and cur_mu, based on cur_style 879> ≡
begin if cur_style < script_style then cur_size ← text_size
else cur_size ← 16 * ((cur_style - text_style) div 2);
cur_mu ← x_over_n(math_quad(cur_size), 18);
end
```

This code is used in sections 896, 902, 903, 906, 930, 936, 938, and 939.

880. Here is a function that returns a pointer to a rule node having a given thickness *t*. The rule will extend horizontally to the boundary of the vlist that eventually contains it.

```
function fraction_rule(t : scaled): pointer; { construct the bar for a fraction }
var p: pointer; { the new node }
begin p ← new_rule; height(p) ← t; depth(p) ← 0; fraction_rule ← p;
end;
```

881. The *overbar* function returns a pointer to a vlist box that consists of a given box *b*, above which has been placed a kern of height *k* under a fraction rule of thickness *t* under additional space of height *t*.

```
function overbar(b : pointer; k, t : scaled): pointer;
var p, q: pointer; { nodes being constructed }
begin p ← new_kern(k); link(p) ← b; q ← fraction_rule(t); link(q) ← p; p ← new_kern(t); link(p) ← q;
overbar ← vpack(p, natural);
end;
```

882. The *var_delimiter* function, which finds or constructs a sufficiently large delimiter, is the most interesting of the auxiliary functions that currently concern us. Given a pointer *d* to a delimiter field in some node, together with a size code *s* and a vertical distance *v*, this function returns a pointer to a box that contains the smallest variant of *d* whose height plus depth is *v* or more. (And if no variant is large enough, it returns the largest available variant.) In particular, this routine will construct arbitrarily large delimiters from extensible components, if *d* leads to such characters.

The value returned is a box whose *shift_amount* has been set so that the box is vertically centered with respect to the axis in the given size. If a built-up symbol is returned, the height of the box before shifting will be the height of its topmost component.

```
< Declare subprocedures for var_delimiter 885 >
function var_delimiter(d : pointer; s : small_number; v : scaled): pointer;
label found, continue;
var b: pointer; { the box that will be constructed }
f, g: internal_font_number; { best-so-far and tentative font codes }
c, x, y: quarterword; { best-so-far and tentative character codes }
m, n: integer; { the number of extensible pieces }
u: scaled; { height-plus-depth of a tentative character }
w: scaled; { largest height-plus-depth so far }
q: four_quarters; { character info }
hd: eight_bits; { height-depth byte }
r: four_quarters; { extensible pieces }
z: small_number; { runs through font family members }
large_attempt: boolean; { are we trying the "large" variant? }
begin f ← null_font; w ← 0; large_attempt ← false; z ← small_fam(d); x ← small_char(d);
loop begin < Look at the variants of (z, x); set f and c whenever a better character is found; goto
        found as soon as a large enough variant is encountered 883 >;
        if large_attempt then goto found; { there were none large enough }
        large_attempt ← true; z ← large_fam(d); x ← large_char(d);
        end;
found: if f ≠ null_font then < Make variable b point to a box for (f, c) 886 >
        else begin b ← new_null_box; width(b) ← null_delimiter_space;
                { use this width if no delimiter was found }
        end;
        shift_amount(b) ← half(height(b) – depth(b)) – axis_height(s); var_delimiter ← b;
end;
```

883. The search process is complicated slightly by the facts that some of the characters might not be present in some of the fonts, and they might not be probed in increasing order of height.

```
< Look at the variants of (z, x); set f and c whenever a better character is found; goto found as soon as a
        large enough variant is encountered 883 > ≡
if (z ≠ 0) ∨ (x ≠ min_quarterword) then
begin z ← z + s + 16;
repeat z ← z – 16; g ← fam_fnt(z);
        if g ≠ null_font then < Look at the list of characters starting with x in font g; set f and c whenever
                a better character is found; goto found as soon as a large enough variant is encountered 884 >;
until z < 16;
end
```

This code is used in section 882.

884. ⟨ Look at the list of characters starting with x in font g ; set f and c whenever a better character is found; **goto** *found* as soon as a large enough variant is encountered 884 ⟩ ≡

```

begin  $y \leftarrow x$ ;
if ( $qo(y) \geq font\_bc[g]$ )  $\wedge$  ( $qo(y) \leq font\_ec[g]$ ) then
  begin continue:  $q \leftarrow char\_info(g)(y)$ ;
  if char_exists( $q$ ) then
    begin if char_tag( $q$ ) = ext_tag then
      begin  $f \leftarrow g$ ;  $c \leftarrow y$ ; goto found;
      end;
       $hd \leftarrow height\_depth(q)$ ;  $u \leftarrow char\_height(g)(hd) + char\_depth(g)(hd)$ ;
      if  $u > w$  then
        begin  $f \leftarrow g$ ;  $c \leftarrow y$ ;  $w \leftarrow u$ ;
        if  $u \geq v$  then goto found;
        end;
      if char_tag( $q$ ) = list_tag then
        begin  $y \leftarrow rem\_byte(q)$ ; goto continue;
        end;
      end;
    end;
  end;
end

```

This code is used in section 883.

885. Here is a subroutine that creates a new box, whose list contains a single character, and whose width includes the italic correction for that character. The height or depth of the box will be negative, if the height or depth of the character is negative; thus, this routine may deliver a slightly different result than *hpack* would produce.

⟨ Declare subprocedures for *var_delimiter* 885 ⟩ ≡

```

function char_box( $f : internal\_font\_number$ ;  $c : quarterword$ ): pointer;
  var  $q : four\_quarters$ ;  $hd : eight\_bits$ ; { height_depth byte }
     $b, p : pointer$ ; { the new box and its character node }
  begin  $q \leftarrow char\_info(f)(c)$ ;  $hd \leftarrow height\_depth(q)$ ;  $b \leftarrow new\_null\_box$ ;
   $width(b) \leftarrow char\_width(f)(q) + char\_italic(f)(q)$ ;  $height(b) \leftarrow char\_height(f)(hd)$ ;
   $depth(b) \leftarrow char\_depth(f)(hd)$ ;  $p \leftarrow get\_avail$ ;  $character(p) \leftarrow c$ ;  $font(p) \leftarrow f$ ;  $list\_ptr(b) \leftarrow p$ ;
   $char\_box \leftarrow b$ ;
  end;

```

See also sections 887 and 888.

This code is used in section 882.

886. When the following code is executed, *char_tag*(q) will be equal to *ext_tag* if and only if a built-up symbol is supposed to be returned.

⟨ Make variable b point to a box for (f, c) 886 ⟩ ≡

```

if char_tag( $q$ ) = ext_tag then
  ⟨ Construct an extensible character in a new box  $b$ , using recipe rem_byte( $q$ ) and font  $f$  889 ⟩
else  $b \leftarrow char\_box(f, c)$ 

```

This code is used in section 882.

887. When we build an extensible character, it's handy to have the following subroutine, which puts a given character on top of the characters already in box b :

```
< Declare subprocedures for var_delimiter 885 > +≡
procedure stack_into_box( $b$  : pointer;  $f$  : internal_font_number;  $c$  : quarterword);
  var  $p$ : pointer; { new node placed into  $b$  }
  begin  $p \leftarrow \text{char\_box}(f, c)$ ;  $\text{link}(p) \leftarrow \text{list\_ptr}(b)$ ;  $\text{list\_ptr}(b) \leftarrow p$ ;  $\text{height}(b) \leftarrow \text{height}(p)$ ;
  end;
```

888. Another handy subroutine computes the height plus depth of a given character:

```
< Declare subprocedures for var_delimiter 885 > +≡
function height_plus_depth( $f$  : internal_font_number;  $c$  : quarterword): scaled;
  var  $q$ : four_quarters;  $hd$ : eight_bits; { height_depth byte }
  begin  $q \leftarrow \text{char\_info}(f)(c)$ ;  $hd \leftarrow \text{height\_depth}(q)$ ;
  height_plus_depth  $\leftarrow \text{char\_height}(f)(hd) + \text{char\_depth}(f)(hd)$ ;
  end;
```

889. { Construct an extensible character in a new box b , using recipe $\text{rem_byte}(q)$ and font f } ≡
begin $b \leftarrow \text{new_null_box}$; $\text{type}(b) \leftarrow \text{vlist_node}$; $r \leftarrow \text{font_info}[\text{exten_base}[f] + \text{rem_byte}(q)].qqqq$;
 { Compute the minimum suitable height, w , and the corresponding number of extension steps, n ; also set
 width(b) 890 };
 $c \leftarrow \text{ext_bot}(r)$;
if $c \neq \text{min_quarterword}$ **then** $\text{stack_into_box}(b, f, c)$;
 $c \leftarrow \text{ext_rep}(r)$;
for $m \leftarrow 1$ **to** n **do** $\text{stack_into_box}(b, f, c)$;
 $c \leftarrow \text{ext_mid}(r)$;
if $c \neq \text{min_quarterword}$ **then**
begin $\text{stack_into_box}(b, f, c)$; $c \leftarrow \text{ext_rep}(r)$;
for $m \leftarrow 1$ **to** n **do** $\text{stack_into_box}(b, f, c)$;
end;
 $c \leftarrow \text{ext_top}(r)$;
if $c \neq \text{min_quarterword}$ **then** $\text{stack_into_box}(b, f, c)$;
 $\text{depth}(b) \leftarrow w - \text{height}(b)$;
end

This code is used in section 886.

890. The width of an extensible character is the width of the repeatable module. If this module does not have positive height plus depth, we don't use any copies of it, otherwise we use as few as possible (in groups of two if there is a middle part).

⟨ Compute the minimum suitable height, w , and the corresponding number of extension steps, n ; also set

```

width(b) 890 } ≡
c ← ext_rep(r); u ← height_plus_depth(f, c); w ← 0; q ← char_info(f)(c);
width(b) ← char_width(f)(q) + char_italic(f)(q);
c ← ext_bot(r); if c ≠ min_quarterword then w ← w + height_plus_depth(f, c);
c ← ext_mid(r); if c ≠ min_quarterword then w ← w + height_plus_depth(f, c);
c ← ext_top(r); if c ≠ min_quarterword then w ← w + height_plus_depth(f, c);
n ← 0;
if u > 0 then
  while w < v do
    begin w ← w + u; incr(n);
    if ext_mid(r) ≠ min_quarterword then w ← w + u;
    end
  
```

This code is used in section 889.

891. The next subroutine is much simpler; it is used for numerators and denominators of fractions as well as for displayed operators and their limits above and below. It takes a given box b and changes it so that the new box is centered in a box of width w . The centering is done by putting \hss glue at the left and right of the list inside b , then packaging the new box; thus, the actual box might not really be centered, if it already contains infinite glue.

The given box might contain a single character whose italic correction has been added to the width of the box; in this case a compensating kern is inserted.

```

function rebox(b : pointer; w : scaled): pointer;
  var p: pointer; { temporary register for list manipulation }
  f: internal_font_number; { font in a one-character box }
  v: scaled; { width of a character without italic correction }
begin if (width(b) ≠ w) ∧ (list_ptr(b) ≠ null) then
  begin if type(b) = vlist_node then b ← hpack(b, natural);
  p ← list_ptr(b);
  if (is_char_node(p)) ∧ (link(p) = null) then
    begin f ← font(p); v ← char_width(f)(char_info(f)(character(p)));
    if v ≠ width(b) then link(p) ← new_kern(width(b) - v);
    end;
  free_node(b, box_node_size); b ← new_glue(ss_glue); link(b) ← p;
  while link(p) ≠ null do p ← link(p);
  link(p) ← new_glue(ss_glue); rebox ← hpack(b, w, exactly);
  end
else begin width(b) ← w; rebox ← b;
  end;
end; 
```

892. Here is a subroutine that creates a new glue specification from another one that is expressed in ‘mu’, given the value of the math unit.

```

define mu_mult (#) ≡ nx_plus_y(n, #, xn_over_d(#, f, '200000))

function math_glue(g : pointer; m : scaled): pointer;
  var p: pointer; { the new glue specification }
    n: integer; { integer part of m }
    f: scaled; { fraction part of m }
  begin n ← x_over_n(m, '200000); f ← remainder;
  if f < 0 then
    begin decr(n); f ← f + '200000;
    end;
  p ← get_node(glue_spec_size); width(p) ← mu_mult(width(g)); { convert mu to pt }
  stretch_order(p) ← stretch_order(g);
  if stretch_order(p) = normal then stretch(p) ← mu_mult(stretch(g))
  else stretch(p) ← stretch(g);
  shrink_order(p) ← shrink_order(g);
  if shrink_order(p) = normal then shrink(p) ← mu_mult(shrink(g))
  else shrink(p) ← shrink(g);
  math_glue ← p;
  end;

```

893. The *math_kern* subroutine removes *mu_glue* from a kern node, given the value of the math unit.

```

procedure math_kern(p : pointer; m : scaled);
  var n: integer; { integer part of m }
    f: scaled; { fraction part of m }
  begin if subtype(p) = mu_glue then
    begin n ← x_over_n(m, '200000); f ← remainder;
    if f < 0 then
      begin decr(n); f ← f + '200000;
      end;
    width(p) ← mu_mult(width(p)); subtype(p) ← explicit;
    end;
  end;

```

894. Sometimes it is necessary to destroy an mlist. The following subroutine empties the current list, assuming that *abs(mode) = mmode*.

```

procedure flush_math;
  begin flush_node_list(link(head)); flush_node_list(incompleat_noad); link(head) ← null; tail ← head;
  incompleat_noad ← null;
  end;

```

895. Typesetting math formulas. TeX's most important routine for dealing with formulas is called *mlist_to_hlist*. After a formula has been scanned and represented as an *mlist*, this routine converts it to an *hlist* that can be placed into a box or incorporated into the text of a paragraph. There are three implicit parameters, passed in global variables: *cur_mlist* points to the first node or noad in the given *mlist* (and it might be *null*); *cur_style* is a style code; and *mlist_penalties* is *true* if penalty nodes for potential line breaks are to be inserted into the resulting *hlist*. After *mlist_to_hlist* has acted, *link(temp_head)* points to the translated *hlist*.

Since *mlists* can be inside *mlists*, the procedure is recursive. And since this is not part of TeX's inner loop, the program has been written in a manner that stresses compactness over efficiency.

```
(Global variables 13) +≡
cur_mlist: pointer; { beginning of mlist to be translated }
cur_style: small_number; { style code at current place in the list }
cur_size: small_number; { size code corresponding to cur_style }
cur_mu: scaled; { the math unit width corresponding to cur_size }
mlist_penalties: boolean; { should mlist_to_hlist insert penalties? }
```

896. The recursion in *mlist_to_hlist* is due primarily to a subroutine called *clean_box* that puts a given noad field into a box using a given math style; *mlist_to_hlist* can call *clean_box*, which can call *mlist_to_hlist*.

The box returned by *clean_box* is "clean" in the sense that its *shift_amount* is zero.

```
procedure mlist_to_hlist; forward;
function clean_box(p: pointer; s: small_number): pointer;
label found;
var q: pointer; { beginning of a list to be boxed }
  save_style: small_number; { cur_style to be restored }
  x: pointer; { box to be returned }
  r: pointer; { temporary pointer }
begin case math_type(p) of
  math_char: begin cur_mlist ← new_noad; mem[nucleus(cur_mlist)] ← mem[p];
    end;
  sub_box: begin q ← info(p); goto found;
    end;
  sub_mlist: cur_mlist ← info(p);
  othercases begin q ← new_null_box; goto found;
    end
  endcases;
  save_style ← cur_style; cur_style ← s; mlist_penalties ← false;
  mlist_to_hlist; q ← link(temp_head); { recursive call }
  cur_style ← save_style; { restore the style }
  { Set up the values of cur_size and cur_mu, based on cur_style 879 };
found: if is_char_node(q) ∨ (q = null) then x ← hpack(q, natural)
  else if (link(q) = null) ∧ (type(q) ≤ vlist_node) ∧ (shift_amount(q) = 0) then x ← q
    { it's already clean }
  else x ← hpack(q, natural);
  { Simplify a trivial box 897 };
  clean_box ← x;
end;
```

897. Here we save memory space in a common case.

```
< Simplify a trivial box 897 > ≡
  q ← list_ptr(x);
  if is_char_node(q) then
    begin r ← link(q);
    if r ≠ null then
      if link(r) = null then
        if ¬is_char_node(r) then
          if type(r) = kern_node then { unneeded italic correction }
            begin free_node(r, small_node_size); link(q) ← null;
            end;
        end;
    end
```

This code is used in section 896.

898. It is convenient to have a procedure that converts a *math_char* field to an “unpacked” form. The *fetch* routine sets *cur_f*, *cur_c*, and *cur_i* to the font code, character code, and character information bytes of a given noad field. It also takes care of issuing error messages for nonexistent characters; in such cases, *char_exists*(*cur_i*) will be *false* after *fetch* has acted, and the field will also have been reset to *empty*.

```
procedure fetch(a : pointer); { unpack the math_char field a }
  begin cur_c ← character(a); cur_f ← fam_fnt(fam(a) + cur_size);
  if cur_f = null_font then < Complain about an undefined family and set cur_i null 899 >
  else begin if (qo(cur_c) ≥ font_bc[cur_f]) ∧ (qo(cur_c) ≤ font_ec[cur_f]) then
    cur_i ← char_info(cur_f)(cur_c)
  else cur_i ← null_character;
  if ¬(char_exists(cur_i)) then
    begin char_warning(cur_f, qo(cur_c)); math_type(a) ← empty; cur_i ← null_character;
  end;
  end;
end;
```

899. < Complain about an undefined family and set *cur_i* null 899 > ≡

```
begin print_err(""); print_size(cur_size); print_char(" "); print_int(fam(a));
print(" is undefined (character "); print_ASCII(qo(cur_c)); print_char(" "));
help4("Somewhere in the formula just ended, you used the")
("stated character from an undefined font family. For example, ")
("plain TeX doesn't allow \it or \sl in subscripts. Proceed, ")
("and I'll try to forget that I needed that character."); error; cur_i ← null_character;
math_type(a) ← empty;
end
```

This code is used in section 898.

900. The outputs of *fetch* are placed in global variables.

```
< Global variables 13 > +≡
cur_f: internal_font_number; { the font field of a math_char }
cur_c: quarterword; { the character field of a math_char }
cur_i: four_quarters; { the char_info of a math_char, or a lig/kern instruction }
```

901. We need to do a lot of different things, so *mlist_to_hlist* makes two passes over the given mlist.

The first pass does most of the processing: It removes “mu” spacing from glue, it recursively evaluates all subsidiary mlists so that only the top-level mlist remains to be handled, it puts fractions and square roots and such things into boxes, it attaches subscripts and superscripts, and it computes the overall height and depth of the top-level mlist so that the size of delimiters for a *left_noad* and a *right_noad* will be known. The hlist resulting from each noad is recorded in that noad’s *new_hlist* field, an integer field that replaces the *nucleus* or *thickness*.

The second pass eliminates all noads and inserts the correct glue and penalties between nodes.

```
define new_hlist (#) ≡ mem[nucleus(#)].int { the translation of an mlist }
```

902. Here is the overall plan of *mlist_to_hlist*, and the list of its local variables.

```
define done_with_noad = 80 { go here when a noad has been fully translated }
define done_with_node = 81 { go here when a node has been fully converted }
define check_dimensions = 82 { go here to update max_h and max_d }
define delete_q = 83 { go here to delete q and move to the next node }

⟨ Declare math construction procedures 910 ⟩
procedure mlist_to_hlist;
label reswitch, check_dimensions, done_with_noad, done_with_node, delete_q, done;
var mlist: pointer; { beginning of the given list }
penalties: boolean; { should penalty nodes be inserted? }
style: small_number; { the given style }
save_style: small_number; { holds cur_style during recursion }
q: pointer; { runs through the mlist }
r: pointer; { the most recent noad preceding q }
r_type: small_number; { the type of noad r, or op_noad if r = null }
t: small_number; { the effective type of noad q during the second pass }
p, x, y, z: pointer; { temporary registers for list construction }
pen: integer; { a penalty to be inserted }
s: small_number; { the size of a noad to be deleted }
max_h, max_d: scaled; { maximum height and depth of the list translated so far }
delta: scaled; { offset between subscript and superscript }
begin mlist ← cur_mlist; penalties ← mlist_penalties; style ← cur_style;
{ tuck global parameters away as local variables }
q ← mlist; r ← null; r_type ← op_noad; max_h ← 0; max_d ← 0;
{ Set up the values of cur_size and cur_mu, based on cur_style 879 };
while q ≠ null do { 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 903 };
{ Convert a final bin_noad to an ord_noad 905 };
{ Make a second pass over the mlist, removing all noads and inserting the proper spacing and
penalties 936 };
end;
```

903. 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 903 ⟩ ≡

```

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 904 ⟩;
 $check\_dimensions$ :  $z \leftarrow hpack(new\_hlist(q), natural)$ ;
  if  $height(z) > max\_h$  then  $max\_h \leftarrow height(z)$ ;
  if  $depth(z) > max\_d$  then  $max\_d \leftarrow depth(z)$ ;
  free_node( $z$ ,  $box\_node\_size$ );
 $done\_with\_noad$ :  $r \leftarrow q$ ;  $r\_type \leftarrow type(r)$ ;
  if  $r\_type = right\_noad$  then
    begin  $r\_type \leftarrow left\_noad$ ;  $cur\_style \leftarrow style$ ;
    ⟨ Set up the values of  $cur\_size$  and  $cur\_mu$ , based on  $cur\_style$  879 ⟩;
    end;
 $done\_with\_node$ :  $q \leftarrow link(q)$ ;
end
  
```

This code is used in section 902.

904. One of the things we must do on the first pass is change a bin_noad to an ord_noad if the bin_noad is not in the context of a binary operator. The values of r and r_type make this fairly easy.

```

⟨ 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 904 ⟩ ≡
reswitch:  $\delta \leftarrow 0$ ;
case  $type(q)$  of
  bin_noad: case  $r\_type$  of
    bin_noad, op_noad, rel_noad, open_noad, punct_noad, left_noad: begin  $type(q) \leftarrow ord\_noad$ ;
      goto reswitch;
    end;
    othercases do_nothing
  endcases;
  rel_noad, close_noad, punct_noad, right_noad: begin
    ⟨ Convert a final  $bin\_noad$  to an  $ord\_noad$  905 ⟩;
    if  $type(q) = right\_noad$  then goto  $done\_with\_noad$ ;
  end;
  ⟨ Cases for noads that can follow a  $bin\_noad$  909 ⟩
  ⟨ Cases for nodes that can appear in an mlist, after which we goto  $done\_with\_node$  906 ⟩
  othercases confusion("mlist1")
  endcases;
  ⟨ Convert  $nucleus(q)$  to an hlist and attach the sub/superscripts 930 ⟩
  
```

This code is used in section 903.

905. ⟨ Convert a final bin_noad to an ord_noad 905 ⟩ ≡

```

if  $r\_type = bin\_noad$  then  $type(r) \leftarrow ord\_noad$ 
  
```

This code is used in sections 902 and 904.

906. ⟨Cases for nodes that can appear in an mlist, after which we **goto** *done_with_node* 906⟩ ≡

```

style_node: begin cur_style ← subtype(q);
  ⟨ Set up the values of cur_size and cur_mu, based on cur_style 879 ⟩;
  goto done_with_node;
end;

choice_node: ⟨ Change this node to a style node followed by the correct choice, then goto
  done_with_node 907 ⟩;
ins_node, mark_node, adjust_node, whatsit_node, penalty_node, disc_node: goto done_with_node;
rule_node: begin if height(q) > max_h then max_h ← height(q);
  if depth(q) > max_d then max_d ← depth(q);
  goto done_with_node;
end;

glue_node: begin ⟨ Convert math glue to ordinary glue 908 ⟩;
  goto done_with_node;
end;

kern_node: begin math_kern(q, cur_mu); goto done_with_node;
end;
```

This code is used in section 904.

907. **define** *choose_mlist*(#) ≡

```

begin p ← #(q); #(q) ← null; end

⟨ Change this node to a style node followed by the correct choice, then goto done_with_node 907 ⟩ ≡
begin case cur_style div 2 of
  0: choose_mlist(display_mlist); { display_style = 0 }
  1: choose_mlist(text_mlist); { text_style = 2 }
  2: choose_mlist(script_mlist); { script_style = 4 }
  3: choose_mlist(script_script_mlist); { script_script_style = 6 }
end; { there are no other cases }
flush_node_list(display_mlist(q)); flush_node_list(text_mlist(q)); flush_node_list(script_mlist(q));
flush_node_list(script_script_mlist(q));
type(q) ← style_node; subtype(q) ← cur_style; width(q) ← 0; depth(q) ← 0;
if p ≠ null then
  begin z ← link(q); link(q) ← p;
  while link(p) ≠ null do p ← link(p);
  link(p) ← z;
  end;
goto done_with_node;
end
```

This code is used in section 906.

908. Conditional math glue ('\nonscript') results in a *glue_node* pointing to *zero_glue*, with *subtype(q) = cond_math_glue*; in such a case the node following will be eliminated if it is a glue or kern node and if the current size is different from *text_size*. Unconditional math glue ('\muskip') is converted to normal glue by multiplying the dimensions by *cur_mu*.

```
<Convert math glue to ordinary glue 908> ≡
  if subtype(q) = mu_glue then
    begin x ← glue_ptr(q); y ← math_glue(x, cur_mu); delete_glue_ref(x); glue_ptr(q) ← y;
    subtype(q) ← normal;
    end
  else if (cur_size ≠ text_size) ∧ (subtype(q) = cond_math_glue) then
    begin p ← link(q);
    if p ≠ null then
      if (type(p) = glue_node) ∨ (type(p) = kern_node) then
        begin link(q) ← link(p); link(p) ← null; flush_node_list(p);
        end;
    end
```

This code is used in section 906.

909. <Cases for noads that can follow a bin_noad 909> ≡

```
left_noad: goto done_with_noad;
fraction_noad: begin make_fraction(q); goto check_dimensions;
end;
op_noad: begin delta ← make_op(q);
if subtype(q) = limits then goto check_dimensions;
end;
ord_noad: make_ord(q);
open_noad, inner_noad: do_nothing;
radical_noad: make_radical(q);
over_noad: make_over(q);
under_noad: make_under(q);
accent_noad: make_math_accent(q);
vcenter_noad: make_vcenter(q);
```

This code is used in section 904.

910. Most of the actual construction work of *mlist_to_hlist* is done by procedures with names like *make_fraction*, *make_radical*, etc. To illustrate the general setup of such procedures, let's begin with a couple of simple ones.

<Declare math construction procedures 910> ≡

```
procedure make_over(q : pointer);
begin info(nucleus(q)) ← overbar(clean_box(nucleus(q), cramped_style(cur_style)),
  3 * default_rule_thickness, default_rule_thickness); math_type(nucleus(q)) ← sub_box;
end;
```

See also sections 911, 912, 913, 914, 919, 925, 928, 932, and 938.

This code is used in section 902.

911. \langle Declare math construction procedures 910 $\rangle + \equiv$

```
procedure make_under(q : pointer);
  var p, x, y: pointer; { temporary registers for box construction }
  delta: scaled; { overall height plus depth }
  begin x ← clean_box(nucleus(q), cur_style); p ← new_kern(3 * default_rule_thickness); link(x) ← p;
  link(p) ← fraction_rule(default_rule_thickness); y ← vpack(x, natural);
  delta ← height(y) + depth(y) + default_rule_thickness; height(y) ← height(x);
  depth(y) ← delta - height(y); info(nucleus(q)) ← y; math_type(nucleus(q)) ← sub_box;
  end;
```

912. \langle Declare math construction procedures 910 $\rangle + \equiv$

```
procedure make_vcenter(q : pointer);
  var v: pointer; { the box that should be centered vertically }
  delta: scaled; { its height plus depth }
  begin v ← info(nucleus(q));
  if type(v) ≠ vlist_node then confusion("vcenter");
  delta ← height(v) + depth(v); height(v) ← axis_height(cur_size) + half(delta);
  depth(v) ← delta - height(v);
  end;
```

913. According to the rules in the DVI file specifications, we ensure alignment between a square root sign and the rule above its nucleus by assuming that the baseline of the square-root symbol is the same as the bottom of the rule. The height of the square-root symbol will be the thickness of the rule, and the depth of the square-root symbol should exceed or equal the height-plus-depth of the nucleus plus a certain minimum clearance *clr*. The symbol will be placed so that the actual clearance is *clr* plus half the excess.

\langle Declare math construction procedures 910 $\rangle + \equiv$

```
procedure make_radical(q : pointer);
  var x, y: pointer; { temporary registers for box construction }
  delta, clr: scaled; { dimensions involved in the calculation }
  begin x ← clean_box(nucleus(q), cramped_style(cur_style));
  if cur_style < text_style then { display style }
    clr ← default_rule_thickness + (abs(math_x_height(cur_size)) div 4)
  else begin clr ← default_rule_thickness; clr ← clr + (abs(clr) div 4);
  end;
  y ← var_delimiter(left_delimiter(q), cur_size, height(x) + depth(x) + clr + default_rule_thickness);
  delta ← depth(y) - (height(x) + depth(x) + clr);
  if delta > 0 then clr ← clr + half(delta); { increase the actual clearance }
  shift_amount(y) ← -(height(x) + clr); link(y) ← overbar(x, clr, height(y));
  info(nucleus(q)) ← hpack(y, natural); math_type(nucleus(q)) ← sub_box;
  end;
```

914. Slants are not considered when placing accents in math mode. The accenter is centered over the accentee, and the accent width is treated as zero with respect to the size of the final box.

```

⟨ Declare math construction procedures 910 ⟩ +≡
procedure make_math_accent(q : pointer);
label done, done1;
var p, x, y: pointer; { temporary registers for box construction }
a: integer; { address of lig/kern instruction }
c: quarterword; { accent character }
f: internal_font_number; { its font }
i: four_quarters; { its char_info }
s: scaled; { amount to skew the accent to the right }
h: scaled; { height of character being accented }
delta: scaled; { space to remove between accent and accentee }
w: scaled; { width of the accentee, not including sub/superscripts }
begin fetch(accent_chr(q));
if char_exists(cur_i) then
begin i ← cur_i; c ← cur_c; f ← cur_f;
⟨ Compute the amount of skew 917 ⟩;
x ← clean_box(nucleus(q), cramped_style(cur_style)); w ← width(x); h ← height(x);
⟨ Switch to a larger accent if available and appropriate 916 ⟩;
if h < x_height(f) then delta ← h else delta ← x.height(f);
if (math_type(supscr(q)) ≠ empty) ∨ (math_type(subscr(q)) ≠ empty) then
  if math_type(nucleus(q)) = math_char then ⟨ Swap the subscript and superscript into box x 918 ⟩;
  y ← char_box(f, c); shift_amount(y) ← s + half(w - width(y)); width(y) ← 0; p ← new_kern(-delta);
  link(p) ← x; link(y) ← p; y ← vpack(y, natural); width(y) ← width(x);
  if height(y) < h then ⟨ Make the height of box y equal to h 915 ⟩;
  info(nucleus(q)) ← y; math_type(nucleus(q)) ← sub_box;
end;
end;
end;
```

915. ⟨ Make the height of box y equal to h 915 ⟩ ≡

```

begin p ← new_kern(h - height(y)); link(p) ← list_ptr(y); list_ptr(y) ← p; height(y) ← h;
end
```

This code is used in section 914.

916. ⟨ Switch to a larger accent if available and appropriate 916 ⟩ ≡

```

loop begin if char_tag(i) ≠ list_tag then goto done;
y ← rem_byte(i); i ← char_info(f)(y);
if ¬char_exists(i) then goto done;
if char_width(f)(i) > w then goto done;
c ← y;
end;
done:
```

This code is used in section 914.

917. ⟨ Compute the amount of skew 917 ⟩ ≡

```

 $s \leftarrow 0;$ 
 $\text{if } \text{math\_type}(\text{nucleus}(q)) = \text{math\_char} \text{ then}$ 
 $\quad \text{begin } \text{fetch}(\text{nucleus}(q));$ 
 $\quad \text{if } \text{char\_tag}(\text{cur\_i}) = \text{lig\_tag} \text{ then}$ 
 $\quad \quad \text{begin } a \leftarrow \text{lig\_kern\_start}(\text{cur\_f})(\text{cur\_i}); \text{ cur\_i} \leftarrow \text{font\_info}[a].qqqq;$ 
 $\quad \quad \text{if } \text{skip\_byte}(\text{cur\_i}) > \text{stop\_flag} \text{ then}$ 
 $\quad \quad \quad \text{begin } a \leftarrow \text{lig\_kern\_restart}(\text{cur\_f})(\text{cur\_i}); \text{ cur\_i} \leftarrow \text{font\_info}[a].qqqq;$ 
 $\quad \quad \quad \text{end};$ 
 $\quad \text{loop begin if } qo(\text{next\_char}(\text{cur\_i})) = \text{skew\_char}[\text{cur\_f}] \text{ then}$ 
 $\quad \quad \quad \text{begin if } op\_byte(\text{cur\_i}) \geq \text{kern\_flag} \text{ then}$ 
 $\quad \quad \quad \quad \text{if } \text{skip\_byte}(\text{cur\_i}) \leq \text{stop\_flag} \text{ then } s \leftarrow \text{char\_kern}(\text{cur\_f})(\text{cur\_i});$ 
 $\quad \quad \quad \quad \text{goto done1;}$ 
 $\quad \quad \quad \text{end};$ 
 $\quad \quad \quad \text{if } \text{skip\_byte}(\text{cur\_i}) \geq \text{stop\_flag} \text{ then goto done1;}$ 
 $\quad \quad \quad a \leftarrow a + qo(\text{skip\_byte}(\text{cur\_i})) + 1; \text{ cur\_i} \leftarrow \text{font\_info}[a].qqqq;$ 
 $\quad \quad \quad \text{end};$ 
 $\quad \text{end};$ 
 $\text{end};$ 
 $\text{done1:}$ 
```

This code is used in section 914.

918. ⟨ Swap the subscript and superscript into box x 918 ⟩ ≡

```

 $\text{begin flush\_node\_list}(x); x \leftarrow \text{new\_noad}; \text{mem}[\text{nucleus}(x)] \leftarrow \text{mem}[\text{nucleus}(q)];$ 
 $\text{mem}[\text{supscr}(x)] \leftarrow \text{mem}[\text{supscr}(q)]; \text{mem}[\text{subscr}(x)] \leftarrow \text{mem}[\text{subscr}(q)];$ 
 $\text{mem}[\text{supscr}(q)].hh \leftarrow \text{empty\_field}; \text{mem}[\text{subscr}(q)].hh \leftarrow \text{empty\_field};$ 
 $\text{math\_type}(\text{nucleus}(q)) \leftarrow \text{sub\_mlist}; \text{info}(\text{nucleus}(q)) \leftarrow x; x \leftarrow \text{clean\_box}(\text{nucleus}(q), \text{cur\_style});$ 
 $\text{delta} \leftarrow \text{delta} + \text{height}(x) - h; h \leftarrow \text{height}(x);$ 
 $\text{end}$ 
```

This code is used in section 914.

919. The *make_fraction* procedure is a bit different because it sets *new_hlist*(*q*) directly rather than making a sub-box.

⟨ Declare math construction procedures 910 ⟩ +≡

```

procedure make_fraction(q : pointer);
  var p, v, x, y, z: pointer; { temporary registers for box construction }
  delta, delta1, delta2, shift_up, shift_down, clr: scaled; { dimensions for box calculations }
  begin if thickness(q) = default_code then thickness(q) ← default_rule.thickness;
  { Create equal-width boxes x and z for the numerator and denominator, and compute the default amounts
    shift_up and shift_down by which they are displaced from the baseline 920 };
  if thickness(q) = 0 then { Adjust shift_up and shift_down for the case of no fraction line 921 };
  else { Adjust shift_up and shift_down for the case of a fraction line 922 };
  { Construct a vlist box for the fraction, according to shift_up and shift_down 923 };
  { Put the fraction into a box with its delimiters, and make new_hlist(q) point to it 924 };
  end;
```

920. ⟨ Create equal-width boxes x and z for the numerator and denominator, and compute the default amounts $shift_up$ and $shift_down$ by which they are displaced from the baseline 920 ⟩ ≡

```

 $x \leftarrow clean\_box(numerator(q), num\_style(cur\_style));$ 
 $z \leftarrow clean\_box(denominator(q), denom\_style(cur\_style));$ 
 $\text{if } width(x) < width(z) \text{ then } x \leftarrow rebox(x, width(z))$ 
 $\text{else } z \leftarrow rebox(z, width(x));$ 
 $\text{if } cur\_style < text\_style \text{ then } \{ \text{display style} \}$ 
 $\quad \begin{aligned} \text{begin } shift\_up &\leftarrow num1(cur\_size); shift\_down &\leftarrow denom1(cur\_size); \\ &\text{end} \end{aligned}$ 
 $\text{else begin } shift\_down &\leftarrow denom2(cur\_size);$ 
 $\quad \begin{aligned} \text{if } thickness(q) &\neq 0 \text{ then } shift\_up &\leftarrow num2(cur\_size) \\ &\text{else } shift\_up &\leftarrow num3(cur\_size); \\ &\text{end} \end{aligned}$ 

```

This code is used in section 919.

921. The numerator and denominator must be separated by a certain minimum clearance, called clr in the following program. The difference between clr and the actual clearance is twice $delta$.

⟨ Adjust $shift_up$ and $shift_down$ for the case of no fraction line 921 ⟩ ≡

```

 $\begin{aligned} \text{begin if } cur\_style < text\_style \text{ then } clr &\leftarrow 7 * default\_rule\_thickness \\ \text{else } clr &\leftarrow 3 * default\_rule\_thickness; \\ delta &\leftarrow half(clr - ((shift\_up - depth(x)) - (height(z) - shift\_down))); \\ \text{if } delta > 0 \text{ then } \\ \quad \begin{aligned} \text{begin } shift\_up &\leftarrow shift\_up + delta; shift\_down &\leftarrow shift\_down + delta; \\ &\text{end}; \\ &\text{end} \end{aligned} \end{aligned}$ 

```

This code is used in section 919.

922. In the case of a fraction line, the minimum clearance depends on the actual thickness of the line.

⟨ Adjust $shift_up$ and $shift_down$ for the case of a fraction line 922 ⟩ ≡

```

 $\begin{aligned} \text{begin if } cur\_style < text\_style \text{ then } clr &\leftarrow 3 * thickness(q) \\ \text{else } clr &\leftarrow thickness(q); \\ delta &\leftarrow half(thickness(q)); delta1 &\leftarrow clr - ((shift\_up - depth(x)) - (axis\_height(cur\_size) + delta)); \\ delta2 &\leftarrow clr - ((axis\_height(cur\_size) - delta) - (height(z) - shift\_down)); \\ \text{if } delta1 > 0 \text{ then } shift\_up &\leftarrow shift\_up + delta1; \\ \text{if } delta2 > 0 \text{ then } shift\_down &\leftarrow shift\_down + delta2; \\ &\text{end} \end{aligned}$ 

```

This code is used in section 919.

923. ⟨ Construct a vlist box for the fraction, according to $shift_up$ and $shift_down$ 923 ⟩ ≡

```

 $v \leftarrow new\_null\_box; type(v) \leftarrow vlist\_node; height(v) \leftarrow shift\_up + height(x);$ 
 $depth(v) \leftarrow depth(z) + shift\_down; width(v) \leftarrow width(x); \{ \text{this also equals } width(z) \}$ 
 $\text{if } thickness(q) = 0 \text{ then }$ 
 $\quad \begin{aligned} \text{begin } p &\leftarrow new\_kern((shift\_up - depth(x)) - (height(z) - shift\_down)); link(p) &\leftarrow z; \\ &\text{end} \end{aligned}$ 
 $\text{else begin } y &\leftarrow fraction\_rule(thickness(q));$ 
 $\quad \begin{aligned} p &\leftarrow new\_kern((axis\_height(cur\_size) - delta) - (height(z) - shift\_down)); \\ link(y) &\leftarrow p; link(p) &\leftarrow z; \\ p &\leftarrow new\_kern((shift\_up - depth(x)) - (axis\_height(cur\_size) + delta)); link(p) &\leftarrow y; \\ &\text{end}; \end{aligned}$ 
 $link(x) \leftarrow p; list\_ptr(v) \leftarrow x$ 

```

This code is used in section 919.

```

924. ⟨ Put the fraction into a box with its delimiters, and make new_hlist(q) point to it 924 ⟩ ≡
  if cur_style < text_style then delta ← delim1(cur_size)
  else delta ← delim2(cur_size);
  x ← var_delimiter(left_delimiter(q), cur_size, delta); link(x) ← v;
  z ← var_delimiter(right_delimiter(q), cur_size, delta); link(v) ← z;
  new_hlist(q) ← hpack(x, natural)

```

This code is used in section 919.

925. If the nucleus of an *op_noad* is a single character, it is to be centered vertically with respect to the axis, after first being enlarged (via a character list in the font) if we are in display style. The normal convention for placing displayed limits is to put them above and below the operator in display style.

The italic correction is removed from the character if there is a subscript and the limits are not being displayed. The *make_op* routine returns the value that should be used as an offset between subscript and superscript.

After *make_op* has acted, *subtype*(*q*) will be *limits* if and only if the limits have been set above and below the operator. In that case, *new_hlist*(*q*) will already contain the desired final box.

```

⟨ Declare math construction procedures 910 ⟩ +≡
function make_op(q : pointer): scaled;
  var delta: scaled; { offset between subscript and superscript }
  p, v, x, y, z: pointer; { temporary registers for box construction }
  c: quarterword; i: four-quarters; { registers for character examination }
  shift_up, shift_down: scaled; { dimensions for box calculation }
begin if (subtype(q) = normal) ∧ (cur_style < text_style) then subtype(q) ← limits;
if math_type(nucleus(q)) = math_char then
  begin fetch(nucleus(q));
  if (cur_style < text_style) ∧ (char_tag(cur_i) = list_tag) then { make it larger }
    begin c ← rem_byte(cur_i); i ← char_info(cur_f)(c);
    if char_exists(i) then
      begin cur_c ← c; cur_i ← i; character(nucleus(q)) ← c;
      end;
    end;
  delta ← char_italic(cur_f)(cur_i); x ← clean_box(nucleus(q), cur_style);
  if (math_type(subscr(q)) ≠ empty) ∧ (subtype(q) ≠ limits) then width(x) ← width(x) − delta;
    { remove italic correction }
  shift_amount(x) ← half(height(x) − depth(x)) − axis_height(cur_size); { center vertically }
  math_type(nucleus(q)) ← sub_box; info(nucleus(q)) ← x;
  end
else delta ← 0;
if subtype(q) = limits then ⟨ Construct a box with limits above and below it, skewed by delta 926 ⟩;
  make_op ← delta;
end;

```

926. The following program builds a vlist box v for displayed limits. The width of the box is not affected by the fact that the limits may be skewed.

```
< Construct a box with limits above and below it, skewed by delta 926 > ≡
begin x ← clean_box(supscr(q), sup_style(cur_style)); y ← clean_box(nucleus(q), cur_style);
z ← clean_box(subscr(q), sub_style(cur_style)); v ← new_null_box; type(v) ← vlist_node;
width(v) ← width(y);
if width(x) > width(v) then width(v) ← width(x);
if width(z) > width(v) then width(v) ← width(z);
x ← rebox(x, width(v)); y ← rebox(y, width(v)); z ← rebox(z, width(v));
shift_amount(x) ← half(delta); shift_amount(z) ← -shift_amount(x); height(v) ← height(y);
depth(v) ← depth(y);
< Attach the limits to y and adjust height(v), depth(v) to account for their presence 927 >;
new_hlist(q) ← v;
end
```

This code is used in section 925.

927. We use $shift_up$ and $shift_down$ in the following program for the amount of glue between the displayed operator y and its limits x and z . The vlist inside box v will consist of x followed by y followed by z , with kern nodes for the spaces between and around them.

```
< Attach the limits to y and adjust height(v), depth(v) to account for their presence 927 > ≡
if math_type(supscr(q)) = empty then
begin free_node(x, box_node_size); list_ptr(v) ← y;
end
else begin shift_up ← big_op_spacing3 – depth(x);
if shift_up < big_op_spacing1 then shift_up ← big_op_spacing1;
p ← new_kern(shift_up); link(p) ← y; link(x) ← p;
p ← new_kern(big_op_spacing5); link(p) ← x; list_ptr(v) ← p;
height(v) ← height(v) + big_op_spacing5 + height(x) + depth(x) + shift_up;
end;
if math_type(subscr(q)) = empty then free_node(z, box_node_size)
else begin shift_down ← big_op_spacing4 – height(z);
if shift_down < big_op_spacing2 then shift_down ← big_op_spacing2;
p ← new_kern(shift_down); link(y) ← p; link(p) ← z;
p ← new_kern(big_op_spacing5); link(z) ← p;
depth(v) ← depth(v) + big_op_spacing5 + height(z) + depth(z) + shift_down;
end
```

This code is used in section 926.

928. A ligature found in a math formula does not create a *ligature_node*, because there is no question of hyphenation afterwards; the ligature will simply be stored in an ordinary *char_node*, after residing in an *ord_noad*.

The *math_type* is converted to *math_text_char* here if we would not want to apply an italic correction to the current character unless it belongs to a math font (i.e., a font with *space* = 0).

No boundary characters enter into these ligatures.

```
< Declare math construction procedures 910 > +≡
procedure make_ord(q : pointer);
label restart, exit;
var a: integer; { address of lig/kern instruction }
p, r: pointer; { temporary registers for list manipulation }
begin restart:
if math_type(subscr(q)) = empty then
  if math_type(supscr(q)) = empty then
    if math_type(nucleus(q)) = math_char then
      begin p ← link(q);
      if p ≠ null then
        if (type(p) ≥ ord_noad) ∧ (type(p) ≤ punct_noad) then
          if math_type(nucleus(p)) = math_char then
            if fam(nucleus(p)) = fam(nucleus(q)) then
              begin math_type(nucleus(q)) ← math_text_char; fetch(nucleus(q));
              if char_tag(cur_i) = lig_tag then
                begin a ← lig_kern_start(cur_f)(cur_i); cur_c ← character(nucleus(p));
                cur_i ← font_info[a].qqqq;
                if skip_byte(cur_i) > stop_flag then
                  begin a ← lig_kern_restart(cur_f)(cur_i); cur_i ← font_info[a].qqqq;
                  end;
                loop begin < If instruction cur_i is a kern with cur_c, attach the kern after q; or if it is
                  a ligature with cur_c, combine noads q and p appropriately; then return if the
                  cursor has moved past a noad, or goto restart 929 >;
                  if skip_byte(cur_i) ≥ stop_flag then return;
                  a ← a + qo(skip_byte(cur_i)) + 1; cur_i ← font_info[a].qqqq;
                  end;
                end;
              end;
            end;
          end;
        end;
      end;
    end;
  end;
exit: end;
```

929. Note that a ligature between an *ord_noad* and another kind of noad is replaced by an *ord_noad*, when the two noads collapse into one. But we could make a parenthesis (say) change shape when it follows certain letters. Presumably a font designer will define such ligatures only when this convention makes sense.

```

⟨ If instruction cur_i is a kern with cur_c, attach the kern after q; or if it is a ligature with cur_c,
    combine noads q and p appropriately; then return if the cursor has moved past a noad, or goto
    restart 929 ⟩ ≡
if next_char(cur_i) = cur_c then
  if skip_byte(cur_i) ≤ stop_flag then
    if op_byte(cur_i) ≥ kern_flag then
      begin p ← new_kern(char_kern(cur_f)(cur_i)); link(p) ← link(q); link(q) ← p; return;
      end
    else begin check_interrupt; { allow a way out of infinite ligature loop }
      case op_byte(cur_i) of
        qi(1), qi(5): character(nucleus(q)) ← rem_byte(cur_i); { =: |, =: |> }
        qi(2), qi(6): character(nucleus(p)) ← rem_byte(cur_i); { |=:, |=:> }
        qi(3), qi(7), qi(11): begin r ← new_noad; { |=: |, |=: |>, |=: |>> }
          character(nucleus(r)) ← rem_byte(cur_i); fam(nucleus(r)) ← fam(nucleus(q));
          link(q) ← r; link(r) ← p;
          if op_byte(cur_i) < qi(11) then math_type(nucleus(r)) ← math_char
          else math_type(nucleus(r)) ← math_text_char; { prevent combination }
          end;
        othercases begin link(q) ← link(p); character(nucleus(q)) ← rem_byte(cur_i); { =: }
          mem[subscr(q)] ← mem[subscr(p)]; mem[supscr(q)] ← mem[supscr(p)];
          free_node(p, noad_size);
          end
        endcases;
        if op_byte(cur_i) > qi(3) then return;
        math_type(nucleus(q)) ← math_char; goto restart;
      end

```

This code is used in section 928.

930. When we get to the following part of the program, we have “fallen through” from cases that did not lead to *check_dimensions* or *done_with_noad* or *done_with_node*. Thus, *q* points to a noad whose nucleus may need to be converted to an hlist, and whose subscripts and superscripts need to be appended if they are present.

If *nucleus(q)* is not a *math_char*, the variable *delta* is the amount by which a superscript should be moved right with respect to a subscript when both are present.

```
<Convert nucleus(q) to an hlist and attach the sub/superscripts 930> ≡
  case math_type(nucleus(q)) of
    math_char, math_text_char: <Create a character node p for nucleus(q), possibly followed by a kern node
      for the italic correction, and set delta to the italic correction if a subscript is present 931>;
    empty: p ← null;
    sub_box: p ← info(nucleus(q));
    sub_mlist: begin cur_mlist ← info(nucleus(q)); save_style ← cur_style; mlist_penalties ← false;
      mlist_to_hlist; { recursive call }
      cur_style ← save_style; { Set up the values of cur_size and cur_mu, based on cur_style 879 };
      p ← hpack(link(temp_head), natural);
    end;
    othercases confusion("mlist2")
  endcases;
  new_hlist(q) ← p;
  if (math_type(subscr(q)) = empty) ∧ (math_type(supscr(q)) = empty) then goto check_dimensions;
  make_scripts(q, delta)
```

This code is used in section 904.

931. <Create a character node *p* for *nucleus(q)*, possibly followed by a kern node for the italic correction, and set *delta* to the italic correction if a subscript is present 931> ≡

```
begin fetch(nucleus(q));
if char_exists(cur_i) then
  begin delta ← char_italic(cur_f)(cur_i); p ← new_character(cur_f, qo(cur_c));
  if (math_type(nucleus(q)) = math_text_char) ∧ (space(cur_f) ≠ 0) then delta ← 0;
    { no italic correction in mid-word of text font }
  if (math_type(subscr(q)) = empty) ∧ (delta ≠ 0) then
    begin link(p) ← new_kern(delta); delta ← 0;
    end;
  end
else p ← null;
end
```

This code is used in section 930.

932. The purpose of *make_scripts*(*q, delta*) is to attach the subscript and/or superscript of node *q* to the list that starts at *new_hlist*(*q*), given that the subscript and superscript aren't both empty. The superscript will appear to the right of the subscript by a given distance *delta*.

We set *shift_down* and *shift_up* to the minimum amounts to shift the baseline of subscripts and superscripts based on the given nucleus.

```

⟨ Declare math construction procedures 910 ⟩ +≡
procedure make_scripts(q : pointer; delta : scaled);
  var p, x, y, z: pointer; { temporary registers for box construction }
    shift_up, shift_down, clr: scaled; { dimensions in the calculation }
    t: small_number; { subsidiary size code }
  begin p ← new_hlist(q);
    if is_char_node(p) then
      begin shift_up ← 0; shift_down ← 0;
      end
    else begin z ← hpack(p, natural);
      if cur_style < script_style then t ← script_size else t ← script_script_size;
      shift_up ← height(z) - sup_drop(t); shift_down ← depth(z) + sub_drop(t); free_node(z, box_node_size);
      end;
    if math_type(supscr(q)) = empty then ⟨ Construct a subscript box x when there is no superscript 933 ⟩
    else begin ⟨ Construct a superscript box x 934 ⟩;
      if math_type(subscr(q)) = empty then shift_amount(x) ← -shift_up
      else ⟨ Construct a sub/superscript combination box x, with the superscript offset by delta 935 ⟩;
      end;
    if new_hlist(q) = null then new_hlist(q) ← x
    else begin p ← new_hlist(q);
      while link(p) ≠ null do p ← link(p);
      link(p) ← x;
      end;
    end;
  end;
```

933. When there is a subscript without a superscript, the top of the subscript should not exceed the baseline plus four-fifths of the x-height.

```

⟨ Construct a subscript box x when there is no superscript 933 ⟩ ≡
begin x ← clean_box(subscr(q), sub_style(cur_style)); width(x) ← width(x) + script_space;
  if shift_down < sub1(cur_size) then shift_down ← sub1(cur_size);
  clr ← height(x) - (abs(math_x_height(cur_size) * 4) div 5);
  if shift_down < clr then shift_down ← clr;
  shift_amount(x) ← shift_down;
end
```

This code is used in section 932.

934. The bottom of a superscript should never descend below the baseline plus one-fourth of the x-height.

$\langle \text{Construct a superscript box } x \text{ 934} \rangle \equiv$

```
begin  $x \leftarrow \text{clean\_box}(\text{supscr}(q), \text{sup\_style}(\text{cur\_style}))$ ;  $\text{width}(x) \leftarrow \text{width}(x) + \text{script\_space}$ ;
if  $\text{odd}(\text{cur\_style})$  then  $\text{clr} \leftarrow \text{sup3}(\text{cur\_size})$ 
else if  $\text{cur\_style} < \text{text\_style}$  then  $\text{clr} \leftarrow \text{sup1}(\text{cur\_size})$ 
else  $\text{clr} \leftarrow \text{sup2}(\text{cur\_size})$ ;
if  $\text{shift\_up} < \text{clr}$  then  $\text{shift\_up} \leftarrow \text{clr}$ ;
 $\text{clr} \leftarrow \text{depth}(x) + (\text{abs}(\text{math\_x\_height}(\text{cur\_size})) \text{ div } 4)$ ;
if  $\text{shift\_up} < \text{clr}$  then  $\text{shift\_up} \leftarrow \text{clr}$ ;
end
```

This code is used in section 932.

935. When both subscript and superscript are present, the subscript must be separated from the superscript by at least four times $\text{default_rule_thickness}$. If this condition would be violated, the subscript moves down, after which both subscript and superscript move up so that the bottom of the superscript is at least as high as the baseline plus four-fifths of the x-height.

$\langle \text{Construct a sub/superscript combination box } x, \text{ with the superscript offset by delta 935} \rangle \equiv$

```
begin  $y \leftarrow \text{clean\_box}(\text{subscr}(q), \text{sub\_style}(\text{cur\_style}))$ ;  $\text{width}(y) \leftarrow \text{width}(y) + \text{script\_space}$ ;
if  $\text{shift\_down} < \text{sub2}(\text{cur\_size})$  then  $\text{shift\_down} \leftarrow \text{sub2}(\text{cur\_size})$ ;
 $\text{clr} \leftarrow 4 * \text{default\_rule\_thickness} - ((\text{shift\_up} - \text{depth}(x)) - (\text{height}(y) - \text{shift\_down}))$ ;
if  $\text{clr} > 0$  then
begin  $\text{shift\_down} \leftarrow \text{shift\_down} + \text{clr}$ ;
 $\text{clr} \leftarrow (\text{abs}(\text{math\_x\_height}(\text{cur\_size})) * 4) \text{ div } 5 - (\text{shift\_up} - \text{depth}(x))$ ;
if  $\text{clr} > 0$  then
begin  $\text{shift\_up} \leftarrow \text{shift\_up} + \text{clr}$ ;  $\text{shift\_down} \leftarrow \text{shift\_down} - \text{clr}$ ;
end;
end;
 $\text{shift\_amount}(x) \leftarrow \text{delta}$ ; { superscript is delta to the right of the subscript }
 $p \leftarrow \text{new\_kern}((\text{shift\_up} - \text{depth}(x)) - (\text{height}(y) - \text{shift\_down}))$ ;  $\text{link}(x) \leftarrow p$ ;  $\text{link}(p) \leftarrow y$ ;
 $x \leftarrow \text{vpack}(x, \text{natural})$ ;  $\text{shift\_amount}(x) \leftarrow \text{shift\_down}$ ;
end
```

This code is used in section 932.

936. 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.

$\langle \text{Make a second pass over the mlist, removing all noads and inserting the proper spacing and penalties 936} \rangle \equiv$

```
p  $\leftarrow \text{temp\_head}$ ;  $\text{link}(p) \leftarrow \text{null}$ ; q  $\leftarrow \text{mlist}$ ; r_type  $\leftarrow 0$ ; cur_style  $\leftarrow \text{style}$ ;
{ Set up the values of cur_size and cur_mu, based on cur_style 879 };
while  $q \neq \text{null}$  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 937 };
{ Append inter-element spacing based on r_type and t 942 };
{ Append any new_hlist entries for q, and any appropriate penalties 943 };
if type(q) = right_noad then t  $\leftarrow \text{open\_noad}$ ;
r_type  $\leftarrow t$ ;
delete_q: r  $\leftarrow q$ ; q  $\leftarrow \text{link}(q)$ ; free_node(r, s);
done: end
```

This code is used in section 902.

937. Just before doing the big **case** switch in the second pass, the program sets up default values so that most of the branches are short.

```

⟨ 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 937 ⟩ ≡
t ← ord_noad; s ← noad_size; pen ← inf_penalty;
case type(q) of
op_noad, open_noad, close_noad, punct_noad, inner_noad: t ← type(q);
bin_noad: begin t ← bin_noad; pen ← bin_op_penalty;
end;
rel_noad: begin t ← rel_noad; pen ← rel_penalty;
end;
ord_noad, vcenter_noad, over_noad, under_noad: do_nothing;
radical_noad: s ← radical_noad_size;
accent_noad: s ← accent_noad_size;
fraction_noad: s ← fraction_noad_size;
left_noad, right_noad: t ← make_left_right(q, style, max_d, max_h);
style_node: ⟨ Change the current style and goto delete_q 939 ⟩;
whatsit_node, penalty_node, rule_node, disc_node, adjust_node, ins_node, mark_node, glue_node, kern_node:
begin link(p) ← q; p ← q; q ← link(q); link(p) ← null; goto done;
end;
othercases confusion("mlist3")
endcases
```

This code is used in section 936.

938. 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 910 ⟩ +≡
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 879 };
        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;
```

939. ⟨ Change the current style and **goto** *delete_q* 939 ⟩ ≡

```

begin cur_style ← subtype(q); s ← style_node_size;
{ Set up the values of cur_size and cur_mu, based on cur_style 879 };
goto delete_q;
end
```

This code is used in section 937.

940. The inter-element spacing in math formulas depends on an 8×8 table that TeX preloads as a 64-digit string. The elements of this string have the following significance:

- 0 means no space;
- 1 means a conditional thin space (`\nonscript\mskip\thinmuskip`);
- 2 means a thin space (`\mskip\thinmuskip`);
- 3 means a conditional medium space (`\nonscript\mskip\medmuskip`);
- 4 means a conditional thick space (`\nonscript\mskip\thickmuskip`);
- * means an impossible case.

This is all pretty cryptic, but *The TeXbook* explains what is supposed to happen, and the string makes it happen.

A global variable `magic_offset` is computed so that if a and b are in the range `ord_noad .. inner_noad`, then `str_pool[a * 8 + b + magic_offset]` is the digit for spacing between noad types a and b .

If Pascal had provided a good way to preload constant arrays, this part of the program would not have been so strange.

```
define math_spacing =
"0234000122*4000133**3**344*0400400*000000234000111*1111112341011"
⟨ Global variables 13 ⟩ +≡
magic_offset: integer; { used to find inter-element spacing }
```

941. ⟨ Compute the magic offset 941 ⟩ ≡
 $magic_offset \leftarrow str_start[math_spacing] - 9 * ord_noad$

This code is used in section 1517.

942. ⟨ Append inter-element spacing based on r_type and t 942 ⟩ ≡

```
if r_type > 0 then { not the first noad }
begin case so(str_pool[r_type * 8 + t + magic_offset]) of
"0": x ← 0;
"1": if cur_style < script_style then x ← thin_mu_skip_code else x ← 0;
"2": x ← thin_mu_skip_code;
"3": if cur_style < script_style then x ← med_mu_skip_code else x ← 0;
"4": if cur_style < script_style then x ← thick_mu_skip_code else x ← 0;
othercases confusion("mlist4")
endcases;
if x ≠ 0 then
begin y ← math_glue(glue_par(x), cur_mu); z ← new_glue(y); glue_ref_count(y) ← null;
link(p) ← z; p ← z;
subtype(z) ← x + 1; { store a symbolic subtype }
end;
end
```

This code is used in section 936.

943. We insert a penalty node after the hlist entries of noad q if pen is not an “infinite” penalty, and if the node immediately following q is not a penalty node or a rel_noad or absent entirely.

\langle Append any new_hlist entries for q , and any appropriate penalties 943 $\rangle \equiv$

```
if  $new\_hlist(q) \neq null$  then
  begin  $link(p) \leftarrow new\_hlist(q)$ ;
  repeat  $p \leftarrow link(p)$ ;
  until  $link(p) = null$ ;
  end;
if  $penalties$  then
  if  $link(q) \neq null$  then
    if  $pen < inf\_penalty$  then
      begin  $r\_type \leftarrow type(link(q))$ ;
      if  $r\_type \neq penalty\_node$  then
        if  $r\_type \neq rel\_noad$  then
          begin  $z \leftarrow new\_penalty(pen)$ ;  $link(p) \leftarrow z$ ;  $p \leftarrow z$ ;
          end;
    end
```

This code is used in section 936.

944. Alignment. It's sort of a miracle whenever `\halign` and `\valign` work, because they cut across so many of the control structures of TeX.

Therefore the present page is probably not the best place for a beginner to start reading this program; it is better to master everything else first.

Let us focus our thoughts on an example of what the input might be, in order to get some idea about how the alignment miracle happens. The example doesn't do anything useful, but it is sufficiently general to indicate all of the special cases that must be dealt with; please do not be disturbed by its apparent complexity and meaninglessness.

```
\tabskip 2pt plus 3pt
\halign to 300pt{u1#v1&
    \tabskip 1pt plus 1fil u2#v2&
    u3#v3\cr
    a1&\omit a2&\vrule\cr
    \noalign{\vskip 3pt}
    b1\span b2\cr
    \omit&c2\span\omit\cr}
```

Here's what happens:

(0) When '`\halign to 300pt{`' is scanned, the `scan_spec` routine places the 300pt dimension onto the `save_stack`, and an `align_group` code is placed above it. This will make it possible to complete the alignment when the matching '`}`' is found.

(1) The preamble is scanned next. Macros in the preamble are not expanded, except as part of a tabskip specification. For example, if `u2` had been a macro in the preamble above, it would have been expanded, since TeX must look for '`minus...`' as part of the tabskip glue. A "preamble list" is constructed based on the user's preamble; in our case it contains the following seven items:

<code>\glue 2pt plus 3pt</code>	(the tabskip preceding column 1)
<code>\alignrecord, width -∞</code>	(preamble info for column 1)
<code>\glue 2pt plus 3pt</code>	(the tabskip between columns 1 and 2)
<code>\alignrecord, width -∞</code>	(preamble info for column 2)
<code>\glue 1pt plus 1fil</code>	(the tabskip between columns 2 and 3)
<code>\alignrecord, width -∞</code>	(preamble info for column 3)
<code>\glue 1pt plus 1fil</code>	(the tabskip following column 3)

These "alignrecord" entries have the same size as an `unset_node`, since they will later be converted into such nodes. However, at the moment they have no `type` or `subtype` fields; they have `info` fields instead, and these `info` fields are initially set to the value `end_span`, for reasons explained below. Furthermore, the alignrecord nodes have no `height` or `depth` fields; these are renamed `u_part` and `v_part`, and they point to token lists for the templates of the alignment. For example, the `u_part` field in the first alignrecord points to the token list '`u1`', i.e., the template preceding the '#' for column 1.

(2) TeX now looks at what follows the `\cr` that ended the preamble. It is not '`\noalign`' or '`\omit`', so this input is put back to be read again, and the template '`u1`' is fed to the scanner. Just before reading '`u1`', TeX goes into restricted horizontal mode. Just after reading '`u1`', TeX will see '`a1`', and then (when the `&` is sensed) TeX will see '`v1`'. Then TeX scans an `endv` token, indicating the end of a column. At this point an `unset_node` is created, containing the contents of the current hlist (i.e., '`u1a1v1`'). The natural width of this unset node replaces the `width` field of the alignrecord for column 1; in general, the alignrecords will record the maximum natural width that has occurred so far in a given column.

(3) Since '`\omit`' follows the `&`, the templates for column 2 are now bypassed. Again TeX goes into restricted horizontal mode and makes an `unset_node` from the resulting hlist; but this time the hlist contains simply '`a2`'. The natural width of the new unset box is remembered in the `width` field of the alignrecord for column 2.

(4) A third `unset_node` is created for column 3, using essentially the mechanism that worked for column 1; this unset box contains '`u3\vrule v3`'. The vertical rule in this case has running dimensions that will later

extend to the height and depth of the whole first row, since each *unset_node* in a row will eventually inherit the height and depth of its enclosing box.

(5) The first row has now ended; it is made into a single unset box comprising the following seven items:

```
\glue 2pt plus 3pt
\unsetbox for 1 column: u1a1v1
\glue 2pt plus 3pt
\unsetbox for 1 column: a2
\glue 1pt plus 1fil
\unsetbox for 1 column: u3\vrule v3
\glue 1pt plus 1fil
```

The width of this unset row is unimportant, but it has the correct height and depth, so the correct baselineskip glue will be computed as the row is inserted into a vertical list.

(6) Since ‘\noalign’ follows the current \cr, TeX appends additional material (in this case \vskip 3pt) to the vertical list. While processing this material, TeX will be in internal vertical mode, and *no_align_group* will be on *save_stack*.

(7) The next row produces an unset box that looks like this:

```
\glue 2pt plus 3pt
\unsetbox for 2 columns: u1b1v1u2b2v2
\glue 1pt plus 1fil
\unsetbox for 1 column: (empty)
\glue 1pt plus 1fil
```

The natural width of the unset box that spans columns 1 and 2 is stored in a “span node,” which we will explain later; the *info* field of the alignrecord for column 1 now points to the new span node, and the *info* of the span node points to *end_span*.

(8) The final row produces the unset box

```
\glue 2pt plus 3pt
\unsetbox for 1 column: (empty)
\glue 2pt plus 3pt
\unsetbox for 2 columns: u2c2v2
\glue 1pt plus 1fil
```

A new span node is attached to the alignrecord for column 2.

(9) The last step is to compute the true column widths and to change all the unset boxes to hboxes, appending the whole works to the vertical list that encloses the \halign. The rules for deciding on the final widths of each unset column box will be explained below.

Note that as \halign is being processed, we fearlessly give up control to the rest of TeX. At critical junctures, an alignment routine is called upon to step in and do some little action, but most of the time these routines just lurk in the background. It’s something like post-hypnotic suggestion.

945. We have mentioned that alignrecords contain no *height* or *depth* fields. Their *glue_sign* and *glue_order* are pre-empted as well, since it is necessary to store information about what to do when a template ends. This information is called the *extra_info* field.

```
define u_part(#{) ≡ mem[# + height_offset].int { pointer to  $\langle u_j \rangle$  token list }
define v_part(#{) ≡ mem[# + depth_offset].int { pointer to  $\langle v_j \rangle$  token list }
define extra_info(#{) ≡ info(# + list_offset) { info to remember during template }
```

946. Alignments can occur within alignments, so a small stack is used to access the alignrecord information. At each level we have a *preamble* pointer, indicating the beginning of the preamble list; a *cur_align* pointer, indicating the current position in the preamble list; a *cur_span* pointer, indicating the value of *cur_align* at the beginning of a sequence of spanned columns; a *cur_loop* pointer, indicating the tabskip glue before an alignrecord that should be copied next if the current list is extended; and the *align_state* variable, which indicates the nesting of braces so that \cr and \span and tab marks are properly intercepted. There also are pointers *cur_head* and *cur_tail* to the head and tail of a list of adjustments being moved out from horizontal mode to vertical mode.

The current values of these seven quantities appear in global variables; when they have to be pushed down, they are stored in 5-word nodes, and *align_ptr* points to the topmost such node.

```
define preamble ≡ link(align_head) { the current preamble list }
define align_stack_node_size = 6 { number of mem words to save alignment states }

⟨Global variables 13⟩ +≡
cur_align: pointer; { current position in preamble list }
cur_span: pointer; { start of currently spanned columns in preamble list }
cur_loop: pointer; { place to copy when extending a periodic preamble }
align_ptr: pointer; { most recently pushed-down alignment stack node }
cur_head, cur_tail: pointer; { adjustment list pointers }
cur_pre_head, cur_pre_tail: pointer; { pre-adjustment list pointers }
```

947. The *align_state* and *preamble* variables are initialized elsewhere.

```
⟨ Set initial values of key variables 21 ⟩ +≡
align_ptr ← null; cur_align ← null; cur_span ← null; cur_loop ← null; cur_head ← null;
cur_tail ← null; cur_pre_head ← null; cur_pre_tail ← null;
```

948. Alignment stack maintenance is handled by a pair of trivial routines called *push_alignment* and *pop_alignment*.

```
procedure push_alignment;
  var p: pointer; { the new alignment stack node }
  begin p ← get_node(align_stack_node_size); link(p) ← align_ptr; info(p) ← cur_align;
    llink(p) ← preamble; rlink(p) ← cur_span; mem[p + 2].int ← cur_loop; mem[p + 3].int ← align_state;
    info(p + 4) ← cur_head; link(p + 4) ← cur_tail; info(p + 5) ← cur_pre_head; link(p + 5) ← cur_pre_tail;
    align_ptr ← p; cur_head ← get_avail; cur_pre_head ← get_avail;
  end;

procedure pop_alignment;
  var p: pointer; { the top alignment stack node }
  begin free_avail(cur_head); free_avail(cur_pre_head); p ← align_ptr; cur.tail ← link(p + 4);
    cur_head ← info(p + 4); cur_pre.tail ← link(p + 5); cur_pre_head ← info(p + 5);
    align_state ← mem[p + 3].int; cur_loop ← mem[p + 2].int; cur_span ← rlink(p); preamble ← llink(p);
    cur_align ← info(p); align_ptr ← link(p); free_node(p, align_stack_node_size);
  end;
```

949. TeX has eight procedures that govern alignments: *init_align* and *fin_align* are used at the very beginning and the very end; *init_row* and *fin_row* are used at the beginning and end of individual rows; *init_span* is used at the beginning of a sequence of spanned columns (possibly involving only one column); *init_col* and *fin_col* are used at the beginning and end of individual columns; and *align_peek* is used after \cr to see whether the next item is \noalign.

We shall consider these routines in the order they are first used during the course of a complete \halign, namely *init_align*, *align_peek*, *init_row*, *init_span*, *init_col*, *fin_col*, *fin_row*, *fin_align*.

950. When `\halign` or `\valign` has been scanned in an appropriate mode, TeX calls `init_align`, whose task is to get everything off to a good start. This mostly involves scanning the preamble and putting its information into the preamble list.

```
< Declare the procedure called get_preamble_token 958 >
procedure align_peek; forward;
procedure normal_paragraph; forward;
procedure init_align;
label done, done1, done2, continue;
var save_cs_ptr: pointer; { warning_index value for error messages }
p: pointer; { for short-term temporary use }
begin save_cs_ptr ← cur_cs; { \halign or \valign, usually }
push_alignment; align_state ← -1000000; { enter a new alignment level }
{ Check for improper alignment in displayed math 952 };
push_nest; { enter a new semantic level }
{ Change current mode to -vmode for \halign, -hmode for \valign 951 };
scan_spec(align_group, false);
{ Scan the preamble and record it in the preamble list 953 };
new_save_level(align_group);
if every_cr ≠ null then begin_token_list(every_cr, every_cr_text);
align_peek; { look for \noalign or \omit }
end;
```

951. In vertical modes, `prev_depth` already has the correct value. But if we are in `mmode` (displayed formula mode), we reach out to the enclosing vertical mode for the `prev_depth` value that produces the correct baseline calculations.

```
< Change current mode to -vmode for \halign, -hmode for \valign 951 > ≡
if mode = mmode then
begin mode ← -vmode; prev_depth ← nest[nest_ptr - 2].aux_field.sc;
end
else if mode > 0 then negate(mode)
```

This code is used in section 950.

952. When `\halign` is used as a displayed formula, there should be no other pieces of mlists present.

```
< Check for improper alignment in displayed math 952 > ≡
if (mode = mmode) ∧ ((tail ≠ head) ∨ (incompleat_noad ≠ null)) then
begin print_err("Improper "); print_esc("halign"); print(" inside $$ `s");
help3("Displays can use special alignments (like \\eqalignno)");
("only if nothing but the alignment itself is between $$ `s .")
("So I've deleted the formulas that preceded this alignment."); error; flush_math;
end
```

This code is used in section 950.

953. \langle Scan the preamble and record it in the *preamble* list 953 $\rangle \equiv$
 $\text{preamble} \leftarrow \text{null}; \text{cur_align} \leftarrow \text{align_head}; \text{cur_loop} \leftarrow \text{null}; \text{scanner_status} \leftarrow \text{aligning};$
 $\text{warning_index} \leftarrow \text{save_cs_ptr}; \text{align_state} \leftarrow -1000000; \{ \text{at this point, } \text{cur_cmd} = \text{left_brace} \}$
loop begin \langle Append the current tabskip glue to the preamble list 954 $\rangle;$
if $\text{cur_cmd} = \text{car_ret}$ **then goto** done; $\{ \backslash\text{cr} \text{ ends the preamble} \}$
 \langle Scan preamble text until cur_cmd is *tab_mark* or *car_ret*, looking for changes in the tabskip glue;
append an alignrecord to the preamble list 955 $\rangle;$
end;
done: $\text{scanner_status} \leftarrow \text{normal}$

This code is used in section 950.

954. \langle Append the current tabskip glue to the preamble list 954 $\rangle \equiv$
 $\text{link}(\text{cur_align}) \leftarrow \text{new_param_glue}(\text{tab_skip_code}); \text{cur_align} \leftarrow \text{link}(\text{cur_align})$

This code is used in section 953.

955. \langle Scan preamble text until cur_cmd is *tab_mark* or *car_ret*, looking for changes in the tabskip glue;
append an alignrecord to the preamble list 955 $\rangle \equiv$
 \langle Scan the template $\langle u_j \rangle$, putting the resulting token list in *hold_head* 959 $\rangle;$
 $\text{link}(\text{cur_align}) \leftarrow \text{new_null_box}; \text{cur_align} \leftarrow \text{link}(\text{cur_align}); \{ \text{a new alignrecord} \}$
 $\text{info}(\text{cur_align}) \leftarrow \text{end_span}; \text{width}(\text{cur_align}) \leftarrow \text{null_flag}; \text{u_part}(\text{cur_align}) \leftarrow \text{link}(\text{hold_head});$
 \langle Scan the template $\langle v_j \rangle$, putting the resulting token list in *hold_head* 960 $\rangle;$
 $\text{v_part}(\text{cur_align}) \leftarrow \text{link}(\text{hold_head})$

This code is used in section 953.

956. We enter ‘\span’ into *eqtb* with *tab_mark* as its command code, and with *span_code* as the command modifier. This makes TeX interpret it essentially the same as an alignment delimiter like ‘&’, yet it is recognizably different when we need to distinguish it from a normal delimiter. It also turns out to be useful to give a special *cr_code* to ‘\cr’, and an even larger *cr_cr_code* to ‘\crr’.

The end of a template is represented by two “frozen” control sequences called *\endtemplate*. The first has the command code *end_template*, which is > *outer_call*, so it will not easily disappear in the presence of errors. The *get_x_token* routine converts the first into the second, which has *endv* as its command code.

```
define span_code = 256 { distinct from any character }
define cr_code = 257 { distinct from span_code and from any character }
define cr_cr_code = cr_code + 1 { this distinguishes \crr from \cr }
define end_template_token ≡ cs_token_flag + frozen_end_template

⟨ Put each of TeX’s primitives into the hash table 244 ⟩ +≡
primitive("span", tab_mark, span_code);
primitive("cr", car_ret, cr_code); text(frozen_cr) ← "cr"; eqtb[frozen_cr] ← eqtb[cur_val];
primitive("crr", car_ret, cr_cr_code); text(frozen_end_template) ← "endtemplate";
text(frozen_endv) ← "endtemplate"; eq_type(frozen_endv) ← endv; equiv(frozen_endv) ← null_list;
eq_level(frozen_endv) ← level_one;
eqtb[frozen_end_template] ← eqtb[frozen_endv]; eq_type(frozen_end_template) ← end_template;
```

957. \langle Cases of *print_cmd_chr* for symbolic printing of primitives 245 $\rangle +≡$

```
tab_mark: if chr_code = span_code then print_esc("span")
else chr_cmd("alignment\!tab\!character\!");
car_ret: if chr_code = cr_code then print_esc("cr")
else print_esc("crr");
```

958. The preamble is copied directly, except that `\tabskip` causes a change to the `\tabskip` glue, thereby possibly expanding macros that immediately follow it. An appearance of `\span` also causes such an expansion.

Note that if the preamble contains '`\global\tabskip`', the '`\global`' token survives in the preamble and the '`\tabskip`' defines new `\tabskip` glue (locally).

(Declare the procedure called `get_preamble_token` 958) ≡

```
procedure get_preamble_token;
label restart;
begin restart: get_token;
while (cur_chr = span_code) ∧ (cur_cmd = tab_mark) do
  begin get_token; { this token will be expanded once }
  if cur_cmd > max_command then
    begin expand; get_token;
    end;
  end;
if cur_cmd = endv then fatal_error("(interwoven_alignment_preambles_are_not_allowed)");
if (cur_cmd = assign_glue) ∧ (cur_chr = glue_base + tab_skip_code) then
  begin scan_optional_equals; scan_glue(glue_val);
  if global_defs > 0 then geq_define(glue_base + tab_skip_code, glue_ref, cur_val)
  else eq_define(glue_base + tab_skip_code, glue_ref, cur_val);
  goto restart;
  end;
end;
```

This code is used in section 950.

959. Spaces are eliminated from the beginning of a template.

(Scan the template $\langle u_j \rangle$, putting the resulting token list in `hold_head` 959) ≡

```
p ← hold_head; link(p) ← null;
loop begin get_preamble_token;
  if cur_cmd = mac_param then goto done1;
  if (cur_cmd ≤ car_ret) ∧ (cur_cmd ≥ tab_mark) ∧ (align_state = -1000000) then
    if (p = hold_head) ∧ (cur_loop = null) ∧ (cur_cmd = tab_mark) then cur_loop ← cur_align
    else begin print_err("Missing # inserted in alignment preamble");
    help3("There should be exactly one # between & 's, when an")
    ("\\halign or \\valign is being set up. In this case you had")
    ("none, so I've put one in; maybe that will work."); back_error; goto done1;
    end
  else if (cur_cmd ≠ spacer) ∨ (p ≠ hold_head) then
    begin link(p) ← get_avail; p ← link(p); info(p) ← cur_tok;
    end;
  end;
done1:
```

This code is used in section 955.

```

960.  { Scan the template  $\langle v_j \rangle$ , putting the resulting token list in hold_head 960 } ≡
   $p \leftarrow hold\_head$ ;  $link(p) \leftarrow null$ ;
  loop begin continue: get_preamble_token;
    if (cur_cmd ≤ car_ret) ∧ (cur_cmd ≥ tab_mark) ∧ (align_state = -1000000) then goto done2;
    if cur_cmd = mac_param then
      begin print_err("Only one # is allowed per tab");
      help3("There should be exactly one # between & 's, when an")
      ("\\halign or \\valign is being set up. In this case you had")
      ("more than one, so I'm ignoring all but the first."); error; goto continue;
      end;
    link(p) ← get_avail;  $p \leftarrow link(p)$ ; info(p) ← cur_tok;
    end;
done2: link(p) ← get_avail;  $p \leftarrow link(p)$ ; info(p) ← end_template_token { put \endtemplate at the end }
```

This code is used in section 955.

961. 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 961 ⟩ ≡
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 976.

962. To start a row (i.e., a ‘row’ that rhymes with ‘dough’ but not with ‘bough’), we enter a new semantic level, copy the first tabskip glue, and change from internal vertical mode to restricted horizontal mode or vice versa. The *space_factor* and *prev_depth* are not used on this semantic level, but we clear them to zero just to be tidy.

```

⟨ Declare the procedure called init_span 963 ⟩
procedure init_row;
  begin push_nest; mode ← (-hmode - vmode) - mode;
  if mode = -hmode then space_factor ← 0 else prev_depth ← 0;
  tail_append(new_glue(glue_ptr(preamble))); subtype(tail) ← tab_skip_code + 1;
  cur_align ← link(preamble); cur_tail ← cur_head; cur_pre_tail ← cur_pre_head; init_span(cur_align);
  end;
```

963. The parameter to *init_span* is a pointer to the alignrecord where the next column or group of columns will begin. A new semantic level is entered, so that the columns will generate a list for subsequent packaging.

⟨ Declare the procedure called *init_span* 963 ⟩ ≡

```
procedure init_span(p : pointer);
begin push_nest;
if mode = -hmode then space_factor ← 1000
else begin prev_depth ← pdf_ignored_dimen; normal_paragraph;
end;
cur_span ← p;
end;
```

This code is used in section 962.

964. When a column begins, we assume that *cur_cmd* is either *omit* or else the current token should be put back into the input until the $\langle u_j \rangle$ template has been scanned. (Note that *cur_cmd* might be *tab_mark* or *car_ret*.) We also assume that *align_state* is approximately 1000000 at this time. We remain in the same mode, and start the template if it is called for.

```
procedure init_col;
begin extra_info(cur_align) ← cur_cmd;
if cur_cmd = omit then align_state ← 0
else begin back_input; begin_token_list(u_part(cur_align), u_template);
end; { now align_state = 1000000 }
end;
```

965. The scanner sets *align_state* to zero when the $\langle u_j \rangle$ template ends. When a subsequent \cr or \span or tab mark occurs with *align_state* = 0, the scanner activates the following code, which fires up the $\langle v_j \rangle$ template. We need to remember the *cur_chr*, which is either *cr_cr_code*, *cr_code*, *span_code*, or a character code, depending on how the column text has ended.

This part of the program had better not be activated when the preamble to another alignment is being scanned, or when no alignment preamble is active.

⟨ Insert the $\langle v_j \rangle$ template and goto *restart* 965 ⟩ ≡

```
begin if (scanner_status = aligning) ∨ (cur_align = null) then
fatal_error("(interwoven_alignment_preambles_are_not_allowed)");
cur_cmd ← extra_info(cur_align); extra_info(cur_align) ← cur_chr;
if cur_cmd = omit then begin_token_list(omit_template, v_template)
else begin_token_list(v_part(cur_align), v_template);
align_state ← 1000000; goto restart;
end
```

This code is used in section 364.

966. The token list *omit_template* just referred to is a constant token list that contains the special control sequence \endtemplate only.

⟨ Initialize the special list heads and constant nodes 966 ⟩ ≡

```
info(omit_template) ← end_template_token; { link(omit_template) = null }
```

See also sections 973, 996, 1158, and 1165.

This code is used in section 182.

967. 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 968 };
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 972 };
  { Copy the tabskip glue between columns 971 };
  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;

```

968. { If the preamble list has been traversed, check that the row has ended 968 } ≡

```

if (p = null) ∧ (extra_info(cur_align) < cr_code) then
  if cur_loop ≠ null then { Lengthen the preamble periodically 969 }
  else begin print_err("Extra_alignment_tab_has_been_changed_to_"); print_esc("cr");
  help3("You_have_given_more_\span_or_&_marks_than_there_were")
  ("in_the_preamble_to_the_\halign_or_\valign_now_in_progress.")
  ("So_I'll_assume_that_you_meant_to_type_\cr_instead."); extra_info(cur_align) ← cr_code;
  error;
end

```

This code is used in section 967.

969. { Lengthen the preamble periodically 969 } ≡

```

begin link(q) ← new_null_box; p ← link(q); { a new alignrecord }
info(p) ← end_span; width(p) ← null_flag; cur_loop ← link(cur_loop);
{ Copy the templates from node cur_loop into node p 970 };
cur_loop ← link(cur_loop); link(p) ← new_glue(glue_ptr(cur_loop)); subtype(link(p)) ← tab_skip_code + 1;
end

```

This code is used in section 968.

970. *< Copy the templates from node `cur_loop` into node `p` 970 >* ≡
`q ← hold_head; r ← u_part(cur_loop);`
`while r ≠ null do`
`begin link(q) ← get_avail; q ← link(q); info(q) ← info(r); r ← link(r);`
`end;`
`link(q) ← null; u_part(p) ← link(hold_head); q ← hold_head; r ← v_part(cur_loop);`
`while r ≠ null do`
`begin link(q) ← get_avail; q ← link(q); info(q) ← info(r); r ← link(r);`
`end;`
`link(q) ← null; v_part(p) ← link(hold_head)`

This code is used in section 969.

971. *< Copy the tabskip glue between columns 971 >* ≡
`tail_append(new_glue(glue_ptr(link(cur_align)))); subtype(tail) ← tab_skip_code + 1`

This code is used in section 967.

972. *< Package an unset box for the current column and record its width 972 >* ≡
`begin if mode = -hmode then`
`begin adjust_tail ← cur_tail; pre_adjust_tail ← cur_pre_tail; u ← hpack(link(head), natural);`
`w ← width(u); cur_tail ← adjust_tail; adjust_tail ← null; cur_pre_tail ← pre_adjust_tail;`
`pre_adjust_tail ← null;`
`end`
`else begin u ← vpackage(link(head), natural, 0); w ← height(u);`
`end;`
`n ← min_quarterword; { this represents a span count of 1 }`
`if cur_span ≠ cur_align then < Update width entry for spanned columns 974 >`
`else if w > width(cur_align) then width(cur_align) ← w;`
`type(u) ← unset_node; span_count(u) ← n;`
`{ Determine the stretch order 835};`
`glue_order(u) ← o; glue_stretch(u) ← total_stretch[o];`
`{ Determine the shrink order 841};`
`glue_sign(u) ← o; glue_shrink(u) ← total_shrink[o];`
`pop_nest; link(tail) ← u; tail ← u;`
`end`

This code is used in section 967.

973. A span node is a 2-word record containing *width*, *info*, and *link* fields. The *link* field is not really a link, it indicates the number of spanned columns; the *info* field points to a span node for the same starting column, having a greater extent of spanning, or to *end_span*, which has the largest possible *link* field; the *width* field holds the largest natural width corresponding to a particular set of spanned columns.

A list of the maximum widths so far, for spanned columns starting at a given column, begins with the *info* field of the alignrecord for that column.

```
define span_node_size = 2 { number of mem words for a span node }
< Initialize the special list heads and constant nodes 966 > +≡
link(end_span) ← max_quarterword + 1; info(end_span) ← null;
```

974. \langle Update width entry for spanned columns 974 $\rangle \equiv$

```

begin  $q \leftarrow cur\_span$ ;
repeat  $incr(n); q \leftarrow link(link(q))$ ;
until  $q = cur\_align$ ;
if  $n > max\_quarterword$  then  $confusion("256\_spans")$ ; { this can happen, but won't }
 $q \leftarrow cur\_span$ ;
while  $link(info(q)) < n$  do  $q \leftarrow info(q)$ ;
if  $link(info(q)) > n$  then
  begin  $s \leftarrow get\_node(span\_node\_size)$ ;  $info(s) \leftarrow info(q)$ ;  $link(s) \leftarrow n$ ;  $info(q) \leftarrow s$ ;  $width(s) \leftarrow w$ ;
  end
else if  $width(info(q)) < w$  then  $width(info(q)) \leftarrow w$ ;
end

```

This code is used in section 972.

975. At the end of a row, we append an unset box to the current vlist (for `\halign`) or the current hlist (for `\valign`). This unset box contains the unset boxes for the columns, separated by the tabskip glue. Everything will be set later.

```

procedure fin_row;
var  $p$ : pointer; { the new unset box }
begin if mode = -hmode then
  begin  $p \leftarrow hpack(link(head), natural)$ ; pop_nest;
  if  $cur\_pre\_head \neq cur\_pre\_tail$  then append_list( $cur\_pre\_head$ )( $cur\_pre\_tail$ );
  append_to_vlist( $p$ );
  if  $cur\_head \neq cur\_tail$  then append_list( $cur\_head$ )( $cur\_tail$ );
  end
else begin  $p \leftarrow vpack(link(head), natural)$ ; pop_nest;  $link(tail) \leftarrow p$ ;  $tail \leftarrow p$ ; space_factor  $\leftarrow 1000$ ;
  end;
type( $p$ )  $\leftarrow$  unset_node; glue_stretch( $p$ )  $\leftarrow 0$ ;
if every_cr  $\neq$  null then begin_token_list(every_cr, every_cr_text);
align_peek;
end; { note that  $glue\_shrink(p) = 0$  since  $glue\_shrink \equiv shift\_amount$  }

```

976. Finally, we will reach the end of the alignment, and we can breathe a sigh of relief that memory hasn't overflowed. All the unset boxes will now be set so that the columns line up, taking due account of spanned columns.

```

procedure do_assignments; forward;
procedure resume_after_display; forward;
procedure build_page; forward;
procedure fin_align;
var p,q,r,s,u,v: pointer; { registers for the list operations }
t,w: scaled; { width of column }
o: scaled; { shift offset for unset boxes }
n: halfword; { matching span amount }
rule_save: scaled; { temporary storage for overfull_rule }
aux_save: memory_word; { temporary storage for aux }
begin if cur_group ≠ align_group then confusion("align1");
unsafe; { that align_group was for individual entries }
if cur_group ≠ align_group then confusion("align0");
unsafe; { that align_group was for the whole alignment }
if nest[nest_ptr - 1].mode_field = mmode then o ← display_indent
else o ← 0;
{ Go through the preamble list, determining the column widths and changing the alignrecords to dummy
unset boxes 977 };
{ Package the preamble list, to determine the actual tabskip glue amounts, and let p point to this
prototype box 980 };
{ Set the glue in all the unset boxes of the current list 981 };
flush_node_list(p); pop_alignment; { Insert the current list into its environment 988 };
end;
{ Declare the procedure called align_peek 961 }

```

977. It's time now to dismantle the preamble list and to compute the column widths. Let w_{ij} be the maximum of the natural widths of all entries that span columns i through j , inclusive. The alignrecord for column i contains w_{ii} in its *width* field, and there is also a linked list of the nonzero w_{ij} for increasing j , accessible via the *info* field; these span nodes contain the value $j - i + \min_quarterword$ in their *link* fields. The values of w_{ii} were initialized to *null_flag*, which we regard as $-\infty$.

The final column widths are defined by the formula

$$w_j = \max_{1 \leq i \leq j} \left(w_{ij} - \sum_{i \leq k < j} (t_k + w_k) \right),$$

where t_k is the natural width of the tabskip glue between columns k and $k + 1$. However, if $w_{ij} = -\infty$ for all i in the range $1 \leq i \leq j$ (i.e., if every entry that involved column j also involved column $j + 1$), we let $w_j = 0$, and we zero out the tabskip glue after column j .

TeX computes these values by using the following scheme: First $w_1 = w_{11}$. Then replace w_{2j} by $\max(w_{2j}, w_{1j} - t_1 - w_1)$, for all $j > 1$. Then $w_2 = w_{22}$. Then replace w_{3j} by $\max(w_{3j}, w_{2j} - t_2 - w_2)$ for all $j > 2$; and so on. If any w_j turns out to be $-\infty$, its value is changed to zero and so is the next tabskip.

```
<Go through the preamble list, determining the column widths and changing the alignrecords to dummy
unset boxes 977> ≡
q ← link(preamble);
repeat flush_list(u_part(q)); flush_list(v_part(q)); p ← link(link(q));
  if width(q) = null_flag then <Nullify width(q) and the tabskip glue following this column 978>;
    if info(q) ≠ end_span then
      <Merge the widths in the span nodes of q with those of p, destroying the span nodes of q 979>;
      type(q) ← unset_node; span_count(q) ← min_quarterword; height(q) ← 0; depth(q) ← 0;
      glue_order(q) ← normal; glue_sign(q) ← normal; glue_stretch(q) ← 0; glue_shrink(q) ← 0; q ← p;
    until q = null
```

This code is used in section 976.

978. <Nullify width(q) and the tabskip glue following this column 978> ≡

```
begin width(q) ← 0; r ← link(q); s ← glue_ptr(r);
if s ≠ zero_glue then
  begin add_glue_ref(zero_glue); delete_glue_ref(s); glue_ptr(r) ← zero_glue;
  end;
end
```

This code is used in section 977.

979. Merging of two span-node lists is a typical exercise in the manipulation of linearly linked data structures. The essential invariant in the following **repeat** loop is that we want to dispense with node r , in q 's list, and u is its successor; all nodes of p 's list up to and including s have been processed, and the successor of s matches r or precedes r or follows r , according as $\text{link}(r) = n$ or $\text{link}(r) > n$ or $\text{link}(r) < n$.

```
< Merge the widths in the span nodes of  $q$  with those of  $p$ , destroying the span nodes of  $q$  979 > ≡
begin  $t \leftarrow \text{width}(q) + \text{width}(\text{glue_ptr}(\text{link}(q)))$ ;  $r \leftarrow \text{info}(q)$ ;  $s \leftarrow \text{end\_span}$ ;  $\text{info}(s) \leftarrow p$ ;
 $n \leftarrow \text{min\_quarterword} + 1$ ;
repeat  $\text{width}(r) \leftarrow \text{width}(r) - t$ ;  $u \leftarrow \text{info}(r)$ ;
while  $\text{link}(r) > n$  do
begin  $s \leftarrow \text{info}(s)$ ;  $n \leftarrow \text{link}(\text{info}(s)) + 1$ ;
end;
if  $\text{link}(r) < n$  then
begin  $\text{info}(r) \leftarrow \text{info}(s)$ ;  $\text{info}(s) \leftarrow r$ ;  $\text{decr}(\text{link}(r))$ ;  $s \leftarrow r$ ;
end
else begin if  $\text{width}(r) > \text{width}(\text{info}(s))$  then  $\text{width}(\text{info}(s)) \leftarrow \text{width}(r)$ ;
 $\text{free\_node}(r, \text{span\_node\_size})$ ;
end;
 $r \leftarrow u$ ;
until  $r = \text{end\_span}$ ;
end
```

This code is used in section 977.

980. Now the preamble list has been converted to a list of alternating unset boxes and tabskip glue, where the box widths are equal to the final column sizes. In case of `\valign`, we change the widths to heights, so that a correct error message will be produced if the alignment is overfull or underfull.

```
< Package the preamble list, to determine the actual tabskip glue amounts, and let  $p$  point to this prototype
box 980 > ≡
 $\text{save\_ptr} \leftarrow \text{save\_ptr} - 2$ ;  $\text{pack\_begin\_line} \leftarrow -\text{mode\_line}$ ;
if  $\text{mode} = -\text{vmode}$  then
begin  $\text{rule\_save} \leftarrow \text{overfull\_rule}$ ;  $\text{overfull\_rule} \leftarrow 0$ ; { prevent rule from being packaged }
 $p \leftarrow \text{hpack}(\text{preamble}, \text{saved}(1), \text{saved}(0))$ ;  $\text{overfull\_rule} \leftarrow \text{rule\_save}$ ;
end
else begin  $q \leftarrow \text{link}(\text{preamble})$ ;
repeat  $\text{height}(q) \leftarrow \text{width}(q)$ ;  $\text{width}(q) \leftarrow 0$ ;  $q \leftarrow \text{link}(\text{link}(q))$ ;
until  $q = \text{null}$ ;
 $p \leftarrow \text{vpack}(\text{preamble}, \text{saved}(1), \text{saved}(0))$ ;  $q \leftarrow \text{link}(\text{preamble})$ ;
repeat  $\text{width}(q) \leftarrow \text{height}(q)$ ;  $\text{height}(q) \leftarrow 0$ ;  $q \leftarrow \text{link}(\text{link}(q))$ ;
until  $q = \text{null}$ ;
end;
 $\text{pack\_begin\_line} \leftarrow 0$ 
```

This code is used in section 976.

981. *⟨ Set the glue in all the unset boxes of the current list 981 ⟩* ≡
 $q \leftarrow link(head); s \leftarrow head;$
while $q \neq null$ **do**
 begin if $\neg is_char_node(q)$ **then**
 if $type(q) = unset_node$ **then** *⟨ Set the unset box q and the unset boxes in it 983 ⟩*
 else if $type(q) = rule_node$ **then**
 ⟨ Make the running dimensions in rule q extend to the boundaries of the alignment 982 ⟩;
 $s \leftarrow q; q \leftarrow link(q);$
 end

This code is used in section 976.

982. *⟨ Make the running dimensions in rule q extend to the boundaries of the alignment 982 ⟩* ≡
begin if $is_running(width(q))$ **then** $width(q) \leftarrow width(p);$
if $is_running(height(q))$ **then** $height(q) \leftarrow height(p);$
if $is_running(depth(q))$ **then** $depth(q) \leftarrow depth(p);$
if $o \neq 0$ **then**
 begin $r \leftarrow link(q); link(q) \leftarrow null; q \leftarrow hpack(q, natural); shift_amount(q) \leftarrow o; link(q) \leftarrow r;$
 $link(s) \leftarrow q;$
 end;
end

This code is used in section 981.

983. 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 983 ⟩ ≡
begin if $mode = -vmode$ **then**
 begin $type(q) \leftarrow hlist_node; width(q) \leftarrow width(p);$
 if $nest[nest_ptr - 1].mode_field = mmode$ **then** $set_box_lr(q)(dlist); \{ for ship_out \}$
 end
else begin $type(q) \leftarrow vlist_node; height(q) \leftarrow height(p);$
 end;
 $glue_order(q) \leftarrow glue_order(p); glue_sign(q) \leftarrow glue_sign(p); glue_set(q) \leftarrow glue_set(p);$
 $shift_amount(q) \leftarrow o; r \leftarrow link(list_ptr(q)); s \leftarrow link(list_ptr(p));$
repeat *⟨ Set the glue in node r and change it from an unset node 984 ⟩*;
 $r \leftarrow link(link(r)); s \leftarrow link(link(s));$
until $r = null;$
end

This code is used in section 981.

984. 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 984 > ≡
  n ← span_count(r); t ← width(s); w ← t; u ← 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 985 };
    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 986 }
  else { Make the unset node r into a vlist_node of height w, setting the glue as if the height were t 987 };
    shift_amount(r) ← 0;
  if u ≠ hold_head then { append blank boxes to account for spanned nodes }
    begin link(u) ← link(r); link(r) ← link(hold_head); r ← u;
    end
```

This code is used in section 983.

985. { Append tabskip glue and an empty box to list *u*, and update *s* and *t* as the prototype nodes are passed 985 } ≡

```

  s ← link(s); v ← glue_ptr(s); link(u) ← new_glue(v); u ← link(u); subtype(u) ← tab_skip_code + 1;
  t ← t + width(v);
  if glue_sign(p) = stretching then
    begin if stretch_order(v) = glue_order(p) then t ← t + round(float(glue_set(p)) * stretch(v));
    end
  else if glue_sign(p) = shrinking then
    begin if shrink_order(v) = glue_order(p) then t ← t - round(float(glue_set(p)) * shrink(v));
    end;
  s ← link(s); link(u) ← new_null_box; u ← link(u); t ← t + width(s);
  if mode = -vmode then width(u) ← width(s) else begin type(u) ← vlist_node; height(u) ← width(s);
  end
```

This code is used in section 984.

986. { Make the unset node *r* into an *hlist_node* of width *w*, setting the glue as if the width were *t* 986 } ≡

```

  begin height(r) ← height(q); depth(r) ← depth(q);
  if t = width(r) then
    begin glue_sign(r) ← normal; glue_order(r) ← normal; set_glue_ratio_zero(glue_set(r));
    end
  else if t > width(r) then
    begin glue_sign(r) ← stretching;
    if glue_stretch(r) = 0 then set_glue_ratio_zero(glue_set(r))
    else glue_set(r) ← unfloat((t - width(r)) / glue_stretch(r));
    end
  else begin glue_order(r) ← glue_sign(r); glue_sign(r) ← shrinking;
    if glue_shrink(r) = 0 then set_glue_ratio_zero(glue_set(r))
    else if (glue_order(r) = normal) ∧ (width(r) - t > glue_shrink(r)) then
      set_glue_ratio_one(glue_set(r))
    else glue_set(r) ← unfloat((width(r) - t) / glue_shrink(r));
    end;
  width(r) ← w; type(r) ← hlist_node;
  end
```

This code is used in section 984.

987. ⟨ Make the unset node r into a $vlist_node$ of height w , setting the glue as if the height were t 987 ⟩ ≡

```

begin width(r) ← width(q);
if t = height(r) then
  begin glue_sign(r) ← normal; glue_order(r) ← normal; set_glue_ratio_zero(glue_set(r));
  end
else if t > height(r) then
  begin glue_sign(r) ← stretching;
  if glue_stretch(r) = 0 then set_glue_ratio_zero(glue_set(r))
  else glue_set(r) ← unfloat((t - height(r))/glue_stretch(r));
  end
else begin glue_order(r) ← glue_sign(r); glue_sign(r) ← shrinking;
  if glue_shrink(r) = 0 then set_glue_ratio_zero(glue_set(r))
  else if (glue_order(r) = normal) ∧ (height(r) - t > glue_shrink(r)) then
    set_glue_ratio_one(glue_set(r))
  else glue_set(r) ← unfloat((height(r) - t)/glue_shrink(r));
  end;
end;
height(r) ← w; type(r) ← vlist_node;
end

```

This code is used in section 984.

988. We now have a completed alignment, in the list that starts at $head$ and ends at $tail$. This list will be merged with the one that encloses it. (In case the enclosing mode is $mmode$, for displayed formulas, we will need to insert glue before and after the display; that part of the program will be deferred until we're more familiar with such operations.)

In restricted horizontal mode, the $clang$ part of aux is undefined; an over-cautious Pascal runtime system may complain about this.

⟨ Insert the current list into its environment 988 ⟩ ≡

```

aux_save ← aux; p ← link(head); q ← tail; pop_nest;
if mode = mmode then ⟨ Finish an alignment in a display 1384 ⟩
else begin aux ← aux_save; link(tail) ← p;
  if p ≠ null then tail ← q;
  if mode = vmode then build_page;
end

```

This code is used in section 976.

989. Breaking paragraphs into lines. We come now to what is probably the most interesting algorithm of TeX: the mechanism for choosing the “best possible” breakpoints that yield the individual lines of a paragraph. TeX’s line-breaking algorithm takes a given horizontal list and converts it to a sequence of boxes that are appended to the current vertical list. In the course of doing this, it creates a special data structure containing three kinds of records that are not used elsewhere in TeX. Such nodes are created while a paragraph is being processed, and they are destroyed afterwards; thus, the other parts of TeX do not need to know anything about how line-breaking is done.

The method used here is based on an approach devised by Michael F. Plass and the author in 1977, subsequently generalized and improved by the same two people in 1980. A detailed discussion appears in *Software—Practice and Experience* 11 (1981), 1119–1184, where it is shown that the line-breaking problem can be regarded as a special case of the problem of computing the shortest path in an acyclic network. The cited paper includes numerous examples and describes the history of line breaking as it has been practiced by printers through the ages. The present implementation adds two new ideas to the algorithm of 1980: Memory space requirements are considerably reduced by using smaller records for inactive nodes than for active ones, and arithmetic overflow is avoided by using “delta distances” instead of keeping track of the total distance from the beginning of the paragraph to the current point.

990. 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 }

991. 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 1002)
procedure line_break(d : boolean);
  label done, done1, done2, done3, done4, done5, continue;
  var (Local variables for line breaking 1038)
  begin pack_begin_line ← mode_line; { this is for over/underfull box messages }
    (Get ready to start line breaking 992);
    (Find optimal breakpoints 1039);
    (Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and
     append them to the current vertical list 1052);
    (Clean up the memory by removing the break nodes 1041);
    pack_begin_line ← 0;
  end;
(Declare ε-TEx procedures for use by main_control 1656)
```

992. 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 992) ≡
  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_lhyf ← prev_graf div '20000000;
  init_r_hyf ← (prev_graf div '200000) mod '100; pop_nest;
```

See also sections 1003, 1010, and 1024.

This code is used in section 991.

993. When looking for optimal line breaks, TeX creates a “break node” for each break that is *feasible*, in the sense that there is a way to end a line at the given place without requiring any line to stretch more than a given tolerance. A break node is characterized by three things: the position of the break (which is a pointer to a *glue_node*, *math_node*, *penalty_node*, or *disc_node*); the ordinal number of the line that will follow this breakpoint; and the fitness classification of the line that has just ended, i.e., *tight_fit*, *decent_fit*, *loose_fit*, or *very_loose_fit*.

```
define tight_fit = 3 { fitness classification for lines shrinking 0.5 to 1.0 of their shrinkability }
define loose_fit = 1 { fitness classification for lines stretching 0.5 to 1.0 of their stretchability }
define very_loose_fit = 0 { fitness classification for lines stretching more than their stretchability }
define decent_fit = 2 { fitness classification for all other lines }
```

994. The algorithm essentially determines the best possible way to achieve each feasible combination of position, line, and fitness. Thus, it answers questions like, “What is the best way to break the opening part of the paragraph so that the fourth line is a tight line ending at such-and-such a place?” However, the fact that all lines are to be the same length after a certain point makes it possible to regard all sufficiently large line numbers as equivalent, when the looseness parameter is zero, and this makes it possible for the algorithm to save space and time.

An “active node” and a “passive node” are created in *mem* for each feasible breakpoint that needs to be considered. Active nodes are three words long and passive nodes are two words long. We need active nodes only for breakpoints near the place in the paragraph that is currently being examined, so they are recycled within a comparatively short time after they are created.

995. 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 }
```

996. ⟨ Initialize the special list heads and constant nodes 966 ⟩ +≡

```
type(last_active) ← hyphenated; line_number(last_active) ← max_halfword; subtype(last_active) ← 0;
{ the subtype is never examined by the algorithm }
```

997. The passive node for a given breakpoint contains only four fields:

link points to the passive node created just before this one, if any, otherwise it is *null*.

cur_break points to the position of this breakpoint in the horizontal list for the paragraph being broken.

prev_break points to the passive node that should precede this one in an optimal path to this breakpoint.

serial is equal to *n* if this passive node is the *n*th one created during the current pass. (This field is used only when printing out detailed statistics about the line-breaking calculations.)

There is a global variable called *passive* that points to the most recently created passive node. Another global variable, *printed_node*, is used to help print out the paragraph when detailed information about the line-breaking computation is being displayed.

```
define passive_node_size = 2 { number of words in passive nodes }
define cur_break ≡ rlink { in passive node, points to position of this breakpoint }
define prev_break ≡ llink { points to passive node that should precede this one }
define serial ≡ info { serial number for symbolic identification }
```

⟨ Global variables 13 ⟩ +≡

```
passive: pointer; { most recent node on passive list }
```

```
printed_node: pointer; { most recent node that has been printed }
```

```
pass_number: halfword; { the number of passive nodes allocated on this pass }
```

998. The active list also contains “delta” nodes that help the algorithm compute the badness of individual lines. Such nodes appear only between two active nodes, and they have $type = delta_node$. If p and r are active nodes and if q is a delta node between them, so that $link(p) = q$ and $link(q) = r$, then q tells the space difference between lines in the horizontal list that start after breakpoint p and lines that start after breakpoint r . In other words, if we know the length of the line that starts after p and ends at our current position, then the corresponding length of the line that starts after r is obtained by adding the amounts in node q . A delta node contains six scaled numbers, since it must record the net change in glue stretchability with respect to all orders of infinity. The natural width difference appears in $mem[q + 1].sc$; the stretch differences in units of pt, fil, fill, and filll appear in $mem[q + 2 \dots q + 5].sc$; and the shrink difference appears in $mem[q + 6].sc$. The *subtype* field of a delta node is not used.

```
define delta_node_size = 9 { number of words in a delta node }
define delta_node = 2 { type field in a delta node }
```

999. As the algorithm runs, it maintains a set of six delta-like registers for the length of the line following the first active breakpoint to the current position in the given hlist. When it makes a pass through the active list, it also maintains a similar set of six registers for the length following the active breakpoint of current interest. A third set holds the length of an empty line (namely, the sum of `\leftskip` and `\rightskip`); and a fourth set is used to create new delta nodes.

When we pass a delta node we want to do operations like

```
for k ← 1 to 6 do cur_active_width[k] ← cur_active_width[k] + mem[q + k].sc;
```

and we want to do this without the overhead of `for` loops. The `do_all_six` macro makes such six-tuples convenient.

```
define do_all_six(#) ≡ #(1); #(2); #(3); #(4); #(5); #(6)
define do_seven_eight(#)
  if pdf_adjust_spacing > 1 then
    begin #(7); #(8);
    end
define do_all_eight(#) ≡ do_all_six(#); do_seven_eight(#)
define do_one_seven_eight(#) ≡ #(1); do_seven_eight(#)
define total_font_stretch ≡ cur_active_width[7]
define total_font_shrink ≡ cur_active_width[8]
define save_active_width(#) ≡ prev_active_width[#] ← active_width[#]
define restore_active_width(#) ≡ active_width[#] ← prev_active_width[#]

⟨ Global variables 13 ⟩ +==
active_width: array [1 .. 8] of scaled; { distance from first active node to cur_p }
cur_active_width: array [1 .. 8] of scaled; { distance from current active node }
background: array [1 .. 8] of scaled; { length of an “empty” line }
break_width: array [1 .. 8] of scaled; { length being computed after current break }
auto_breaking: boolean; { make auto_breaking accessible out of line_break }
prev_p: pointer; { make prev_p accessible out of line_break }
first_p: pointer; { to access the first node of the paragraph }
prev_char_p: pointer; { pointer to the previous char of an implicit kern }
next_char_p: pointer; { pointer to the next char of an implicit kern }
try_prev_break: boolean; { force break at the previous legal breakpoint? }
prev_legal: pointer; { the previous legal breakpoint }
prev_prev_legal: pointer; { to save prev_p corresponding to prev_legal }
prev_auto_breaking: boolean; { to save auto_breaking corresponding to prev_legal }
prev_active_width: array [1 .. 8] of scaled; { to save active_width corresponding to prev_legal }
rejected_cur_p: pointer; { the last cur_p that has been rejected }
before_rejected_cur_p: boolean; { cur_p is still before rejected_cur_p? }
max_stretch_ratio: integer; { maximal stretch ratio of expanded fonts }
max_shrink_ratio: integer; { maximal shrink ratio of expanded fonts }
cur_font_step: integer; { the current step of expanded fonts }
```

1000. Let's state the principles of the delta nodes more precisely and concisely, so that the following programs will be less obscure. For each legal breakpoint p in the paragraph, we define two quantities $\alpha(p)$ and $\beta(p)$ such that the length of material in a line from breakpoint p to breakpoint q is $\gamma + \beta(q) - \alpha(p)$, for some fixed γ . Intuitively, $\alpha(p)$ and $\beta(q)$ are the total length of material from the beginning of the paragraph to a point "after" a break at p and to a point "before" a break at q ; and γ is the width of an empty line, namely the length contributed by `\leftskip` and `\rightskip`.

Suppose, for example, that the paragraph consists entirely of alternating boxes and glue skips; let the boxes have widths $x_1 \dots x_n$ and let the skips have widths $y_1 \dots y_n$, so that the paragraph can be represented by $x_1 y_1 \dots x_n y_n$. Let p_i be the legal breakpoint at y_i ; then $\alpha(p_i) = x_1 + y_1 + \dots + x_i + y_i$, and $\beta(p_i) = x_1 + y_1 + \dots + x_i$. To check this, note that the length of material from p_2 to p_5 , say, is $\gamma + x_3 + y_3 + x_4 + y_4 + x_5 = \gamma + \beta(p_5) - \alpha(p_2)$.

The quantities α , β , γ involve glue stretchability and shrinkability as well as a natural width. If we were to compute $\alpha(p)$ and $\beta(p)$ for each p , we would need multiple precision arithmetic, and the multiprecision numbers would have to be kept in the active nodes. TeX avoids this problem by working entirely with relative differences or "deltas." Suppose, for example, that the active list contains $a_1 \delta_1 a_2 \delta_2 a_3$, where the a 's are active breakpoints and the δ 's are delta nodes. Then $\delta_1 = \alpha(a_1) - \alpha(a_2)$ and $\delta_2 = \alpha(a_2) - \alpha(a_3)$. If the line breaking algorithm is currently positioned at some other breakpoint p , the `active_width` array contains the value $\gamma + \beta(p) - \alpha(a_1)$. If we are scanning through the list of active nodes and considering a tentative line that runs from a_2 to p , say, the `cur_active_width` array will contain the value $\gamma + \beta(p) - \alpha(a_2)$. Thus, when we move from a_2 to a_3 , we want to add $\alpha(a_2) - \alpha(a_3)$ to `cur_active_width`; and this is just δ_2 , which appears in the active list between a_2 and a_3 . The `background` array contains γ . The `break_width` array will be used to calculate values of new delta nodes when the active list is being updated.

1001. Glue nodes in a horizontal list that is being paragraphed are not supposed to include "infinite" shrinkability; that is why the algorithm maintains four registers for stretching but only one for shrinking. If the user tries to introduce infinite shrinkability, the shrinkability will be reset to finite and an error message will be issued. A boolean variable `no_shrink_error_yet` prevents this error message from appearing more than once per paragraph.

```
define check_shrinkage (#) ≡
  if (shrink_order (#) ≠ normal) ∧ (shrink (#) ≠ 0) then
    begin # ← finite_shrink (#);
    end
⟨ Global variables 13 ⟩ +==
no_shrink_error_yet: boolean; { have we complained about infinite shrinkage? }
```

1002. \langle Declare subprocedures for *line_break* 1002 $\rangle \equiv$

```
function finite_shrink(p : pointer): pointer; { recovers from infinite shrinkage }
  var q: pointer; { new glue specification }
  begin if no_shrink_error_yet then
    begin no_shrink_error_yet  $\leftarrow$  false;
    stat if tracing_paragraphs > 0 then end_diagnostic(true);
    tats print_err("Infinite_glue_shrinkage_found_in_a_paragraph");
    help5("The_paragraph_just-ended_includes_some_glue_that_has")
    ("infinite_shrinkability,_e.g.,`\hskip Opt minus 1fil`.")
    ("Such_glue_doesn't_belong_there---itAllows_a_paragraph")
    ("of_any_length_to_fit_on_one_line._But_it's_safe_to_proceed,")
    ("since_the_offensive_shrinkability_has_been_made_finite."); error;
    stat if tracing_paragraphs > 0 then begin_diagnostic;
    tats
    end;
  q  $\leftarrow$  new_spec(p); shrink_order(q)  $\leftarrow$  normal; delete_glue_ref(p); finite_shrink  $\leftarrow$  q;
end;
```

See also sections 1005, 1053, 1072, and 1119.

This code is used in section 991.

1003. \langle Get ready to start line breaking 992 $\rangle +\equiv$

```
no_shrink_error_yet  $\leftarrow$  true;
check_shrinkage(left_skip); check_shrinkage(right_skip);
q  $\leftarrow$  left_skip; r  $\leftarrow$  right_skip; background[1]  $\leftarrow$  width(q) + width(r);
background[2]  $\leftarrow$  0; background[3]  $\leftarrow$  0; background[4]  $\leftarrow$  0; background[5]  $\leftarrow$  0;
background[2 + stretch_order(q)]  $\leftarrow$  stretch(q);
background[2 + stretch_order(r)]  $\leftarrow$  background[2 + stretch_order(r)] + stretch(r);
background[6]  $\leftarrow$  shrink(q) + shrink(r);
if pdf_adjust_spacing > 1 then
  begin background[7]  $\leftarrow$  0; background[8]  $\leftarrow$  0; max_stretch_ratio  $\leftarrow$  -1; max_shrink_ratio  $\leftarrow$  -1;
  cur_font_step  $\leftarrow$  -1; prev_char_p  $\leftarrow$  null;
  end;
⟨ Check for special treatment of last line of paragraph 1843 ⟩;
```

1004. A pointer variable *cur_p* runs through the given horizontal list as we look for breakpoints. This variable is global, since it is used both by *line_break* and by its subprocedure *try_break*.

Another global variable called *threshold* is used to determine the feasibility of individual lines: Breakpoints are feasible if there is a way to reach them without creating lines whose badness exceeds *threshold*. (The badness is compared to *threshold* before penalties are added, so that penalty values do not affect the feasibility of breakpoints, except that no break is allowed when the penalty is 10000 or more.) If *threshold* is 10000 or more, all legal breaks are considered feasible, since the *badness* function specified above never returns a value greater than 10000.

Up to three passes might be made through the paragraph in an attempt to find at least one set of feasible breakpoints. On the first pass, we have *threshold* = *pretolerance* and *second_pass* = *final_pass* = *false*. If this pass fails to find a feasible solution, *threshold* is set to *tolerance*, *second_pass* is set *true*, and an attempt is made to hyphenate as many words as possible. If that fails too, we add *emergency_stretch* to the background stretchability and set *final_pass* = *true*.

\langle Global variables 13 $\rangle +\equiv$

```
cur_p: pointer; { the current breakpoint under consideration }
second_pass: boolean; { is this our second attempt to break this paragraph? }
final_pass: boolean; { is this our final attempt to break this paragraph? }
threshold: integer; { maximum badness on feasible lines }
```

1005. 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 }
define cp_skipable (#) ≡ { skipable nodes at the margins during character protrusion }
    (¬is_char_node (#) ∧ ((type (#) = ins_node) ∨ (type (#) = mark_node) ∨ (type (#) =
        adjust_node) ∨ (type (#) = penalty_node) ∨ ((type (#) = whatsit_node) ∧ (subtype (#) ≠
        pdf_refximage_node) ∧ (subtype (#) ≠ pdf_refxform_node))
    { reference to an image or XObject form }
    ∨((type (#) = disc_node) ∧ (pre_break (#) = null) ∧ (post_break (#) = null) ∧ (replace_count (#) = 0))
    { an empty disc_node }
    ∨((type (#) = math_node) ∧ (width (#) = 0)) ∨ ((type (#) = kern_node) ∧ (width (#) = 0) ∨ (subtype (#) =
        normal) ∨ (subtype (#) = auto_kern))) ∨ ((type (#) = glue_node) ∧ (glue_ptr (#) = zero_glue)) ∨
    ((type (#) = hlist_node) ∧ (width (#) = 0) ∧ (height (#) = 0) ∧ (depth (#) = 0) ∧ (list_ptr (#) = null)))
< Declare subprocedures for line_break 1002 > +≡
procedure push_node (p : pointer);
    begin if hlist_stack_level > max_hlist_stack then pdf_error ("push_node", "stack_overflow");
    hlist_stack [hlist_stack_level] ← p; hlist_stack_level ← hlist_stack_level + 1;
    end;
function pop_node: pointer;
    begin hlist_stack.level ← hlist_stack.level - 1;
    if hlist_stack_level < 0 then { would point to some bug }
        pdf_error ("pop_node", "stack_underflow(internal_error)");
    pop_node ← hlist_stack [hlist_stack_level];
    end;
function find_protchar_left (l : pointer; d : boolean): pointer;
    { searches left to right from list head l, returns 1st non-skipable item }
    var t: pointer; run: boolean;
    begin if (link (l) ≠ null) ∧ (type (l) = hlist_node) ∧ (width (l) = 0) ∧ (height (l) = 0) ∧ (depth (l) =
        0) ∧ (list_ptr (l) = null) then l ← link (l) { for paragraph start with \parindent = 0pt }
    else if d then
        while (link (l) ≠ null) ∧ (¬(is_char_node (l) ∨ non_discardable (l))) do l ← link (l);
        { std. discardables at line break, TEXbook, p 95 }
    hlist_stack_level ← 0; run ← true;
    repeat t ← l;
    while run ∧ (type (l) = hlist_node) ∧ (list_ptr (l) ≠ null) do
        begin push_node (l); l ← list_ptr (l);
        end;
    while run ∧ cp_skipable (l) do
        begin while (link (l) = null) ∧ (hlist_stack_level > 0) do
            begin l ← pop_node; { don't visit this node again }
            end;
        if link (l) ≠ null then l ← link (l)
        else if hlist_stack_level = 0 then run ← false
        end;
```

```

until  $t = l$ ;
 $find\_protchar\_left \leftarrow l$ ;
end;
function  $find\_protchar\_right(l, r : \text{pointer})$ :  $\text{pointer}$ ;
    { searches right to left from list tail  $r$  to head  $l$ , returns 1st non-skipable item }
    var  $t : \text{pointer}$ ;  $run : \text{boolean}$ ;
    begin  $find\_protchar\_right \leftarrow \text{null}$ ;
    if  $r = \text{null}$  then return;
     $hlist\_stack\_level \leftarrow 0$ ;  $run \leftarrow \text{true}$ ;
    repeat  $t \leftarrow r$ ;
        while  $run \wedge (\text{type}(r) = hlist\_node) \wedge (\text{list\_ptr}(r) \neq \text{null})$  do
            begin  $push\_node(l)$ ;  $push\_node(r)$ ;  $l \leftarrow \text{list\_ptr}(r)$ ;  $r \leftarrow l$ ;
            while  $\text{link}(r) \neq \text{null}$  do  $r \leftarrow \text{link}(r)$ ;
            end;
        while  $run \wedge cp\_skipable(r)$  do
            begin while  $(r = l) \wedge (hlist\_stack\_level > 0)$  do
                begin  $r \leftarrow pop\_node$ ; { don't visit this node again }
                 $l \leftarrow pop\_node$ ;
                end;
            if  $(r \neq l) \wedge (r \neq \text{null})$  then  $r \leftarrow prev\_rightmost(l, r)$ 
            else if  $(r = l) \wedge (hlist\_stack\_level = 0)$  then  $run \leftarrow \text{false}$ 
            end;
        until  $t = r$ ;
         $find\_protchar\_right \leftarrow r$ ;
    end;
function  $total\_pw(q, p : \text{pointer})$ :  $\text{scaled}$ ;
    { returns the total width of character protrusion of a line;  $cur\_break(break\_node(q))$  and  $p$  is the
     leftmost resp. rightmost node in the horizontal list representing the actual line }
    var  $l, r : \text{pointer}$ ;  $n : \text{integer}$ ;
    begin if  $break\_node(q) = \text{null}$  then  $l \leftarrow first\_p$ 
    else  $l \leftarrow cur\_break(break\_node(q))$ ;
     $r \leftarrow prev\_rightmost(prev\_p, p)$ ; { get  $\text{link}(r) = p$  }
    { let's look at the right margin first }
    @{ $short\_display\_n(r, 2)$ ;  $print("amp;")$ ;  $short\_display\_n(p, 2)$ ;  $print\_ln$ ; @}
    if  $(p \neq \text{null}) \wedge (\text{type}(p) = disc\_node) \wedge (pre\_break(p) \neq \text{null})$  then
        { a  $disc\_node$  with non-empty  $pre\_break$ , protrude the last char of  $pre\_break$  }
        begin  $r \leftarrow pre\_break(p)$ ;
        while  $\text{link}(r) \neq \text{null}$  do  $r \leftarrow \text{link}(r)$ ;
        end
    else  $r \leftarrow find\_protchar\_right(l, r)$ ; { now the left margin }
    @{ $short\_display\_n(l, 2)$ ;  $print\_ln$ ;  $breadth\_max \leftarrow 10$ ;  $depth\_threshold \leftarrow 2$ ;  $show\_node\_list(l)$ ;  $print\_ln$ ;
    @}
    if  $(l \neq \text{null}) \wedge (\text{type}(l) = disc\_node)$  then
        begin if  $post\_break(l) \neq \text{null}$  then
            begin  $l \leftarrow post\_break(l)$ ; { protrude the first char }
            goto done;
            end
        else { discard  $replace\_count(l)$  nodes }
        begin  $n \leftarrow replace\_count(l)$ ;  $l \leftarrow \text{link}(l)$ ;
        while  $n > 0$  do
            begin if  $\text{link}(l) \neq \text{null}$  then  $l \leftarrow \text{link}(l)$ ;
             $decr(n)$ ;
        end
    end

```

```

    end;
end;
end;

l ← find_protchar_left(l, true);
done: total_pw ← left_pw(l) + right_pw(r);
    end;

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 }
margin_kern_stretch: scaled; margin_kern_shrink: scaled; lp, rp, cp: pointer; 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 1006}

begin {Make sure that pi is in the proper range 1007};
no_break_yet ← true; prev_r ← active; old_l ← 0; do_all_eight(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 1008};
{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 1011};
{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 1027};
end;
exit: stat {Update the value of printed_node for symbolic displays 1034} tats
end;

```

1006. {Other local variables for try_break 1006} ≡
prev_prev_r: pointer; {a step behind prev_r, if type(prev_r) = delta_node}
s: pointer; {runs through nodes ahead of cur_p}
q: pointer; {points to a new node being created}
v: pointer; {points to a glue specification or a node ahead of cur_p}
t: integer; {node count, if cur_p is a discretionary node}
f: internal_font_number; {used in character width calculation}
l: halfword; {line number of current active node}
node_r_stays_active: boolean; {should node r remain in the active list?}
line_width: scaled; {the current line will be justified to this width}
fit_class: very_loose_fit .. tight_fit; {possible fitness class of test line}
b: halfword; {badness of test line}
d: integer; {demerits of test line}
artificial_demerits: boolean; {has d been forced to zero?}
save_link: pointer; {temporarily holds value of link(cur_p)}
shortfall: scaled; {used in badness calculations}

See also section 1844.

This code is used in section 1005.

1007. {Make sure that pi is in the proper range 1007} ≡
if *abs(pi)* ≥ inf_penalty then
 if *pi* > 0 then return {this breakpoint is inhibited by infinite penalty}
 else *pi* ← eject_penalty {this breakpoint will be forced}

This code is used in section 1005.

1008. The following code uses the fact that $\text{type}(\text{last_active}) \neq \text{delta_node}$.

```
define update_width(#) ≡ cur_active_width[#] ← cur_active_width[#] + mem[r + #].sc
⟨ If node r is of type delta_node, update cur_active_width, set prev_r and prev_prev_r, then goto
    continue 1008 ⟩ ≡
if type(r) = delta_node then
begin do_all_eight(update_width); prev_prev_r ← prev_r; prev_r ← r; goto continue;
end
```

This code is used in section 1005.

1009. As we consider various ways to end a line at cur_p , in a given line number class, we keep track of the best total demerits known, in an array with one entry for each of the fitness classifications. For example, $\text{minimal_demerits}[\text{tight_fit}]$ contains the fewest total demerits of feasible line breaks ending at cur_p with a tight_fit line; $\text{best_place}[\text{tight_fit}]$ points to the passive node for the break before cur_p that achieves such an optimum; and $\text{best_pl_line}[\text{tight_fit}]$ is the line_number field in the active node corresponding to $\text{best_place}[\text{tight_fit}]$. When no feasible break sequence is known, the minimal_demerits entries will be equal to awful_bad , which is $2^{30} - 1$. Another variable, minimum_demerits , keeps track of the smallest value in the minimal_demerits array.

```
define awful_bad ≡ '777777777777 { more than a billion demerits }
⟨ Global variables 13 ⟩ +≡
minimal_demerits: array [very_loose_fit .. tight_fit] of integer;
    { best total demerits known for current line class and position, given the fitness }
minimum_demerits: integer; { best total demerits known for current line class and position }
best_place: array [very_loose_fit .. tight_fit] of pointer; { how to achieve minimal_demerits }
best_pl_line: array [very_loose_fit .. tight_fit] of halfword; { corresponding line number }
```

1010. ⟨ Get ready to start line breaking 992 ⟩ +≡

```
minimum_demerits ← awful_bad; minimal_demerits[tight_fit] ← awful_bad;
minimal_demerits[decent_fit] ← awful_bad; minimal_demerits[loose_fit] ← awful_bad;
minimal_demerits[very_loose_fit] ← awful_bad;
```

1011. The first part of the following code is part of TeX's inner loop, so we don't want to waste any time. The current active node, namely node r , contains the line number that will be considered next. At the end of the list we have arranged the data structure so that $r = \text{last_active}$ and $\text{line_number}(\text{last_active}) > \text{old_l}$.

⟨ If a line number class has ended, create new active nodes for the best feasible breaks in that class; then

```
return if  $r = \text{last\_active}$ , otherwise compute the new line_width 1011 ⟩ ≡
begin  $l \leftarrow \text{line\_number}(r)$ ;
if  $l > \text{old\_l}$  then
begin { now we are no longer in the inner loop }
if ( $\text{minimum_demerits} < \text{awful_bad}$ )  $\wedge ((\text{old\_l} \neq \text{easy\_line}) \vee (r = \text{last\_active}))$  then
    { Create new active nodes for the best feasible breaks just found 1012 };
if  $r = \text{last\_active}$  then return;
{ Compute the new line width 1026 };
end;
end
```

This code is used in section 1005.

1012. It is not necessary to create new active nodes having *minimal_demerits* greater than *minimum_demerits* + $\text{abs}(\text{adj_demerits})$, since such active nodes will never be chosen in the final paragraph breaks. This observation allows us to omit a substantial number of feasible breakpoints from further consideration.

```
< Create new active nodes for the best feasible breaks just found 1012 > ≡
begin if no_break_yet then < Compute the values of break_width 1013 >;
  < Insert a delta node to prepare for breaks at cur_p 1019 >;
  if abs(adj_demerits) ≥ awful_bad - minimum_demerits then minimum_demerits ← awful_bad - 1
  else minimum_demerits ← minimum_demerits + abs(adj_demerits);
  for fit_class ← very_loose_fit to tight_fit do
    begin if minimal_demerits[fit_class] ≤ minimum_demerits then
      < Insert a new active node from best_place[fit_class] to cur_p 1021 >;
      minimal_demerits[fit_class] ← awful_bad;
    end;
    minimum_demerits ← awful_bad; < Insert a delta node to prepare for the next active node 1020 >;
  end
```

This code is used in section 1011.

1013. When we insert a new active node for a break at *cur_p*, suppose this new node is to be placed just before active node *a*; then we essentially want to insert ‘ $\delta \text{cur}_p \delta'$ before *a*, where $\delta = \alpha(a) - \alpha(\text{cur}_p)$ and $\delta' = \alpha(\text{cur}_p) - \alpha(a)$ in the notation explained above. The *cur_active_width* array now holds $\gamma + \beta(\text{cur}_p) - \alpha(a)$; so δ can be obtained by subtracting *cur_active_width* from the quantity $\gamma + \beta(\text{cur}_p) - \alpha(\text{cur}_p)$. The latter quantity can be regarded as the length of a line “from *cur_p* to *cur_p*”; we call it the *break_width* at *cur_p*.

The *break_width* is usually negative, since it consists of the background (which is normally zero) minus the width of nodes following *cur_p* that are eliminated after a break. If, for example, node *cur_p* is a glue node, the width of this glue is subtracted from the background; and we also look ahead to eliminate all subsequent glue and penalty and kern and math nodes, subtracting their widths as well.

Kern nodes do not disappear at a line break unless they are *explicit*.

```
define set_break_width_to_background(#[#]) ≡ break_width[#] ← background[#[#]]
< Compute the values of break_width 1013 > ≡
begin no_break_yet ← false; do_all_eight(set_break_width_to_background); s ← cur_p;
if break_type > unhyphenated then
  if cur_p ≠ null then < Compute the discretionary break_width values 1016 >;
while s ≠ null do
  begin if is_char_node(s) then goto done;
  case type(s) of
    glue_node: < Subtract glue from break_width 1014 >;
    penalty_node: do_nothing;
    math_node: break_width[1] ← break_width[1] - width(s);
    kern_node: if subtype(s) ≠ explicit then goto done
      else break_width[1] ← break_width[1] - width(s);
    othercases goto done
  endcases;
  s ← link(s);
end;
done: end
```

This code is used in section 1012.

1014. $\langle \text{Subtract glue from } \text{break_width } 1014 \rangle \equiv$

```
begin  $v \leftarrow \text{glue\_ptr}(s)$ ;  $\text{break\_width}[1] \leftarrow \text{break\_width}[1] - \text{width}(v)$ ;
 $\text{break\_width}[2 + \text{stretch\_order}(v)] \leftarrow \text{break\_width}[2 + \text{stretch\_order}(v)] - \text{stretch}(v)$ ;
 $\text{break\_width}[6] \leftarrow \text{break\_width}[6] - \text{shrink}(v)$ ;
end
```

This code is used in section 1013.

1015. When cur_p is a discretionary break, the length of a line “from cur_p to cur_p ” has to be defined properly so that the other calculations work out. Suppose that the pre-break text at cur_p has length l_0 , the post-break text has length l_1 , and the replacement text has length l . Suppose also that q is the node following the replacement text. Then length of a line from cur_p to q will be computed as $\gamma + \beta(q) - \alpha(\text{cur_p})$, where $\beta(q) = \beta(\text{cur_p}) - l_0 + l$. The actual length will be the background plus l_1 , so the length from cur_p to cur_p should be $\gamma + l_0 + l_1 - l$. If the post-break text of the discretionary is empty, a break may also discard q ; in that unusual case we subtract the length of q and any other nodes that will be discarded after the discretionary break.

The value of l_0 need not be computed, since `line_break` will put it into the global variable `disc_width` before calling `try_break`.

```
define  $\text{reset\_disc\_width}(\#) \equiv \text{disc\_width}[\#] \leftarrow 0$ 
define  $\text{add\_disc\_width\_to\_break\_width}(\#) \equiv \text{break\_width}[\#] \leftarrow \text{break\_width}[\#] + \text{disc\_width}[\#]$ 
define  $\text{add\_disc\_width\_to\_active\_width}(\#) \equiv \text{active\_width}[\#] \leftarrow \text{active\_width}[\#] + \text{disc\_width}[\#]$ 
define  $\text{sub\_disc\_width\_from\_active\_width}(\#) \equiv \text{active\_width}[\#] \leftarrow \text{active\_width}[\#] - \text{disc\_width}[\#]$ 
define  $\text{add\_char\_stretch\_end}(\#) \equiv \text{char\_stretch}(f, \#)$ 
define  $\text{add\_char\_stretch}(\#) \equiv \# \leftarrow \# + \text{add\_char\_stretch\_end}$ 
define  $\text{add\_char\_shrink\_end}(\#) \equiv \text{char\_shrink}(f, \#)$ 
define  $\text{add\_char\_shrink}(\#) \equiv \# \leftarrow \# + \text{add\_char\_shrink\_end}$ 
define  $\text{sub\_char\_stretch\_end}(\#) \equiv \text{char\_stretch}(f, \#)$ 
define  $\text{sub\_char\_stretch}(\#) \equiv \# \leftarrow \# - \text{sub\_char\_stretch\_end}$ 
define  $\text{sub\_char\_shrink\_end}(\#) \equiv \text{char\_shrink}(f, \#)$ 
define  $\text{sub\_char\_shrink}(\#) \equiv \# \leftarrow \# - \text{sub\_char\_shrink\_end}$ 
define  $\text{add\_kern\_stretch\_end}(\#) \equiv \text{kern\_stretch}(\#)$ 
define  $\text{add\_kern\_stretch}(\#) \equiv \# \leftarrow \# + \text{add\_kern\_stretch\_end}$ 
define  $\text{add\_kern\_shrink\_end}(\#) \equiv \text{kern\_shrink}(\#)$ 
define  $\text{add\_kern\_shrink}(\#) \equiv \# \leftarrow \# + \text{add\_kern\_shrink\_end}$ 
define  $\text{sub\_kern\_stretch\_end}(\#) \equiv \text{kern\_stretch}(\#)$ 
define  $\text{sub\_kern\_stretch}(\#) \equiv \# \leftarrow \# - \text{sub\_kern\_stretch\_end}$ 
define  $\text{sub\_kern\_shrink\_end}(\#) \equiv \text{kern\_shrink}(\#)$ 
define  $\text{sub\_kern\_shrink}(\#) \equiv \# \leftarrow \# - \text{sub\_kern\_shrink\_end}$ 

 $\langle \text{Global variables } 13 \rangle + \equiv$ 
 $\text{disc\_width: array [1 .. 8] of scaled; } \{ \text{the length of discretionary material preceding a break} \}$ 
```

1016. \langle Compute the discretionary *break_width* values 1016 $\rangle \equiv$

```

begin  $t \leftarrow \text{replace\_count}(\text{cur\_p})$ ;  $v \leftarrow \text{cur\_p}$ ;  $s \leftarrow \text{post\_break}(\text{cur\_p})$ ;
while  $t > 0$  do
  begin  $\text{decr}(t)$ ;  $v \leftarrow \text{link}(v)$ ;  $\langle$  Subtract the width of node  $v$  from break_width 1017  $\rangle$ ;
  end;
while  $s \neq \text{null}$  do
  begin  $\langle$  Add the width of node  $s$  to break_width 1018  $\rangle$ ;
   $s \leftarrow \text{link}(s)$ ;
  end;
 $\text{do\_one\_seven\_eight}(\text{add\_disc\_width\_to\_break\_width})$ ;
if  $\text{post\_break}(\text{cur\_p}) = \text{null}$  then  $s \leftarrow \text{link}(v)$ ; { nodes may be discardable after the break }
end

```

This code is used in section 1013.

1017. Replacement texts and discretionary texts are supposed to contain only character nodes, kern nodes, ligature nodes, and box or rule nodes.

\langle Subtract the width of node v from *break_width* 1017 $\rangle \equiv$

```

if  $\text{is\_char\_node}(v)$  then
  begin  $f \leftarrow \text{font}(v)$ ;  $\text{break\_width}[1] \leftarrow \text{break\_width}[1] - \text{char\_width}(f)(\text{char\_info}(f)(\text{character}(v)))$ ;
  if  $(\text{pdf\_adjust\_spacing} > 1) \wedge \text{check\_expand\_pars}(f)$  then
    begin  $\text{prev\_char\_p} \leftarrow v$ ;  $\text{sub\_char\_stretch}(\text{break\_width}[7])(\text{character}(v))$ ;
     $\text{sub\_char\_shrink}(\text{break\_width}[8])(\text{character}(v))$ ;
    end;
  end
else case  $\text{type}(v)$  of
  ligature_node: begin  $f \leftarrow \text{font}(\text{lig\_char}(v))$ ;
   $\text{break\_width}[1] \leftarrow \text{break\_width}[1] - \text{char\_width}(f)(\text{char\_info}(f)(\text{character}(\text{lig\_char}(v))))$ ;
  if  $(\text{pdf\_adjust\_spacing} > 1) \wedge \text{check\_expand\_pars}(f)$  then
    begin  $\text{prev\_char\_p} \leftarrow v$ ;  $\text{sub\_char\_stretch}(\text{break\_width}[7])(\text{character}(\text{lig\_char}(v)))$ ;
     $\text{sub\_char\_shrink}(\text{break\_width}[8])(\text{character}(\text{lig\_char}(v)))$ ;
    end;
  end;
  hlist_node, vlist_node, rule_node, kern_node: begin  $\text{break\_width}[1] \leftarrow \text{break\_width}[1] - \text{width}(v)$ ;
  if  $(\text{type}(v) = \text{kern\_node}) \wedge (\text{pdf\_adjust\_spacing} > 1) \wedge (\text{subtype}(v) = \text{normal})$  then
    begin  $\text{sub\_kern\_stretch}(\text{break\_width}[7])(v)$ ;  $\text{sub\_kern\_shrink}(\text{break\_width}[8])(v)$ ;
    end;
  end;
othercases  $\text{confusion}(\text{"disc1"})$ 
endcases

```

This code is used in section 1016.

```

1018. ⟨ Add the width of node  $s$  to  $break\_width$  1018 ⟩ ≡
  if  $is\_char\_node(s)$  then
    begin  $f \leftarrow font(s)$ ;  $break\_width[1] \leftarrow break\_width[1] + char\_width(f)(char\_info(f)(character(s)))$ ;
    if  $(pdf\_adjust\_spacing > 1) \wedge check\_expand\_pars(f)$  then
      begin  $prev\_char\_p \leftarrow s$ ;  $add\_char\_stretch(break\_width[7])(character(s))$ ;
       $add\_char\_shrink(break\_width[8])(character(s))$ ;
      end;
    end
  else case  $type(s)$  of
    ligature_node: begin  $f \leftarrow font(lig\_char(s))$ ;
       $break\_width[1] \leftarrow break\_width[1] + char\_width(f)(char\_info(f)(character(lig\_char(s))))$ ;
      if  $(pdf\_adjust\_spacing > 1) \wedge check\_expand\_pars(f)$  then
        begin  $prev\_char\_p \leftarrow s$ ;  $add\_char\_stretch(break\_width[7])(character(lig\_char(s)))$ ;
         $add\_char\_shrink(break\_width[8])(character(lig\_char(s)))$ ;
        end;
      end;
    hlist_node, vlist_node, rule_node, kern_node: begin  $break\_width[1] \leftarrow break\_width[1] + width(s)$ ;
      if  $(type(s) = kern\_node) \wedge (pdf\_adjust\_spacing > 1) \wedge (subtype(s) = normal)$  then
        begin  $add\_kern\_stretch(break\_width[7])(s)$ ;  $add\_kern\_shrink(break\_width[8])(s)$ ;
        end;
      end;
    othercases  $confusion("disc2")$ 
  endcases

```

This code is used in section 1016.

1019. We use the fact that $type(active) \neq delta_node$.

```

define  $convert\_to\_break\_width(\#) \equiv mem[prev\_r + \#].sc \leftarrow$ 
          $mem[prev\_r + \#].sc - cur\_active\_width[\#] + break\_width[\#]$ 
define  $store\_break\_width(\#) \equiv active\_width[\#] \leftarrow break\_width[\#]$ 
define  $new\_delta\_to\_break\_width(\#) \equiv mem[q + \#].sc \leftarrow break\_width[\#] - cur\_active\_width[\#]$ 

⟨ Insert a delta node to prepare for breaks at  $cur\_p$  1019 ⟩ ≡
  if  $type(prev\_r) = delta\_node$  then { modify an existing delta node }
    begin  $do\_all\_eight(convert\_to\_break\_width)$ ;
    end
  else if  $prev\_r = active$  then { no delta node needed at the beginning }
    begin  $do\_all\_eight(store\_break\_width)$ ;
    end
  else begin  $q \leftarrow get\_node(delta\_node\_size)$ ;  $link(q) \leftarrow r$ ;  $type(q) \leftarrow delta\_node$ ;
     $subtype(q) \leftarrow 0$ ; { the subtype is not used }
     $do\_all\_eight(new\_delta\_to\_break\_width)$ ;  $link(prev\_r) \leftarrow q$ ;  $prev\_prev\_r \leftarrow prev\_r$ ;  $prev\_r \leftarrow q$ ;
  end

```

This code is used in section 1012.

1020. When the following code is performed, we will have just inserted at least one active node before r , so $\text{type}(\text{prev_r}) \neq \text{delta_node}$.

```
define new_delta_from_break_width(#[#]) ≡ mem[q + #].sc ← cur_active_width[#] – break_width[#]
⟨ Insert a delta node to prepare for the next active node 1020 ⟩ ≡
if  $r \neq \text{last\_active}$  then
begin  $q \leftarrow \text{get\_node}(\text{delta\_node\_size})$ ;  $\text{link}(q) \leftarrow r$ ;  $\text{type}(q) \leftarrow \text{delta\_node}$ ;
 $\text{subtype}(q) \leftarrow 0$ ; { the subtype is not used }
do_all_eight(new_delta_from_break_width);  $\text{link}(\text{prev\_r}) \leftarrow q$ ;  $\text{prev\_prev\_r} \leftarrow \text{prev\_r}$ ;  $\text{prev\_r} \leftarrow q$ ;
end
```

This code is used in section 1012.

1021. 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 1021 ⟩ ≡
begin  $q \leftarrow \text{get\_node}(\text{passive\_node\_size})$ ;  $\text{link}(q) \leftarrow \text{passive}$ ;  $\text{passive} \leftarrow q$ ;  $\text{cur\_break}(q) \leftarrow \text{cur\_p}$ ;
stat incr(pass_number); serial(q) ← pass_number; tats
 $\text{prev\_break}(q) \leftarrow \text{best\_place}[fit\_class]$ ;
 $q \leftarrow \text{get\_node}(\text{active\_node\_size})$ ;  $\text{break\_node}(q) \leftarrow \text{passive}$ ;  $\text{line\_number}(q) \leftarrow \text{best\_pl\_line}[fit\_class] + 1$ ;
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 1851 ⟩;
 $\text{link}(q) \leftarrow r$ ;  $\text{link}(\text{prev\_r}) \leftarrow q$ ;  $\text{prev\_r} \leftarrow q$ ;
stat if tracing_paragraphs > 0 then ⟨ Print a symbolic description of the new break node 1022 ⟩;
tats
end
```

This code is used in section 1012.

1022. ⟨ Print a symbolic description of the new break node 1022 ⟩ ≡

```
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 1852 ⟩;
print("->@@");
if prev_break(passive) = null then print_char("0")
else print_int(serial(prev_break(passive)));
end
```

This code is used in section 1021.

1023. The length of lines depends on whether the user has specified `\parshape` or `\hangindent`. If `par_shape_ptr` is not null, it points to a $(2n + 1)$ -word record in `mem`, where the `info` in the first word contains the value of n , and the other $2n$ words contain the left margins and line lengths for the first n lines of the paragraph; the specifications for line n apply to all subsequent lines. If `par_shape_ptr = null`, the shape of the paragraph depends on the value of $n = \text{hang_after}$; if $n \geq 0$, hanging indentation takes place on lines $n + 1, n + 2, \dots$, otherwise it takes place on lines $1, \dots, |n|$. When hanging indentation is active, the left margin is `hang_indent`, if `hang_indent \geq 0`, else it is 0; the line length is `hsize - |hang_indent|`. The normal setting is `par_shape_ptr = null`, `hang_after = 1`, and `hang_indent = 0`. Note that if `hang_indent = 0`, the value of `hang_after` is irrelevant.

```
<Global variables 13> +≡
easy_line: halfword; { line numbers > easy_line are equivalent in break nodes }
last_special_line: halfword; { line numbers > last_special_line all have the same width }
first_width: scaled; { the width of all lines ≤ last_special_line, if no \parshape has been specified }
second_width: scaled; { the width of all lines > last_special_line }
first_indent: scaled; { left margin to go with first_width }
second_indent: scaled; { left margin to go with second_width }
```

1024. We compute the values of `easy_line` and the other local variables relating to line length when the `line_break` procedure is initializing itself.

```
<Get ready to start line breaking 992> +≡
if par_shape_ptr = null then
  if hang_indent = 0 then
    begin last_special_line ← 0; second_width ← hsize; second_indent ← 0;
    end
  else { Set line length parameters in preparation for hanging indentation 1025 }
else begin last_special_line ← info(par_shape_ptr) - 1;
  second_width ← mem[par_shape_ptr + 2 * (last_special_line + 1)].sc;
  second_indent ← mem[par_shape_ptr + 2 * last_special_line + 1].sc;
  end;
if looseness = 0 then easy_line ← last_special_line
else easy_line ← max_halfword
```

1025. { Set line length parameters in preparation for hanging indentation 1025 } ≡

```
begin last_special_line ← abs(hang_after);
if hang_after < 0 then
  begin first_width ← hsize - abs(hang_indent);
  if hang_indent ≥ 0 then first_indent ← hang_indent
  else first_indent ← 0;
  second_width ← hsize; second_indent ← 0;
  end
else begin first_width ← hsize; first_indent ← 0; second_width ← hsize - abs(hang_indent);
  if hang_indent ≥ 0 then second_indent ← hang_indent
  else second_indent ← 0;
  end;
end
```

This code is used in section 1024.

1026. When we come to the following code, we have just encountered the first active node r whose *line_number* field contains l . Thus we want to compute the length of the l th line of the current paragraph. Furthermore, we want to set *old_l* to the last number in the class of line numbers equivalent to l .

```
⟨Compute the new line width 1026⟩ ≡  
if  $l > \text{easy\_line}$  then  
  begin line_width  $\leftarrow \text{second\_width}$ ; old_l  $\leftarrow \text{max\_halfword} - 1$ ;  
  end  
else begin old_l  $\leftarrow l$ ;  
  if  $l > \text{last\_special\_line}$  then line_width  $\leftarrow \text{second\_width}$   
  else if par_shape_ptr = null then line_width  $\leftarrow \text{first\_width}$   
    else line_width  $\leftarrow \text{mem}[\text{par\_shape\_ptr} + 2 * l].sc$ ;  
  end
```

This code is used in section 1011.

1027. 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 1027 ⟩ ≡
begin artificial_demerits ← false;
 $shortfall \leftarrow line\_width - cur\_active\_width[1]$ ; { we're this much too short }
@{
if pdf_output > 2 then
    begin print_ln;
        if ( $r \neq null$ )  $\wedge$  (break_node( $r$ )  $\neq null$ ) then short_display_n(cur_break(break_node( $r$ )), 5);
        print_ln; short_display_n( $cur\_p$ , 5); print_ln;
    end;
}
@{
if pdf_protrude_chars > 1 then  $shortfall \leftarrow shortfall + total\_pw(r, cur\_p)$ ;
if (pdf_adjust_spacing > 1)  $\wedge$  ( $shortfall \neq 0$ ) then
    begin margin_kern_stretch ← 0; margin_kern_shrink ← 0;
        if pdf_protrude_chars > 1 then ⟨ Calculate variations of marginal kerns 822 ⟩;
        if ( $shortfall > 0$ )  $\wedge$  ((total_font_stretch + margin_kern_stretch) > 0) then
            begin if (total_font_stretch + margin_kern_stretch) >  $shortfall$  then
                 $shortfall \leftarrow ((total\_font\_stretch + margin\_kern\_stretch) \text{div } (max\_stretch\_ratio \text{div } cur\_font\_step)) \text{div } 2$ 
            else  $shortfall \leftarrow shortfall - (total\_font\_stretch + margin\_kern\_stretch)$ ;
            end
        else if ( $shortfall < 0$ )  $\wedge$  ((total_font_shrink + margin_kern_shrink) > 0) then
            begin if (total_font_shrink + margin_kern_shrink) >  $-shortfall$  then  $shortfall \leftarrow -((total\_font\_shrink + margin\_kern\_shrink) \text{div } (max\_shrink\_ratio \text{div } cur\_font\_step)) \text{div } 2$ 
            else  $shortfall \leftarrow shortfall + (total\_font\_shrink + margin\_kern\_shrink)$ ;
            end;
        end;
    if  $shortfall > 0$  then
        ⟨ Set the value of  $b$  to the badness for stretching the line, and compute the corresponding fit_class 1028 ⟩
    else ⟨ Set the value of  $b$  to the badness for shrinking the line, and compute the corresponding
        fit_class 1029 ⟩;
    if do_last_line_fit then ⟨ Adjust the additional data for last line 1849 ⟩;
found: if ( $b > inf\_bad$ )  $\vee$  ( $pi = eject\_penalty$ ) then ⟨ Prepare to deactivate node  $r$ , and goto deactivate
unless there is a reason to consider lines of text from  $r$  to  $cur\_p$  1030 ⟩
else begin prev_r ←  $r$ ;
    if  $b > threshold$  then goto continue;
    node_r_stays_active ← true;
end;
⟨ Record a new feasible break 1031 ⟩;
if node_r_stays_active then goto continue; { prev_r has been set to  $r$  }
deactivate: ⟨ Deactivate node  $r$  1036 ⟩;
end

```

This code is used in section 1005.

1028. When a line must stretch, the available stretchability can be found in the subarray $cur_active_width[2 \dots 5]$, 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 1028 > ≡
if (cur_active_width[3] ≠ 0) ∨ (cur_active_width[4] ≠ 0) ∨ (cur_active_width[5] ≠ 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 1846 >;
    shortfall ← 0;
end;
b ← 0; fit_class ← decent_fit; { infinite stretch }
end
else begin if shortfall > 7230584 then
if cur_active_width[2] < 1663497 then
begin b ← inf_bad; fit_class ← very_loose_fit; goto done1;
end;
b ← badness(shortfall, cur_active_width[2]);
if b > 12 then
if b > 99 then fit_class ← very_loose_fit
else fit_class ← loose_fit
else fit_class ← decent_fit;
done1: end
```

This code is used in section 1027.

1029. Shrinkability is never infinite in a paragraph; we can shrink the line from r to cur_p by at most $cur_active_width[6]$.

```
< Set the value of b to the badness for shrinking the line, and compute the corresponding fit_class 1029 > ≡
begin if -shortfall > cur_active_width[6] then b ← inf_bad + 1
else b ← badness(-shortfall, cur_active_width[6]);
if b > 12 then fit_class ← tight_fit else fit_class ← decent_fit;
end
```

This code is used in section 1027.

1030. During the final pass, we dare not lose all active nodes, lest we lose touch with the line breaks already found. The code shown here makes sure that such a catastrophe does not happen, by permitting overfull boxes as a last resort. This particular part of TeX was a source of several subtle bugs before the correct program logic was finally discovered; readers who seek to "improve" TeX should therefore think thrice before daring to make any changes here.

```
< Prepare to deactivate node r, and goto deactivate unless there is a reason to consider lines of text from r
to cur_p 1030 > ≡
begin if final_pass ∧ (minimum_demerits = awful_bad) ∧ (link(r) = last_active) ∧ (prev_r = active) then
    artificial_demerits ← true { set demerits zero, this break is forced }
else if b > threshold then goto deactivate;
node_r_stays_active ← false;
end
```

This code is used in section 1027.

1031. 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 1031 > ≡
  if artificial_demerits then  $d \leftarrow 0$ 
  else < Compute the demerits,  $d$ , from  $r$  to  $cur\_p$  1035 >;
  stat if tracing_paragraphs > 0 then < Print a symbolic description of this feasible break 1032 >;
  tats
   $d \leftarrow d + total\_demerits(r)$ ; { this is the minimum total demerits from the beginning to  $cur\_p$  via  $r$  }
  if  $d \leq minimal\_demerits[fit\_class]$  then
    begin  $minimal\_demerits[fit\_class] \leftarrow d$ ; best_place[fit_class]  $\leftarrow break\_node(r)$ ; best_pl_line[fit_class]  $\leftarrow l$ ;
    if do_last_line_fit then < Store additional data for this feasible break 1850 >;
    if  $d < minimum\_demerits$  then  $minimum\_demerits \leftarrow d$ ;
    end
```

This code is used in section 1027.

1032. < Print a symbolic description of this feasible break 1032 > ≡

```
begin if printed_node  $\neq cur\_p$  then
  < Print the list between printed_node and  $cur\_p$ , then set printed_node  $\leftarrow cur\_p$  1033 >;
  print_nl("@");
  if  $cur\_p = null$  then print_esc("par")
  else if type( $cur\_p$ )  $\neq glue\_node$  then
    begin if type( $cur\_p$ ) = penalty_node then print_esc("penalty")
      else if type( $cur\_p$ ) = disc_node then print_esc("discretionary")
        else if type( $cur\_p$ ) = kern_node then print_esc("kern")
          else print_esc("math");
    end;
  print("via @@");
  if break_node( $r$ ) = null then print_char("0")
  else print_int(serial(break_node( $r$ )));
  print(" b=");
  if  $b > inf\_bad$  then print_char("*") else print_int( $b$ );
  print(" p="); print_int( $pi$ ); print(" d=");
  if artificial_demerits then print_char("*") else print_int( $d$ );
end
```

This code is used in section 1031.

1033. < Print the list between *printed_node* and cur_p , then set *printed_node* $\leftarrow cur_p$ 1033 > ≡

```
begin print_nl("");
  if  $cur\_p = null$  then short_display(link(printed_node))
  else begin save_link  $\leftarrow link(cur\_p)$ ; link( $cur\_p$ )  $\leftarrow null$ ; print_nl("");
    short_display(link(printed_node)); link( $cur\_p$ )  $\leftarrow save\_link$ ;
  end;
  printed_node  $\leftarrow cur\_p$ ;
end
```

This code is used in section 1032.

1034. When the data for a discretionary break is being displayed, we will have printed the *pre_break* and *post_break* lists; we want to skip over the third list, so that the discretionary data will not appear twice. The following code is performed at the very end of *try_break*.

```
< Update the value of printed_node for symbolic displays 1034 > ≡
  if cur_p = printed_node then
    if cur_p ≠ null then
      if type(cur_p) = disc_node then
        begin t ← replace_count(cur_p);
        while t > 0 do
          begin decr(t); printed_node ← link(printed_node);
          end;
        end
```

This code is used in section 1005.

1035. < Compute the demerits, *d*, from *r* to *cur_p* 1035 > ≡

```
begin d ← line_penalty + b;
if abs(d) ≥ 10000 then d ← 100000000 else d ← d * d;
if pi ≠ 0 then
  if pi > 0 then d ← d + pi * pi
  else if pi > eject_penalty then d ← d - pi * pi;
if (break_type = hyphenated) ∧ (type(r) = hyphenated) then
  if cur_p ≠ null then d ← d + double_hyphen_demerits
  else d ← d + final_hyphen_demerits;
if abs(fit_class - fitness(r)) > 1 then d ← d + adj_demerits;
end
```

This code is used in section 1031.

1036. When an active node disappears, we must delete an adjacent delta node if the active node was at the beginning or the end of the active list, or if it was surrounded by delta nodes. We also must preserve the property that *cur_active_width* represents the length of material from *link*(*prev_r*) to *cur_p*.

```
define combine_two_deltas(#) ≡ mem[prev_r + #].sc ← mem[prev_r + #].sc + mem[r + #].sc
define downdate_width(#) ≡ cur_active_width[#] ← cur_active_width[#] - mem[prev_r + #].sc

< Deactivate node r 1036 > ≡
  link(prev_r) ← link(r); free_node(r, active_node_size);
  if prev_r = active then < Update the active widths, since the first active node has been deleted 1037 >
  else if type(prev_r) = delta_node then
    begin r ← link(prev_r);
    if r = last_active then
      begin do_all_eight(downdate_width); link(prev_prev_r) ← last_active;
      free_node(prev_r, delta_node_size); prev_r ← prev_prev_r;
      end
    else if type(r) = delta_node then
      begin do_all_eight(update_width); do_all_eight(combine_two_deltas); link(prev_r) ← link(r);
      free_node(r, delta_node_size);
      end;
    end
```

This code is used in section 1027.

1037. The following code uses the fact that $\text{type}(\text{last_active}) \neq \text{delta_node}$. If the active list has just become empty, we do not need to update the active_width array, since it will be initialized when an active node is next inserted.

```
define update_active(#[#]) ≡ active_width[#[#]] ← active_width[#[#]] + mem[r + #[#]].sc
⟨ Update the active widths, since the first active node has been deleted 1037 ⟩ ≡
begin r ← link(active);
if type(r) = delta_node then
  begin do_all_eight(update_active); do_all_eight(copy_to_cur_active); link(active) ← link(r);
  free_node(r, delta_node_size);
  end;
end
```

This code is used in section 1036.

1038. Breaking paragraphs into lines, continued. So far we have gotten a little way into the *line_break* routine, having covered its important *try_break* subroutine. Now let's consider the rest of the process.

The main loop of *line_break* traverses the given hlist, starting at *link(temp_head)*, and calls *try_break* at each legal breakpoint. A variable called *auto_breaking* is set to true except within math formulas, since glue nodes are not legal breakpoints when they appear in formulas.

The current node of interest in the hlist is pointed to by *cur_p*. Another variable, *prev_p*, is usually one step behind *cur_p*, but the real meaning of *prev_p* is this: If *type(cur_p) = glue_node* then *cur_p* is a legal breakpoint if and only if *auto_breaking* is true and *prev_p* does not point to a glue node, penalty node, explicit kern node, or math node.

The following declarations provide for a few other local variables that are used in special calculations.

```
⟨ Local variables for line breaking 1038 ⟩ ≡  
q, r, s, prev_s: pointer; { miscellaneous nodes of temporary interest }  
f: internal_font_number; { used when calculating character widths }
```

See also section 1070.

This code is used in section 991.

1039. 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 1039⟩ ≡
  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 1068⟩;
    ⟨Create an active breakpoint representing the beginning of the paragraph 1040⟩;
    cur_p ← link(temp_head); auto_breaking ← true;
    prev_p ← cur_p; { glue at beginning is not a legal breakpoint }
    prev_char_p ← null; prev_legal ← null; rejected_cur_p ← null; try_prev_break ← false;
    before_rejected_cur_p ← false; first_p ← cur_p;
    { to access the first node of paragraph as the first active node has break_node = null }
    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 1042⟩;
    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 1049⟩;
    ⟨Clean up the memory by removing the break nodes 1041⟩;
    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 1853⟩;

```

This code is used in section 991.

1040. 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 1040⟩ ≡
  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 1845⟩;
  do_all_eight(store_background);
  passive ← null; printed_node ← temp_head; pass_number ← 0; font_in_short_display ← null_font

```

This code is used in section 1039.

1041. *<Clean up the memory by removing the break nodes 1041>* ≡
q ← *link(active)*;
while *q* ≠ *last_active* **do**
 begin *cur-p* ← *link(q)*;
 if *type(q)* = *delta-node* **then** *free-node(q, delta-node-size)*
 else *free-node(q, active-node-size)*;
 q ← *cur-p*;
 end;
 q ← *passive*;
 while *q* ≠ *null* **do**
 begin *cur-p* ← *link(q)*; *free-node(q, passive-node-size)*; *q* ← *cur-p*;
 end

This code is used in sections 991 and 1039.

1042. 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 1042 ⟩ ≡
begin if is_char_node(cur_p) then
  ⟨ Advance cur_p to the node following the present string of characters 1043 ⟩;
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 1609 ⟩;
  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) 1044 ⟩;
    if second_pass ∧ auto_breaking then ⟨ Try to hyphenate the following word 1071 ⟩;
  end;
  kern_node: if subtype(cur_p) = explicit then kern_break
  else begin act_width ← act_width + width(cur_p);
    if (pdf_adjust_spacing > 1) ∧ (subtype(cur_p) = normal) then
      begin add_kern_stretch(active_width[7])(cur_p); add_kern_shrink(active_width[8])(cur_p);
    end;
  end;
  ligature_node: begin f ← font(lig_char(cur_p));
    act_width ← act_width + char_width(f)(char_info(f)(character(lig_char(cur_p))));
    if (pdf_adjust_spacing > 1) ∧ check_expand_pars(f) then
      begin prev_char_p ← cur_p; add_char_stretch(active_width[7])(character(lig_char(cur_p)));
      add_char_shrink(active_width[8])(character(lig_char(cur_p)));
    end;
  end;
  disc_node: ⟨ Try to break after a discretionary fragment, then goto done5 1045 ⟩;
  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 1039.

1043. The code that passes over the characters of words in a paragraph is part of TeX's inner loop, so it has been streamlined for speed. We use the fact that '\parfillskip' glue appears at the end of each paragraph; it is therefore unnecessary to check if $\text{link}(\text{cur_p}) = \text{null}$ when cur_p is a character node.

```
(Advance  $\text{cur\_p}$  to the node following the present string of characters 1043) ≡
begin  $\text{prev\_p} \leftarrow \text{cur\_p}$ ;
repeat  $f \leftarrow \text{font}(\text{cur\_p})$ ;  $\text{act\_width} \leftarrow \text{act\_width} + \text{char\_width}(f)(\text{char\_info}(f)(\text{character}(\text{cur\_p})))$ ;
if ( $\text{pdf\_adjust\_spacing} > 1$ )  $\wedge$   $\text{check\_expand\_pars}(f)$  then
begin  $\text{prev\_char\_p} \leftarrow \text{cur\_p}$ ;  $\text{add\_char\_stretch}(\text{active\_width}[7])(\text{character}(\text{cur\_p}))$ ;
 $\text{add\_char\_shrink}(\text{active\_width}[8])(\text{character}(\text{cur\_p}))$ ;
end;
 $\text{cur\_p} \leftarrow \text{link}(\text{cur\_p})$ ;
until  $\neg \text{is\_char\_node}(\text{cur\_p})$ ;
end
```

This code is used in section 1042.

1044. When node cur_p is a glue node, we look at prev_p to see whether or not a breakpoint is legal at cur_p , as explained above.

```
(If node  $\text{cur\_p}$  is a legal breakpoint, call  $\text{try\_break}$ ; then update the active widths by including the glue in
 $\text{glue\_ptr}(\text{cur\_p})$  1044) ≡
if  $\text{auto\_breaking}$  then
begin if  $\text{is\_char\_node}(\text{prev\_p})$  then  $\text{try\_break}(0, \text{unhyphenated})$ 
else if  $\text{precedes\_break}(\text{prev\_p})$  then  $\text{try\_break}(0, \text{unhyphenated})$ 
else if ( $\text{type}(\text{prev\_p}) = \text{kern\_node}$ )  $\wedge$  ( $\text{subtype}(\text{prev\_p}) \neq \text{explicit}$ ) then  $\text{try\_break}(0, \text{unhyphenated})$ ;
end;
 $\text{check\_shrinkage}(\text{glue\_ptr}(\text{cur\_p}))$ ;  $q \leftarrow \text{glue\_ptr}(\text{cur\_p})$ ;  $\text{act\_width} \leftarrow \text{act\_width} + \text{width}(q)$ ;
 $\text{active\_width}[2 + \text{stretch\_order}(q)] \leftarrow \text{active\_width}[2 + \text{stretch\_order}(q)] + \text{stretch}(q)$ ;
 $\text{active\_width}[6] \leftarrow \text{active\_width}[6] + \text{shrink}(q)$ 
```

This code is used in section 1042.

1045. The following code knows that discretionary texts contain only character nodes, kern nodes, box nodes, rule nodes, and ligature nodes.

```
(Try to break after a discretionary fragment, then goto  $\text{done5}$  1045) ≡
begin  $s \leftarrow \text{pre\_break}(\text{cur\_p})$ ;  $\text{do\_one\_seven\_eight}(\text{reset\_disc\_width})$ ;
if  $s = \text{null}$  then  $\text{try\_break}(\text{ex\_hyphen\_penalty}, \text{hyphenated})$ 
else begin repeat (Add the width of node  $s$  to  $\text{disc\_width}$  1046);
 $s \leftarrow \text{link}(s)$ ;
until  $s = \text{null}$ ;
 $\text{do\_one\_seven\_eight}(\text{add\_disc\_width\_to\_active\_width})$ ;  $\text{try\_break}(\text{hyphen\_penalty}, \text{hyphenated})$ ;
 $\text{do\_one\_seven\_eight}(\text{sub\_disc\_width\_from\_active\_width})$ ;
end;
 $r \leftarrow \text{replace\_count}(\text{cur\_p})$ ;  $s \leftarrow \text{link}(\text{cur\_p})$ ;
while  $r > 0$  do
begin (Add the width of node  $s$  to  $\text{act\_width}$  1047);
 $\text{decr}(r)$ ;  $s \leftarrow \text{link}(s)$ ;
end;
 $\text{prev\_p} \leftarrow \text{cur\_p}$ ;  $\text{cur\_p} \leftarrow s$ ; goto  $\text{done5}$ ;
end
```

This code is used in section 1042.

1046. \langle Add the width of node s to $disc_width$ \rangle \equiv

```

if is_char_node( $s$ ) then
  begin  $f \leftarrow font(s)$ ;  $disc\_width[1] \leftarrow disc\_width[1] + char\_width(f)(char\_info(f)(character(s)))$ ;
  if ( $pdf\_adjust\_spacing > 1$ )  $\wedge$  check_expand_pars( $f$ ) then
    begin  $prev\_char\_p \leftarrow s$ ; add_char_stretch( $disc\_width[7]$ )( $character(s)$ );
    add_char_shrink( $disc\_width[8]$ )( $character(s)$ );
    end;
  end
else case type( $s$ ) of
  ligature_node: begin  $f \leftarrow font(lig\_char(s))$ ;
   $disc\_width[1] \leftarrow disc\_width[1] + char\_width(f)(char\_info(f)(character(lig\_char(s))))$ ;
  if ( $pdf\_adjust\_spacing > 1$ )  $\wedge$  check_expand_pars( $f$ ) then
    begin  $prev\_char\_p \leftarrow s$ ; add_char_stretch( $disc\_width[7]$ )( $character(lig\_char(s))$ );
    add_char_shrink( $disc\_width[8]$ )( $character(lig\_char(s))$ );
    end;
  end;
  hlist_node, vlist_node, rule_node, kern_node: begin  $disc\_width[1] \leftarrow disc\_width[1] + width(s)$ ;
  if (type( $s$ ) = kern_node)  $\wedge$  ( $pdf\_adjust\_spacing > 1$ )  $\wedge$  (subtype( $s$ ) = normal) then
    begin add_kern_stretch( $disc\_width[7]$ )( $s$ ); add_kern_shrink( $disc\_width[8]$ )( $s$ );
    end;
  end;
othercases confusion("disc3")
endcases

```

This code is used in section 1045.

1047. \langle Add the width of node s to act_width \rangle \equiv

```

if is_char_node( $s$ ) then
  begin  $f \leftarrow font(s)$ ;  $act\_width \leftarrow act\_width + char\_width(f)(char\_info(f)(character(s)))$ ;
  if ( $pdf\_adjust\_spacing > 1$ )  $\wedge$  check_expand_pars( $f$ ) then
    begin  $prev\_char\_p \leftarrow s$ ; add_char_stretch( $active\_width[7]$ )( $character(s)$ );
    add_char_shrink( $active\_width[8]$ )( $character(s)$ );
    end;
  end
else case type( $s$ ) of
  ligature_node: begin  $f \leftarrow font(lig\_char(s))$ ;
   $act\_width \leftarrow act\_width + char\_width(f)(char\_info(f)(character(lig\_char(s))))$ ;
  if ( $pdf\_adjust\_spacing > 1$ )  $\wedge$  check_expand_pars( $f$ ) then
    begin  $prev\_char\_p \leftarrow s$ ; add_char_stretch( $active\_width[7]$ )( $character(lig\_char(s))$ );
    add_char_shrink( $active\_width[8]$ )( $character(lig\_char(s))$ );
    end;
  end;
  hlist_node, vlist_node, rule_node, kern_node: begin  $act\_width \leftarrow act\_width + width(s)$ ;
  if (type( $s$ ) = kern_node)  $\wedge$  ( $pdf\_adjust\_spacing > 1$ )  $\wedge$  (subtype( $s$ ) = normal) then
    begin add_kern_stretch( $active\_width[7]$ )( $s$ ); add_kern_shrink( $active\_width[8]$ )( $s$ );
    end;
  end;
othercases confusion("disc4")
endcases

```

This code is used in section 1045.

1048. The forced line break at the paragraph's end will reduce the list of breakpoints so that all active nodes represent breaks at $cur_p = null$. On the first pass, we insist on finding an active node that has the correct "looseness." On the final pass, there will be at least one active node, and we will match the desired looseness as well as we can.

The global variable *best_bet* will be set to the active node for the best way to break the paragraph, and a few other variables are used to help determine what is best.

```
<Global variables 13> +≡
best_bet: pointer; { use this passive node and its predecessors }
fewest_demerits: integer; { the demerits associated with best_bet }
best_line: halfword; { line number following the last line of the new paragraph }
actual_looseness: integer; { the difference between line_number(best_bet) and the optimum best_line }
line_diff: integer; { the difference between the current line number and the optimum best_line }
```

1049. *<Try the final line break at the end of the paragraph, and goto done if the desired breakpoints have been found 1049>* ≡

```
begin try_break(eject_penalty, hyphenated);
if link(active) ≠ last_active then
  begin <Find an active node with fewest demerits 1050>;
    if looseness = 0 then goto done;
    <Find the best active node for the desired looseness 1051>;
    if (actual_looseness = looseness) ∨ final_pass then goto done;
    end;
  end
```

This code is used in section 1039.

1050. *<Find an active node with fewest demerits 1050>* ≡

```
r ← link(active); fewest_demerits ← awful_bad;
repeat if type(r) ≠ delta_node then
  if total_demerits(r) < fewest_demerits then
    begin fewest_demerits ← total_demerits(r); best_bet ← r;
    end;
  r ← link(r);
until r = last_active;
best_line ← line_number(best_bet)
```

This code is used in section 1049.

1051. The adjustment for a desired looseness is a slightly more complicated version of the loop just considered. Note that if a paragraph is broken into segments by displayed equations, each segment will be subject to the looseness calculation, independently of the other segments.

```
<Find the best active node for the desired looseness 1051> ≡
begin  $r \leftarrow \text{link}(\text{active})$ ;  $\text{actual\_looseness} \leftarrow 0$ ;
repeat if  $\text{type}(r) \neq \text{delta\_node}$  then
  begin  $\text{line\_diff} \leftarrow \text{line\_number}(r) - \text{best\_line}$ ;
  if  $((\text{line\_diff} < \text{actual\_looseness}) \wedge (\text{looseness} \leq \text{line\_diff})) \vee$ 
     $((\text{line\_diff} > \text{actual\_looseness}) \wedge (\text{looseness} \geq \text{line\_diff}))$  then
      begin  $\text{best\_bet} \leftarrow r$ ;  $\text{actual\_looseness} \leftarrow \text{line\_diff}$ ;  $\text{fewest\_demerits} \leftarrow \text{total\_demerits}(r)$ ;
      end;
  else if  $(\text{line\_diff} = \text{actual\_looseness}) \wedge (\text{total\_demerits}(r) < \text{fewest\_demerits})$  then
    begin  $\text{best\_bet} \leftarrow r$ ;  $\text{fewest\_demerits} \leftarrow \text{total\_demerits}(r)$ ;
    end;
  end;
   $r \leftarrow \text{link}(r)$ ;
until  $r = \text{last\_active}$ ;
 $\text{best\_line} \leftarrow \text{line\_number}(\text{best\_bet})$ ;
end
```

This code is used in section 1049.

1052. 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 1052> ≡
post_line_break( $d$ )
```

This code is used in section 991.

1053. 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 1002 ⟩ +≡
procedure post_line_break(d : boolean);
label done, done1;
var q, r, s: pointer; { temporary registers for list manipulation }
p, k: pointer; w: scaled; glue_break: boolean; { was a break at glue? }
ptmp: pointer; 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 1054 ⟩;
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 1056 ⟩;
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 1055 ⟩;
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;
```

1054. The job of reversing links in a list is conveniently regarded as the job of taking items off one stack and putting them on another. In this case we take them off a stack pointed to by *q* and having *prev_break* fields; we put them on a stack pointed to by *cur_p* and having *next_break* fields. Node *r* is the passive node being moved from stack to stack.

```
⟨ Reverse the links of the relevant passive nodes, setting cur_p to the first breakpoint 1054 ⟩ ≡
q ← break_node(best_bet); cur_p ← null;
repeat r ← q; q ← prev_break(q); next_break(r) ← cur_p; cur_p ← r;
until q = null
```

This code is used in section 1053.

1055. 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 1055> ≡
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 1708>;
  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 1053.

1056. 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 1056> ≡
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 1707>;
<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 1057>;
if TeXXeT_en then <Insert LR nodes at the end of the current line 1709>;
<Put the \leftskip glue at the left and detach this line 1063>;
<Call the packaging subroutine, setting just_box to the justified box 1066>;
<Append the new box to the current vertical list, followed by the list of special nodes taken out of the
  box by the packager 1065>;
<Append a penalty node, if a nonzero penalty is appropriate 1067>
```

This code is used in section 1053.

1057. 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` 1057 ⟩ ≡

```

q ← cur_break(cur_p); disc_break ← false; post_disc_break ← false; glue_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); glue_break ← true; goto done;
    end
  else begin if type(q) = disc_node then
    ⟨ Change discretionary to compulsory and set disc_break ← true 1058 ⟩
    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 1708 ⟩;
      end;
    end
  else begin q ← temp_head;
    while link(q) ≠ null do q ← link(q);
    end;
done: { at this point q is the rightmost breakpoint; the only exception is the case of a discretionary break with non-empty pre_break, then q has been changed to the last node of the pre_break list }
if pdf_protrude_chars > 0 then
  begin
    if disc_break ∧ (is_char_node(q) ∨ (type(q) ≠ disc_node))
      { q has been reset to the last node of pre_break }
    then
      begin p ← q; ptmp ← p;
      end
    else begin p ← prev_rightmost(link(temp_head), q); { get link(p) = q }
      ptmp ← p; p ← find_protchar_right(link(temp_head), p);
    end; @{short_display_n(p, 1); print_ln; @} w ← right_pw(p);
    if w ≠ 0 then { we have found a marginal kern, append it after ptmp }
      begin k ← new_margin_kern(−w, last_rightmost_char, right_side); link(k) ← link(ptmp);
      link(ptmp) ← k;
      if (ptmp = q) then q ← link(q);
      end;
    end; { if q was not a breakpoint at glue and has been reset to rightskip then we append rightskip after q now}
    if ¬glue_break then
      begin ⟨ Put the \rightskip glue after node q 1062 ⟩;
      end;
  end;

```

This code is used in section 1056.

1058. ⟨ Change discretionary to compulsory and set `disc_break` ← true 1058 ⟩ ≡

```

begin t ← replace_count(q);
⟨ Destroy the t nodes following q, and make r point to the following node 1059 ⟩;
if post_break(q) ≠ null then ⟨ Transplant the post-break list 1060 ⟩;
if pre_break(q) ≠ null then ⟨ Transplant the pre-break list 1061 ⟩;
link(q) ← r; disc_break ← true;
end

```

This code is used in section 1057.

1059. \langle Destroy the t nodes following q , and make r point to the following node 1059 $\rangle \equiv$

```

if  $t = 0$  then  $r \leftarrow link(q)$ 
else begin  $r \leftarrow q$ ;
  while  $t > 1$  do
    begin  $r \leftarrow link(r)$ ;  $decr(t)$ ;
    end;
 $s \leftarrow link(r)$ ;  $r \leftarrow link(s)$ ;  $link(s) \leftarrow null$ ;  $flush\_node\_list(link(q))$ ;  $replace\_count(q) \leftarrow 0$ ;
end

```

This code is used in section 1058.

1060. We move the post-break list from inside node q to the main list by reattaching it just before the present node r , then resetting r .

\langle Transplant the post-break list 1060 $\rangle \equiv$

```

begin  $s \leftarrow post\_break(q)$ ;
while  $link(s) \neq null$  do  $s \leftarrow link(s)$ ;
 $link(s) \leftarrow r$ ;  $r \leftarrow post\_break(q)$ ;  $post\_break(q) \leftarrow null$ ;  $post\_disc\_break \leftarrow true$ ;
end

```

This code is used in section 1058.

1061. We move the pre-break list from inside node q to the main list by reattaching it just after the present node q , then resetting q .

\langle Transplant the pre-break list 1061 $\rangle \equiv$

```

begin  $s \leftarrow pre\_break(q)$ ;  $link(q) \leftarrow s$ ;
while  $link(s) \neq null$  do  $s \leftarrow link(s)$ ;
 $pre\_break(q) \leftarrow null$ ;  $q \leftarrow s$ ;
end

```

This code is used in section 1058.

1062. \langle Put the \rightskip glue after node q 1062 $\rangle \equiv$

```

 $r \leftarrow new\_param\_glue(right\_skip\_code)$ ;  $link(r) \leftarrow link(q)$ ;  $link(q) \leftarrow r$ ;  $q \leftarrow r$ 

```

This code is used in section 1057.

1063. The following code begins with q at the end of the list to be justified. It ends with q at the beginning of that list, and with $link(temp_head)$ pointing to the remainder of the paragraph, if any.

\langle Put the \leftskip glue at the left and detach this line 1063 $\rangle \equiv$

```

 $r \leftarrow link(q)$ ;  $link(q) \leftarrow null$ ;  $q \leftarrow link(temp\_head)$ ;  $link(temp\_head) \leftarrow r$ ;
  { at this point  $q$  is the leftmost node; all discardable nodes have been discarded }
if  $pdf\_protrude\_chars > 0$  then
  begin  $p \leftarrow q$ ;  $p \leftarrow find\_protchar\_left(p, false)$ ; { no more discardables }
   $w \leftarrow left\_pw(p)$ ;
  if  $w \neq 0$  then
    begin  $k \leftarrow new\_margin\_kern(-w, last\_leftmost\_char, left\_side)$ ;  $link(k) \leftarrow q$ ;  $q \leftarrow k$ ;
    end;
  end;
if  $left\_skip \neq zero\_glue$  then
  begin  $r \leftarrow new\_param\_glue(left\_skip\_code)$ ;  $link(r) \leftarrow q$ ;  $q \leftarrow r$ ;
  end

```

This code is used in section 1056.

1064. ⟨ Initialize table entries (done by INITEX only) 182 ⟩ \equiv

```
pdf_ignored_dimen  $\leftarrow$  ignore_depth; pdf_each_line_height  $\leftarrow$  pdf_ignored_dimen;
pdf_each_line_depth  $\leftarrow$  pdf_ignored_dimen; pdf_first_line_height  $\leftarrow$  pdf_ignored_dimen;
pdf_last_line_depth  $\leftarrow$  pdf_ignored_dimen;
```

1065. ⟨ Append the new box to the current vertical list, followed by the list of special nodes taken out of the box by the packager 1065 ⟩ \equiv

```
if pdf_each_line_height  $\neq$  pdf_ignored_dimen then height(just_box)  $\leftarrow$  pdf_each_line_height;
if pdf_each_line_depth  $\neq$  pdf_ignored_dimen then depth(just_box)  $\leftarrow$  pdf_each_line_depth;
if (pdf_first_line_height  $\neq$  pdf_ignored_dimen)  $\wedge$  (cur_line = prev_graf + 1) then
    height(just_box)  $\leftarrow$  pdf_first_line_height;
if (pdf_last_line_depth  $\neq$  pdf_ignored_dimen)  $\wedge$  (cur_line + 1 = best_line) then
    depth(just_box)  $\leftarrow$  pdf_last_line_depth;
if pre_adjust_head  $\neq$  pre_adjust_tail then append_list(pre_adjust_head)(pre_adjust_tail);
pre_adjust_tail  $\leftarrow$  null; append_to_vlist(just_box);
if adjust_head  $\neq$  adjust_tail then append_list(adjust_head)(adjust_tail);
adjust_tail  $\leftarrow$  null
```

This code is used in section 1056.

1066. Now q points to the hlist that represents the current line of the paragraph. We need to compute the appropriate line width, pack the line into a box of this size, and shift the box by the appropriate amount of indentation.

⟨ Call the packaging subroutine, setting just_box to the justified box 1066 ⟩ \equiv

```
if cur_line > last_special_line then
    begin cur_width  $\leftarrow$  second_width; cur_indent  $\leftarrow$  second_indent;
    end
else if par_shape_ptr = null then
    begin cur_width  $\leftarrow$  first_width; cur_indent  $\leftarrow$  first_indent;
    end
else begin cur_width  $\leftarrow$  mem[par_shape_ptr + 2 * cur_line].sc;
        cur_indent  $\leftarrow$  mem[par_shape_ptr + 2 * cur_line - 1].sc;
        end;
adjust_tail  $\leftarrow$  adjust_head; pre_adjust_tail  $\leftarrow$  pre_adjust_head;
if pdf_adjust_spacing > 0 then just_box  $\leftarrow$  hpack(q, cur_width, cal_expand_ratio)
else just_box  $\leftarrow$  hpack(q, cur_width, exactly);
shift_amount(just_box)  $\leftarrow$  cur_indent
```

This code is used in section 1056.

1067. 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 1067 ⟩ ≡

```

if cur_line + 1 ≠ best_line then
begin q ← inter_line_penalties_ptr;
if q ≠ null then
begin r ← cur_line;
if r > penalty(q) then r ← penalty(q);
pen ← penalty(q + r);
end
else pen ← inter_line_penalty;
q ← club_penalties_ptr;
if q ≠ null then
begin r ← cur_line - prev_graf;
if r > penalty(q) then r ← penalty(q);
pen ← pen + penalty(q + r);
end
else if cur_line = prev_graf + 1 then pen ← pen + club_penalty;
if d then q ← display_widow_penalties_ptr
else q ← widow_penalties_ptr;
if q ≠ null then
begin r ← best_line - cur_line - 1;
if r > penalty(q) then r ← penalty(q);
pen ← pen + penalty(q + r);
end
else if cur_line + 2 = best_line then
if d then pen ← pen + display_widow_penalty
else pen ← pen + widow_penalty;
if disc_break then pen ← pen + broken_penalty;
if pen ≠ 0 then
begin r ← new_penalty(pen); link(tail) ← r; tail ← r;
end;
end

```

This code is used in section 1056.

1068. Pre-hyphenation. When the line-breaking routine is unable to find a feasible sequence of break-points, 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 $p_0 p_1 \dots p_m$ where p_0 is a glue node, $p_1 \dots p_{m-1}$ 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 $p_1 \dots p_{m-1}$ 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 $p_1 \dots p_{m-1}$ 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 $c_1 \dots c_n$ 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 $c_1 \dots c_n$ that are generated by nodes $p_a \dots p_b$, where $1 \leq a \leq b + 1 \leq m$. If $n \geq l_hyf + r_hyf$, 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 $c_1 \dots c_n$ is found; then potential positions for hyphens are determined by referring to hyphenation tables; and finally, the nodes $p_a \dots p_b$ 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 1068 > ≡
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 1039.

1069. The letters $c_1 \dots c_n$ that are candidates for hyphenation are placed into an array called hc ; the number n is placed into hn ; pointers to nodes p_{a-1} and p_b in the description above are placed into variables ha and hb ; and the font number is placed into hf .

```
<Global variables 13> +≡
  hc: array [0 .. 65] of 0 .. 256; { word to be hyphenated }
  hn: 0 .. 64; { the number of positions occupied in hc; not always a small_number }
  ha, hb: pointer; { nodes ha .. hb should be replaced by the hyphenated result }
  hf: internal_font_number; { font number of the letters in hc }
  hu: array [0 .. 63] of 0 .. 256; { like hc, before conversion to lowercase }
  hyf_char: integer; { hyphen character of the relevant font }
  cur_lang, init_cur_lang: ASCII_code; { current hyphenation table of interest }
  l_hyf, r_hyf, init_l_hyf, init_r_hyf: integer; { limits on fragment sizes }
  hyf_bchar: halfword; { boundary character after  $c_n$  }
```

1070. Hyphenation routines need a few more local variables.

```
<Local variables for line breaking 1038> +≡
  j: small_number; { an index into hc or hu }
  c: 0 .. 255; { character being considered for hyphenation }
```

1071. When the following code is activated, the *line_break* procedure is in its second pass, and *cur_p* points to a glue node.

```
<Try to hyphenate the following word 1071> ≡
  begin prev_s ← cur_p; s ← link(prev_s);
  if s ≠ null then
    begin {Skip to node ha, or goto done1 if no hyphenation should be attempted 1073};
    if l_hyf + r_hyf > 63 then goto done1;
    {Skip to node hb, putting letters into hu and hc 1074};
    {Check that the nodes following hb permit hyphenation and that at least l_hyf + r_hyf letters have
     been found, otherwise goto done1 1076};
    hyphenate;
  end;
done1: end
```

This code is used in section 1042.

1072. { Declare subprocedures for *line_break* 1002 } +≡

{ Declare the function called *reconstitute* 1083 }

```
procedure hyphenate;
  label common_ending, done, found, found1, found2, not_found, exit;
  var { Local variables for hyphenation 1078 }
  begin {Find hyphen locations for the word in hc, or return 1100};
  {If no hyphens were found, return 1079};
  {Replace nodes ha .. hb by a sequence of nodes that includes the discretionary hyphens 1080};
exit: end;
```

1073. 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 1073 > ≡
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 1610 >;
    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 1071.

1074. The word to be hyphenated is now moved to the *hu* and *hc* arrays.

```
< Skip to node hb, putting letters into hu and hc 1074 > ≡
  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 1075 >
  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 1071.

1075. 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 1075 > ≡
begin if font(lig_char(s)) ≠ hf then goto done3;
   j ← hn; q ← lig_ptr(s); if q > null then hyf_bchar ← character(q);
   while q > null do
      begin c ← qo(character(q)); set_lc_code(c);
         if hc[0] = 0 then goto done3;
         if j = 63 then goto done3;
         incr(j); hu[j] ← c; hc[j] ← hc[0];
         q ← link(q);
         end;
      hb ← s; hn ← j;
      if odd(subtype(s)) then hyf_bchar ← font_bchar[hf] else hyf_bchar ← non_char;
      end
```

This code is used in section 1074.

1076. <Check that the nodes following *hb* permit hyphenation and that at least $l_hyf + r_hyf$ letters have been found, otherwise **goto** *done1* 1076 > ≡

```
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) ≠ 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 ← link(s);
end;
```

done4:

This code is used in section 1071.

1077. Post-hyphenation. If a hyphen may be inserted between $hc[j]$ and $hc[j + 1]$, the hyphenation procedure will set $hyf[j]$ to some small odd number. But before we look at TeX's hyphenation procedure, which is independent of the rest of the line-breaking algorithm, let us consider what we will do with the hyphens it finds, since it is better to work on this part of the program before forgetting what ha and hb , etc., are all about.

```
<Global variables 13> +≡
hyf: array [0..64] of 0..9; { odd values indicate discretionary hyphens }
init_list: pointer; { list of punctuation characters preceding the word }
init_lig: boolean; { does init_list represent a ligature? }
init_lft: boolean; { if so, did the ligature involve a left boundary? }
```

1078. < Local variables for hyphenation 1078 > ≡

```
i, j, l: 0 .. 65; { indices into hc or hu }
q, r, s: pointer; { temporary registers for list manipulation }
bchar: halfword; { boundary character of hyphenated word, or non_char }
```

See also sections 1089, 1099, and 1106.

This code is used in section 1072.

1079. TeX will never insert a hyphen that has fewer than \leftlefthyphenmin letters before it or fewer than \righthyphenmin after it; hence, a short word has comparatively little chance of being hyphenated. If no hyphens have been found, we can save time by not having to make any changes to the paragraph.

```
<If no hyphens were found, return 1079> ≡
  for j ← l_hyf to hn – r_hyf do
    if odd(hyf[j]) then goto found1;
  return;
found1:
```

This code is used in section 1072.

1080. If hyphens are in fact going to be inserted, TeX first deletes the subsequence of nodes between *ha* and *hb*. An attempt is made to preserve the effect that implicit boundary characters and punctuation marks had on ligatures inside the hyphenated word, by storing a left boundary or preceding character in *hu*[0] and by storing a possible right boundary in *bchar*. We set *j* \leftarrow 0 if *hu*[0] is to be part of the reconstruction; otherwise *j* \leftarrow 1. The variable *s* will point to the tail of the current hlist, and *q* will point to the node following *hb*, so that things can be hooked up after we reconstitute the hyphenated word.

```
< Replace nodes ha .. hb by a sequence of nodes that includes the discretionary hyphens 1080 > ≡
  q ← link(hb); link(hb) ← null; r ← link(ha); link(ha) ← null; bchar ← hyf_bchar;
  if is_char_node(ha) then
    if font(ha) ≠ hf then goto found2
    else begin init_list ← ha; init_lig ← false; hu[0] ← qo(character(ha));
      end
    else if type(ha) = ligature_node then
      if font(lig_char(ha)) ≠ hf then goto found2
      else begin init_list ← lig_ptr(ha); init_lig ← true; init_lft ← (subtype(ha) > 1);
        hu[0] ← qo(character(lig_char(ha)));
        if init_list = null then
          if init_lft then
            begin hu[0] ← 256; init_lig ← false;
            end; { in this case a ligature will be reconstructed from scratch }
          free_node(ha, small_node_size);
          end
        else begin { no punctuation found; look for left boundary }
          if ¬is_char_node(r) then
            if type(r) = ligature_node then
              if subtype(r) > 1 then goto found2;
              j ← 1; s ← ha; init_list ← null; goto common_ending;
            end;
          s ← cur_p; { we have cur_p ≠ ha because type(cur_p) = glue_node }
          while link(s) ≠ ha do s ← link(s);
          j ← 0; goto common_ending;
        found2: s ← ha; j ← 0; hu[0] ← 256; init_lig ← false; init_list ← null;
        common_ending: flush_node_list(r);
        < Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 1090 >;
        flush_list(init_list)
```

This code is used in section 1072.

1081. We must now face the fact that the battle is not over, even though the hyphens have been found: The process of reconstituting a word can be nontrivial because ligatures might change when a hyphen is present. The *TeXbook* discusses the difficulties of the word “difficult”, and the discretionary material surrounding a hyphen can be considerably more complex than that. Suppose abcdef is a word in a font for which the only ligatures are bc, cd, de, and ef. If this word permits hyphenation between b and c, the two patterns with and without hyphenation are a b - cd ef and a bc de f. Thus the insertion of a hyphen might cause effects to ripple arbitrarily far into the rest of the word. A further complication arises if additional hyphens appear together with such rippling, e.g., if the word in the example just given could also be hyphenated between c and d; TeX avoids this by simply ignoring the additional hyphens in such weird cases.

Still further complications arise in the presence of ligatures that do not delete the original characters. When punctuation precedes the word being hyphenated, TeX’s method is not perfect under all possible scenarios, because punctuation marks and letters can propagate information back and forth. For example, suppose the original pre-hyphenation pair *a changes to *y via a |=: ligature, which changes to xy via a =: | ligature; if *p_{a-1}* = x and *p_a* = y, the reconstitution procedure isn’t smart enough to obtain xy again. In such cases the font designer should include a ligature that goes from xa to xy.

1082. The processing is facilitated by a subroutine called *reconstitute*. Given a string of characters $x_j \dots x_n$, there is a smallest index $m \geq j$ such that the “translation” of $x_j \dots x_n$ by ligatures and kerning has the form $y_1 \dots y_t$ followed by the translation of $x_{m+1} \dots x_n$, where $y_1 \dots y_t$ is some nonempty sequence of character, ligature, and kern nodes. We call $x_j \dots x_m$ a “cut prefix” of $x_j \dots x_n$. For example, if $x_1x_2x_3 = \text{fly}$, and if the font contains ‘fl’ as a ligature and a kern between ‘fl’ and ‘y’, then $m = 2$, $t = 2$, and y_1 will be a ligature node for ‘fl’ followed by an appropriate kern node y_2 . In the most common case, x_j forms no ligature with x_{j+1} and we simply have $m = j$, $y_1 = x_j$. If $m < n$ we can repeat the procedure on $x_{m+1} \dots x_n$ until the entire translation has been found.

The *reconstitute* function returns the integer m and puts the nodes $y_1 \dots y_t$ into a linked list starting at *link(hold_head)*, getting the input $x_j \dots x_n$ from the *hu* array. If $x_j = 256$, we consider x_j to be an implicit left boundary character; in this case j must be strictly less than n . There is a parameter *bchar*, which is either 256 or an implicit right boundary character assumed to be present just following x_n . (The value *hu*[$n + 1$] is never explicitly examined, but the algorithm imagines that *bchar* is there.)

If there exists an index k in the range $j \leq k \leq m$ such that *hyf*[k] is odd and such that the result of *reconstitute* would have been different if x_{k+1} had been *hchar*, then *reconstitute* sets *hyphen_passed* to the smallest such k . Otherwise it sets *hyphen_passed* to zero.

A special convention is used in the case $j = 0$: Then we assume that the translation of *hu*[0] appears in a special list of charnodes starting at *init_list*; moreover, if *init_lig* is true, then *hu*[0] will be a ligature character, involving a left boundary if *init_lft* is true. This facility is provided for cases when a hyphenated word is preceded by punctuation (like single or double quotes) that might affect the translation of the beginning of the word.

(Global variables 13) +≡
hyphen_passed: small_number; { first hyphen in a ligature, if any }

1083. *(Declare the function called *reconstitute* 1083) ≡*
function *reconstitute(j, n : small_number; bchar, hchar : halfword): small_number;*
label *continue, done;*
var *p: pointer; { temporary register for list manipulation }*
t: pointer; { a node being appended to }
q: four_quarters; { character information or a lig/kern instruction }
cur_rh: halfword; { hyphen character for ligature testing }
test_char: halfword; { hyphen or other character for ligature testing }
w: scaled; { amount of kerning }
k: font_index; { position of current lig/kern instruction }
begin *hyphen_passed* $\leftarrow 0$; *t* \leftarrow *hold_head*; *w* $\leftarrow 0$; *link(hold_head)* \leftarrow null;
{ at this point ligature_present = lft_hit = rt_hit = false }
{ Set up data structures with the cursor following position j 1085};
continue: *(If there's a ligature or kern at the cursor position, update the data structures, possibly advancing j; continue until the cursor moves 1086);*
{ Append a ligature and/or kern to the translation; goto continue if the stack of inserted ligatures is nonempty 1087};
reconstitute $\leftarrow j;
end;$

This code is used in section 1072.

1084. The reconstitution procedure shares many of the global data structures by which T_EX has processed the words before they were hyphenated. There is an implied “cursor” between characters *cur_l* and *cur_r*; these characters will be tested for possible ligature activity. If *ligature_present* then *cur_l* is a ligature character formed from the original characters following *cur_q* in the current translation list. There is a “ligature stack” between the cursor and character *j* + 1, consisting of pseudo-ligature nodes linked together by their *link* fields. This stack is normally empty unless a ligature command has created a new character that will need to be processed later. A pseudo-ligature is a special node having a *character* field that represents a potential ligature and a *lig_ptr* field that points to a *char_node* or is *null*. We have

$$cur_r = \begin{cases} character(lig_stack), & \text{if } lig_stack > null; \\ qi(hu[j+1]), & \text{if } lig_stack = null \text{ and } j < n; \\ bchar, & \text{if } lig_stack = null \text{ and } j = n. \end{cases}$$

(Global variables 13) +≡
cur_l, cur_r: halfword; { characters before and after the cursor }
cur_q: pointer; { where a ligature should be detached }
lig_stack: pointer; { unfinished business to the right of the cursor }
ligature_present: boolean; { should a ligature node be made for *cur_l*? }
lft_hit, rt_hit: boolean; { did we hit a ligature with a boundary character? }

1085. define *append_charnode_to_t*(#) ≡
begin *link*(*t*) ← *get_avail*; *t* ← *link*(*t*); *font*(*t*) ← *hf*; *character*(*t*) ← #;
end
define *set_cur_r* ≡
begin if *j* < *n* **then** *cur_r* ← *qi*(*hu*[*j* + 1]) **else** *cur_r* ← *bchar*;
if *odd*(*hyf*[*j*]) **then** *cur_rh* ← *hchar* **else** *cur_rh* ← *non_char*;
end
*(Set up data structures with the cursor following position *j* 1085) ≡*
cur_l ← *qi*(*hu*[*j*]); *cur_q* ← *t*;
if *j* = 0 **then**
begin *ligature_present* ← *init_lig*; *p* ← *init_list*;
if *ligature_present* **then** *lft_hit* ← *init_lft*;
while *p* > *null* **do**
begin *append_charnode_to_t*(*character*(*p*)); *p* ← *link*(*p*);
end;
end
else if *cur_l* < *non_char* **then** *append_charnode_to_t*(*cur_l*);
lig_stack ← *null*; *set_cur_r*

This code is used in section 1083.

1086. We may want to look at the lig/kern program twice, once for a hyphen and once for a normal letter. (The hyphen might appear after the letter in the program, so we'd better not try to look for both at once.) {If there's a ligature or kern at the cursor position, update the data structures, possibly advancing j ;

continue until the cursor moves 1086 } ≡

```

if cur_l = non_char then
  begin k ← bchar_label[hf];
  if k = non_address then goto done else q ← font_info[k].qqqq;
  end
else begin q ← char_info(hf)(cur_l);
  if char_tag(q) ≠ lig_tag then goto done;
  k ← lig_kern_start(hf)(q); q ← font_info[k].qqqq;
  if skip_byte(q) > stop_flag then
    begin k ← lig_kern_restart(hf)(q); q ← font_info[k].qqqq;
    end;
  end; { now k is the starting address of the lig/kern program }
if cur_rh < non_char then test_char ← cur_rh else test_char ← cur_r;
loop begin if next_char(q) = test_char then
  if skip_byte(q) ≤ stop_flag then
    if cur_rh < non_char then
      begin hyphen_passed ← j; hchar ← non_char; cur_rh ← non_char; goto continue;
      end
    else begin if hchar < non_char then
      if odd(hyf[j]) then
        begin hyphen_passed ← j; hchar ← non_char;
        end;
      if op_byte(q) < kern_flag then
        {Carry out a ligature replacement, updating the cursor structure and possibly advancing j;
        goto continue if the cursor doesn't advance, otherwise goto done 1088};
      w ← char_kern(hf)(q); goto done; { this kern will be inserted below}
      end;
    end;
  if skip_byte(q) ≥ stop_flag then
    if cur_rh = non_char then goto done
    else begin cur_rh ← non_char; goto continue;
    end;
  k ← k + qo(skip_byte(q)) + 1; q ← font_info[k].qqqq;
end;
done:
```

This code is used in section 1083.

```

1087. define wrap_lig(#) ≡
  if ligature_present then
    begin p ← new_ligature(hf, cur_l, link(cur_q));
    if lft_hit then
      begin subtype(p) ← 2; lft_hit ← false;
      end;
    if # then
      if lig_stack = null then
        begin incr(subtype(p)); rt_hit ← false;
        end;
      link(cur_q) ← p; t ← p; ligature_present ← false;
      end
define pop_lig_stack ≡
  begin if lig_ptr(lig_stack) > null then
    begin link(t) ← lig_ptr(lig_stack); { this is a charnode for hu[j + 1]}
    t ← link(t); incr(j);
    end;
    p ← lig_stack; lig_stack ← link(p); free_node(p, small_node_size);
    if lig_stack = null then set_cur_r else cur_r ← character(lig_stack);
    end { if lig_stack isn't null we have cur_rh = non_char }
{ Append a ligature and/or kern to the translation; goto continue if the stack of inserted ligatures is
  nonempty 1087 } ≡
wrap_lig(rt_hit);
if w ≠ 0 then
  begin link(t) ← new_kern(w); t ← link(t); w ← 0;
  end;
if lig_stack > null then
  begin cur_q ← t; cur_l ← character(lig_stack); ligature_present ← true; pop_lig_stack; goto continue;
  end

```

This code is used in section 1083.

1088. ⟨Carry out a ligature replacement, updating the cursor structure and possibly advancing *j*; **goto continue** if the cursor doesn't advance, otherwise **goto done** 1088⟩ ≡

```

begin if cur_l = non_char then lft_hit ← true;
if j = n then
  if lig_stack = null then rt_hit ← true;
  check_interrupt; { allow a way out in case there's an infinite ligature loop }
  case op_byte(q) of
    qi(1), qi(5): begin cur_l ← rem_byte(q); { =:|, =:|> }
      ligature_present ← true;
    end;
    qi(2), qi(6): begin cur_r ← rem_byte(q); { |=:, |=:> }
      if lig_stack > null then character(lig_stack) ← cur_r
      else begin lig_stack ← new_lig_item(cur_r);
        if j = n then bchar ← non_char
        else begin p ← get_avail; lig_ptr(lig_stack) ← p; character(p) ← qi(hu[j + 1]); font(p) ← hf;
          end;
        end;
      end;
    qi(3): begin cur_r ← rem_byte(q); { |=:| }
      p ← lig_stack; lig_stack ← new_lig_item(cur_r); link(lig_stack) ← p;
      end;
    qi(7), qi(11): begin wrap_lig(false); { |=:|>, |=:|>> }
      cur_q ← t; cur_l ← rem_byte(q); ligature_present ← true;
      end;
  othercases begin cur_l ← rem_byte(q); ligature_present ← true; { =: }
    if lig_stack > null then pop_lig_stack
    else if j = n then goto done
    else begin append_charnode_to_t(cur_r); incr(j); set_cur_r;
    end;
  end
endcases;
if op_byte(q) > qi(4) then
  if op_byte(q) ≠ qi(7) then goto done;
goto continue;
end

```

This code is used in section 1086.

1089. Okay, we're ready to insert the potential hyphenations that were found. When the following program is executed, we want to append the word *hu*[1 .. *hn*] after node *ha*, and node *q* should be appended to the result. During this process, the variable *i* will be a temporary index into *hu*; the variable *j* will be an index to our current position in *hu*; the variable *l* will be the counterpart of *j*, in a discretionary branch; the variable *r* will point to new nodes being created; and we need a few new local variables:

⟨ Local variables for hyphenation 1078 ⟩ ≡

```

major_tail, minor_tail: pointer;
{ the end of lists in the main and discretionary branches being reconstructed }
c: ASCII_code; { character temporarily replaced by a hyphen }
c_loc: 0 .. 63; { where that character came from }
r_count: integer; { replacement count for discretionary }
hyf_node: pointer; { the hyphen, if it exists }

```

1090. When the following code is performed, $hyf[0]$ and $hyf[hn]$ will be zero.

⟨ Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 1090 ⟩ ≡

```

repeat  $l \leftarrow j$ ;  $j \leftarrow \text{reconstitute}(j, hn, bchar, qi(hyf\_char)) + 1$ ;
  if  $\text{hyphen\_passed} = 0$  then
    begin  $link(s) \leftarrow link(\text{hold\_head})$ ;
    while  $link(s) > \text{null}$  do  $s \leftarrow link(s)$ ;
    if  $\text{odd}(hyf[j - 1])$  then
      begin  $l \leftarrow j$ ;  $\text{hyphen\_passed} \leftarrow j - 1$ ;  $link(\text{hold\_head}) \leftarrow \text{null}$ ;
      end;
    end;
  if  $\text{hyphen\_passed} > 0$  then ⟨ Create and append a discretionary node as an alternative to the
    unhyphenated word, and continue to develop both branches until they become equivalent 1091 ⟩;
until  $j > hn$ ;
 $link(s) \leftarrow q$ 
```

This code is used in section 1080.

1091. In this repeat loop we will insert another discretionary if $hyf[j - 1]$ is odd, when both branches of the previous discretionary end at position $j - 1$. Strictly speaking, we aren't justified in doing this, because we don't know that a hyphen after $j - 1$ is truly independent of those branches. But in almost all applications we would rather not lose a potentially valuable hyphenation point. (Consider the word 'difficult', where the letter 'c' is in position j .)

```

define  $\text{advance\_major\_tail} \equiv$ 
  begin  $major\_tail \leftarrow link(major\_tail)$ ;  $\text{incr}(r\_count)$ ;
  end

⟨ Create and append a discretionary node as an alternative to the unhyphenated word, and continue to
  develop both branches until they become equivalent 1091 ⟩ ≡
repeat  $r \leftarrow \text{get\_node}(\text{small\_node\_size})$ ;  $link(r) \leftarrow link(\text{hold\_head})$ ;  $\text{type}(r) \leftarrow \text{disc\_node}$ ;  $major\_tail \leftarrow r$ ;
 $r\_count \leftarrow 0$ ;
  while  $link(major\_tail) > \text{null}$  do  $\text{advance\_major\_tail}$ ;
   $i \leftarrow \text{hyphen\_passed}$ ;  $hyf[i] \leftarrow 0$ ; ⟨ Put the characters  $hu[l \dots i]$  and a hyphen into  $\text{pre\_break}(r)$  1092 ⟩;
  ⟨ Put the characters  $hu[i + 1 \dots ]$  into  $\text{post\_break}(r)$ , appending to this list and to  $major\_tail$  until
    synchronization has been achieved 1093 ⟩;
  ⟨ Move pointer  $s$  to the end of the current list, and set  $\text{replace\_count}(r)$  appropriately 1095 ⟩;
   $hyphen\_passed \leftarrow j - 1$ ;  $link(\text{hold\_head}) \leftarrow \text{null}$ ;
until  $\neg\text{odd}(hyf[j - 1])$ 
```

This code is used in section 1090.

1092. The new hyphen might combine with the previous character via ligature or kern. At this point we have $l - 1 \leq i < j$ and $i < hn$.

```
< Put the characters hu[l .. i] and a hyphen into pre_break(r) 1092 > ≡
  minor_tail ← null; pre_break(r) ← null; hyf_node ← new_character(hf, hyf_char);
  if hyf_node ≠ null then
    begin incr(i); c ← hu[i]; hu[i] ← hyf_char; free_avail(hyf_node);
    end;
  while l ≤ i do
    begin l ← reconstitute(l, i, font_bchar[hf], non_char) + 1;
    if link(holder_head) > null then
      begin if minor_tail = null then pre_break(r) ← link(holder_head)
      else link(minor_tail) ← link(holder_head);
      minor_tail ← link(holder_head);
      while link(minor_tail) > null do minor_tail ← link(minor_tail);
      end;
    end;
  if hyf_node ≠ null then
    begin hu[i] ← c; { restore the character in the hyphen position }
    l ← i; decr(i);
    end
```

This code is used in section 1091.

1093. The synchronization algorithm begins with $l = i + 1 \leq j$.

```
< Put the characters hu[i + 1 .. ] into post_break(r), appending to this list and to major_tail until
  synchronization has been achieved 1093 > ≡
  minor_tail ← null; post_break(r) ← null; c_loc ← 0;
  if bchar_label[hf] ≠ non_address then { put left boundary at beginning of new line }
    begin decr(l); c ← hu[l]; c_loc ← l; hu[l] ← 256;
    end;
  while l < j do
    begin repeat l ← reconstitute(l, hn, bchar, non_char) + 1;
    if c_loc > 0 then
      begin hu[c_loc] ← c; c_loc ← 0;
      end;
    if link(holder_head) > null then
      begin if minor_tail = null then post_break(r) ← link(holder_head)
      else link(minor_tail) ← link(holder_head);
      minor_tail ← link(holder_head);
      while link(minor_tail) > null do minor_tail ← link(minor_tail);
      end;
    until l ≥ j;
    while l > j do < Append characters of hu[j .. ] to major_tail, advancing j 1094 >;
    end
```

This code is used in section 1091.

1094. < Append characters of hu[j ..] to major_tail, advancing j 1094 > ≡

```
begin j ← reconstitute(j, hn, bchar, non_char) + 1; link(major_tail) ← link(holder_head);
while link(major_tail) > null do advance_major_tail;
end
```

This code is used in section 1093.

1095. Ligature insertion can cause a word to grow exponentially in size. Therefore we must test the size of r_count here, even though the hyphenated text was at most 63 characters long.

```
(Move pointer  $s$  to the end of the current list, and set  $replace\_count(r)$  appropriately 1095) ≡  
if  $r\_count > 127$  then { we have to forget the discretionary hyphen }  
begin  $link(s) \leftarrow link(r)$ ;  $link(r) \leftarrow null$ ;  $flush\_node\_list(r)$ ;  
end  
else begin  $link(s) \leftarrow r$ ;  $replace\_count(r) \leftarrow r\_count$ ;  
end;  
 $s \leftarrow major\_tail$ 
```

This code is used in section 1091.

1096. Hyphenation. When a word $hc[1 \dots hn]$ has been set up to contain a candidate for hyphenation, TeX first looks to see if it is in the user's exception dictionary. If not, hyphens are inserted based on patterns that appear within the given word, using an algorithm due to Frank M. Liang.

Let's consider Liang's method first, since it is much more interesting than the exception-lookup routine. The algorithm begins by setting $hyf[j]$ to zero for all j , and invalid characters are inserted into $hc[0]$ and $hc[hn+1]$ to serve as delimiters. Then a reasonably fast method is used to see which of a given set of patterns occurs in the word $hc[0 \dots (hn+1)]$. Each pattern $p_1 \dots p_k$ of length k has an associated sequence of $k+1$ numbers $n_0 \dots n_k$; and if the pattern occurs in $hc[(j+1) \dots (j+k)]$, TeX will set $hyf[j+i] \leftarrow \max(hyf[j+i], n_i)$ for $0 \leq i \leq k$. After this has been done for each pattern that occurs, a discretionary hyphen will be inserted between $hc[j]$ and $hc[j+1]$ when $hyf[j]$ is odd, as we have already seen.

The set of patterns $p_1 \dots p_k$ and associated numbers $n_0 \dots n_k$ depends, of course, on the language whose words are being hyphenated, and on the degree of hyphenation that is desired. A method for finding appropriate p 's and n 's, from a given dictionary of words and acceptable hyphenations, is discussed in Liang's Ph.D. thesis (Stanford University, 1983); TeX simply starts with the patterns and works from there.

1097. The patterns are stored in a compact table that is also efficient for retrieval, using a variant of "trie memory" [cf. *The Art of Computer Programming* 3 (1973), 481–505]. We can find each pattern $p_1 \dots p_k$ by letting z_0 be one greater than the relevant language index and then, for $1 \leq i \leq k$, setting $z_i \leftarrow trie_link(z_{i-1}) + p_i$; the pattern will be identified by the number z_k . Since all the pattern information is packed together into a single *trie_link* array, it is necessary to prevent confusion between the data from inequivalent patterns, so another table is provided such that $trie_char(z_i) = p_i$ for all i . There is also a table *trie_op*(z_k) to identify the numbers $n_0 \dots n_k$ associated with $p_1 \dots p_k$.

Comparatively few different number sequences $n_0 \dots n_k$ actually occur, since most of the n 's are generally zero. Therefore the number sequences are encoded in such a way that *trie_op*(z_k) is only one byte long. If *trie_op*(z_k) $\neq min_quarterword$, when $p_1 \dots p_k$ has matched the letters in $hc[(l-k+1) \dots l]$ of language t , we perform all of the required operations for this pattern by carrying out the following little program: Set $v \leftarrow trie_op(z_k)$. Then set $v \leftarrow v + op_start[t]$, $hyf[l-hyf_distance[v]] \leftarrow \max(hyf[l-hyf_distance[v]], hyf_num[v])$, and $v \leftarrow hyf_next[v]$; repeat, if necessary, until $v = min_quarterword$.

```
<Types in the outer block 18> +≡
  trie_pointer = 0 .. trie_size; { an index into trie }
```

1098. `define trie_link(#) ≡ trie[#].rh { "downward" link in a trie }`
`define trie_char(#) ≡ trie[#].b1 { character matched at this trie location }`
`define trie_op(#) ≡ trie[#].b0 { program for hyphenation at this trie location }`
<Global variables 13> +≡
`trie: array [trie_pointer] of two_halves; { trie_link, trie_char, trie_op }`
`hyf_distance: array [1 .. trie_op_size] of small_number; { position $k-j$ of n_j }`
`hyf_num: array [1 .. trie_op_size] of small_number; { value of n_j }`
`hyf_next: array [1 .. trie_op_size] of quarterword; { continuation code }`
`op_start: array [ASCII_code] of 0 .. trie_op_size; { offset for current language }`

1099. <Local variables for hyphenation 1078> +≡
`z: trie_pointer; { an index into trie }`
`v: integer; { an index into hyf_distance, etc. }`

1100. Assuming that these auxiliary tables have been set up properly, the hyphenation algorithm is quite short. In the following code we set $hc[hn + 2]$ to the impossible value 256, in order to guarantee that $hc[hn + 3]$ will never be fetched.

```

⟨Find hyphen locations for the word in hc, or return 1100⟩ ≡
  for j ← 0 to hn do hyf[j] ← 0;
  ⟨Look for the word hc[1 .. hn] in the exception table, and goto found (with hyf containing the hyphens)
    if an entry is found 1107⟩;
  if trie_char(cur_lang + 1) ≠ qi(cur_lang) then return; { no patterns for cur_lang }
  hc[0] ← 0; hc[hn + 1] ← 0; hc[hn + 2] ← 256; { insert delimiters }
  for j ← 0 to hn – r_hyf + 1 do
    begin z ← trie_link(cur_lang + 1) + hc[j]; l ← j;
    while hc[l] = qo(trie_char(z)) do
      begin if trie_op(z) ≠ min_quarterword then ⟨Store maximum values in the hyf table 1101⟩;
        incr(l); z ← trie_link(z) + hc[l];
      end;
    end;
  found: for j ← 0 to l_hyf – 1 do hyf[j] ← 0;
  for j ← 0 to r_hyf – 1 do hyf[hn – j] ← 0

```

This code is used in section 1072.

1101. ⟨Store maximum values in the hyf table 1101⟩ ≡

```

begin v ← trie_op(z);
repeat v ← v + op_start[cur_lang]; i ← l – hyf_distance[v];
  if hyf_num[v] > hyf[i] then hyf[i] ← hyf_num[v]
  v ← hyf_next[v];
until v = min_quarterword;
end

```

This code is used in section 1100.

1102. The exception table that is built by TeX's \hyphenation primitive is organized as an ordered hash table [cf. Amble and Knuth, *The Computer Journal* 17 (1974), 135–142] using linear probing. If α and β are words, we will say that $\alpha < \beta$ if $|\alpha| < |\beta|$ or if $|\alpha| = |\beta|$ and α is lexicographically smaller than β . (The notation $|\alpha|$ stands for the length of α .) The idea of ordered hashing is to arrange the table so that a given word α can be sought by computing a hash address $h = h(\alpha)$ and then looking in table positions $h, h - 1, \dots$, until encountering the first word $\leq \alpha$. If this word is different from α , we can conclude that α is not in the table.

The words in the table point to lists in *mem* that specify hyphen positions in their *info* fields. The list for $c_1 \dots c_n$ contains the number k if the word $c_1 \dots c_n$ has a discretionary hyphen between c_k and c_{k+1} .

```

⟨Types in the outer block 18⟩ +≡
  hyph_pointer = 0 .. hyph_size; { an index into the ordered hash table }

```

1103. ⟨Global variables 13⟩ +≡

```

hyph_word: array [hyph_pointer] of str_number; { exception words }
hyph_list: array [hyph_pointer] of pointer; { lists of hyphen positions }
hyph_count: hyph_pointer; { the number of words in the exception dictionary }

```

1104. ⟨Local variables for initialization 19⟩ +≡

```

z: hyph_pointer; { runs through the exception dictionary }

```

1105. \langle Set initial values of key variables 21 $\rangle +\equiv$
for $z \leftarrow 0$ **to** $hyph_size$ **do**
 begin $hyph_word[z] \leftarrow 0$; $hyph_list[z] \leftarrow null$;
 end;
 $hyph_count \leftarrow 0$;

1106. The algorithm for exception lookup is quite simple, as soon as we have a few more local variables to work with.

\langle Local variables for hyphenation 1078 $\rangle +\equiv$
 h : $hyph_pointer$; { an index into $hyph_word$ and $hyph_list$ }
 k : str_number ; { an index into str_start }
 u : $pool_pointer$; { an index into str_pool }

1107. First we compute the hash code h , then we search until we either find the word or we don't. Words from different languages are kept separate by appending the language code to the string.

\langle Look for the word $hc[1 \dots hn]$ in the exception table, and **goto** $found$ (with hyf containing the hyphens) if an entry is found 1107 $\rangle \equiv$
 $h \leftarrow hc[1]$; $incr(hn)$; $hc[hn] \leftarrow cur_lang$;
for $j \leftarrow 2$ **to** hn **do** $h \leftarrow (h + h + hc[j]) \bmod hyph_size$;
loop begin \langle If the string $hyph_word[h]$ is less than $hc[1 \dots hn]$, **goto** not_found ; but if the two strings are equal, set hyf to the hyphen positions and **goto** $found$ 1108 \rangle ;
 if $h > 0$ **then** $decr(h)$ **else** $h \leftarrow hyph_size$;
 end;
 not_found : $decr(hn)$

This code is used in section 1100.

1108. \langle If the string $hyph_word[h]$ is less than $hc[1 \dots hn]$, **goto** not_found ; but if the two strings are equal, set hyf to the hyphen positions and **goto** $found$ 1108 $\rangle \equiv$

$k \leftarrow hyph_word[h]$;
if $k = 0$ **then** **goto** not_found ;
if $length(k) < hn$ **then** **goto** not_found ;
if $length(k) = hn$ **then**
 begin $j \leftarrow 1$; $u \leftarrow str_start[k]$;
 repeat if $so(str_pool[u]) < hc[j]$ **then goto** not_found ;
 if $so(str_pool[u]) > hc[j]$ **then goto** $done$;
 $incr(j)$; $incr(u)$;
 until $j > hn$;
 \langle Insert hyphens as specified in $hyph_list[h]$ 1109 \rangle ;
 $decr(hn)$; **goto** $found$;
 end;

done:

This code is used in section 1107.

1109. \langle Insert hyphens as specified in $hyph_list[h]$ 1109 $\rangle \equiv$

$s \leftarrow hyph_list[h]$;
while $s \neq null$ **do**
 begin $hyf[info(s)] \leftarrow 1$; $s \leftarrow link(s)$;
 end

This code is used in section 1108.

```
1110. <Search hyph_list for pointers to p 1110> ≡
  for q ← 0 to hyph_size do
    begin if hyph_list[q] = p then
      begin print_nl("HYPH("); print_int(q); print_char(")");
      end;
    end
```

This code is used in section 190.

1111. 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 1112};
exit: end;
```

1112. \langle Enter as many hyphenation exceptions as are listed, until coming to a right brace; then
return 1112 $\rangle \equiv$
 $n \leftarrow 0; p \leftarrow null;$
loop begin get_x_token;
reswitch: **case** cur_cmd **of**
 letter, other_char, char_given: \langle Append a new letter or hyphen 1114 $\rangle;$
 char_num: **begin** scan_char_num; cur_chr \leftarrow cur_val; cur_cmd \leftarrow char_given; **goto** reswitch;
 end;
 spacer, right_brace: **begin if** $n > 1$ **then** \langle Enter a hyphenation exception 1116 $\rangle;$
 if cur_cmd = right_brace **then return;**
 $n \leftarrow 0; p \leftarrow null;$
 end;
 othercases \langle Give improper \hyphenation error 1113 \rangle
 endcases;
end

This code is used in section 1111.

1113. \langle Give improper \hyphenation error 1113 $\rangle \equiv$
begin print_err("Improper"); print_esc("hyphenation"); print(" will be flushed");
 help2("Hyphenation exceptions must contain only letters")
 ("and hyphens. But continue; I'll forgive and forget."); error;
end

This code is used in section 1112.

1114. \langle Append a new letter or hyphen 1114 $\rangle \equiv$
if cur_chr = "-" **then** \langle Append the value n to list p 1115 \rangle
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] \leftarrow hc[0];
 end;
end

This code is used in section 1112.

1115. \langle Append the value n to list p 1115 $\rangle \equiv$
begin if n < 63 **then**
 begin q \leftarrow get_avail; link(q) \leftarrow p; info(q) \leftarrow n; p \leftarrow q;
 end;
end

This code is used in section 1114.

1116. \langle Enter a hyphenation exception 1116 $\rangle \equiv$
begin *incr*(*n*); *hc*[*n*] \leftarrow *cur_lang*; *str_room*(*n*); *h* \leftarrow 0;
for *j* \leftarrow 1 **to** *n* **do**
 begin *h* \leftarrow (*h* + *h* + *hc*[*j*]) **mod** *hyph_size*; *append_char*(*hc*[*j*]);
 end;
 s \leftarrow *make_string*; \langle Insert the pair (*s*, *p*) into the exception table 1117 \rangle ;
end

This code is used in section 1112.

1117. \langle Insert the pair (*s*, *p*) into the exception table 1117 $\rangle \equiv$
if *hyph_count* = *hyph_size* **then** *overflow*("exception_dictionary", *hyph_size*);
 incr(*hyph_count*);
while *hyph_word*[*h*] \neq 0 **do**
 begin \langle If the string *hyph_word*[*h*] is less than or equal to *s*, interchange (*hyph_word*[*h*], *hyph_list*[*h*])
 with (*s*, *p*) 1118 \rangle ;
 if *h* > 0 **then** *decr*(*h*) **else** *h* \leftarrow *hyph_size*;
 end;
 hyph_word[*h*] \leftarrow *s*; *hyph_list*[*h*] \leftarrow *p*

This code is used in section 1116.

1118. \langle If the string *hyph_word*[*h*] is less than or equal to *s*, interchange (*hyph_word*[*h*], *hyph_list*[*h*]) with
 (*s*, *p*) 1118 $\rangle \equiv$
 k \leftarrow *hyph_word*[*h*];
 if *length*(*k*) < *length*(*s*) **then** **goto** *found*;
 if *length*(*k*) > *length*(*s*) **then** **goto** *not_found*;
 u \leftarrow *str_start*[*k*]; *v* \leftarrow *str_start*[*s*];
 repeat **if** *str_pool*[*u*] < *str_pool*[*v*] **then** **goto** *found*;
 if *str_pool*[*u*] > *str_pool*[*v*] **then** **goto** *not_found*;
 incr(*u*); *incr*(*v*);
 until *u* = *str_start*[*k* + 1];
found: *q* \leftarrow *hyph_list*[*h*]; *hyph_list*[*h*] \leftarrow *p*; *p* \leftarrow *q*;
 t \leftarrow *hyph_word*[*h*]; *hyph_word*[*h*] \leftarrow *s*; *s* \leftarrow *t*;
not_found:

This code is used in section 1117.

1119. Initializing the hyphenation tables. The trie for TeX's hyphenation algorithm is built from a sequence of patterns following a \patterns specification. Such a specification is allowed only in INITEX, since the extra memory for auxiliary tables and for the initialization program itself would only clutter up the production version of TeX with a lot of deadwood.

The first step is to build a trie that is linked, instead of packed into sequential storage, so that insertions are readily made. After all patterns have been processed, INITEX compresses the linked trie by identifying common subtrees. Finally the trie is packed into the efficient sequential form that the hyphenation algorithm actually uses.

```
< Declare subprocedures for line_break 1002 > +≡
  init < Declare procedures for preprocessing hyphenation patterns 1121 >
    tini
```

1120. Before we discuss trie building in detail, let's consider the simpler problem of creating the *hyf_distance*, *hyf_num*, and *hyf_next* arrays.

Suppose, for example, that TeX reads the pattern 'ab2cde1'. This is a pattern of length 5, with $n_0 \dots n_5 = 002001$ in the notation above. We want the corresponding *trie_op* code v to have $hyf_distance[v] = 3$, $hyf_num[v] = 2$, and $hyf_next[v] = v'$, where the auxiliary *trie_op* code v' has $hyf_distance[v'] = 0$, $hyf_num[v'] = 1$, and $hyf_next[v'] = min_quarterword$.

TeX computes an appropriate value v with the *new_trie_op* subroutine below, by setting

$$v' \leftarrow new_trie_op(0, 1, min_quarterword), \quad v \leftarrow new_trie_op(3, 2, v').$$

This subroutine looks up its three parameters in a special hash table, assigning a new value only if these three have not appeared before for the current language.

The hash table is called *trie_op_hash*, and the number of entries it contains is *trie_op_ptr*.

```
< Global variables 13 > +≡
  init trie_op_hash: array [-trie_op_size .. trie_op_size] of 0 .. trie_op_size;
    { trie op codes for quadruples }
  trie_used: array [ASCII_code] of quarterword; { largest opcode used so far for this language }
  trie_op_lang: array [1 .. trie_op_size] of ASCII_code; { language part of a hashed quadruple }
  trie_op_val: array [1 .. trie_op_size] of quarterword; { opcode corresponding to a hashed quadruple }
  trie_op_ptr: 0 .. trie_op_size; { number of stored ops so far }
  tini
```

1121. It's tempting to remove the *overflow* stops in the following procedure; *new_trie_op* could return *min_quarterword* (thereby simply ignoring part of a hyphenation pattern) instead of aborting the job. However, that would lead to different hyphenation results on different installations of TeX using the same patterns. The *overflow* stops are necessary for portability of patterns.

```
< Declare procedures for preprocessing hyphenation patterns 1121 > ≡
function new_trie_op(d, n : small_number; v : quarterword): quarterword;
label exit;
var h: −trie_op_size .. trie_op_size; { trial hash location }
    u: quarterword; { trial op code }
    l: 0 .. trie_op_size; { pointer to stored data }
begin h ← abs(n + 313 * d + 361 * v + 1009 * cur_lang) mod (trie_op_size + trie_op_size) − trie_op_size;
loop begin l ← trie_op_hash[h];
    if l = 0 then { empty position found for a new op }
        begin if trie_op_ptr = trie_op_size then overflow("pattern_memory_ops", trie_op_size);
            u ← trie_used[cur_lang];
            if u = max_quarterword then
                overflow('pattern_memory_ops_per_language', max_quarterword − min_quarterword);
                incr(trie_op_ptr); incr(u); trie_used[cur_lang] ← u; hyf_distance[trie_op_ptr] ← d;
                hyf_num[trie_op_ptr] ← n; hyf_next[trie_op_ptr] ← v; trie_op_lang[trie_op_ptr] ← cur_lang;
                trie_op_hash[h] ← trie_op_ptr; trie_op_val[trie_op_ptr] ← u; new_trie_op ← u; return;
            end;
            if (hyf_distance[l] = d) ∧ (hyf_num[l] = n) ∧ (hyf_next[l] = v) ∧ (trie_op_lang[l] = cur_lang) then
                begin new_trie_op ← trie_op_val[l]; return;
            end;
            if h > −trie_op_size then decr(h) else h ← trie_op_size;
        end;
    exit: end;
```

See also sections 1125, 1126, 1130, 1134, 1136, 1137, and 1143.

This code is used in section 1119.

1122. After *new_trie_op* has compressed the necessary opcode information, plenty of information is available to unscramble the data into the final form needed by our hyphenation algorithm.

```
< Sort the hyphenation op tables into proper order 1122 > ≡
op_start[0] ← −min_quarterword;
for j ← 1 to 255 do op_start[j] ← op_start[j − 1] + qo(trie_used[j − 1]);
for j ← 1 to trie_op_ptr do trie_op_hash[j] ← op_start[trie_op_lang[j]] + trie_op_val[j] { destination }
for j ← 1 to trie_op_ptr do
    while trie_op_hash[j] > j do
        begin k ← trie_op_hash[j];
            t ← hyf_distance[k]; hyf_distance[k] ← hyf_distance[j]; hyf_distance[j] ← t;
            t ← hyf_num[k]; hyf_num[k] ← hyf_num[j]; hyf_num[j] ← t;
            t ← hyf_next[k]; hyf_next[k] ← hyf_next[j]; hyf_next[j] ← t;
            trie_op_hash[j] ← trie_op_hash[k]; trie_op_hash[k] ← k;
        end
```

This code is used in section 1129.

1123. Before we forget how to initialize the data structures that have been mentioned so far, let's write down the code that gets them started.

```
( Initialize table entries (done by INITEX only) 182 ) +≡
  for k ← −trie_op_size to trie_op_size do trie_op_hash[k] ← 0;
  for k ← 0 to 255 do trie_used[k] ← min_quarterword;
  trie_op_ptr ← 0;
```

1124. The linked trie that is used to preprocess hyphenation patterns appears in several global arrays. Each node represents an instruction of the form "if you see character c , then perform operation o , move to the next character, and go to node l ; otherwise go to node r ." The four quantities c , o , l , and r are stored in four arrays trie_c , trie_o , trie_l , and trie_r . The root of the trie is $\text{trie_l}[0]$, and the number of nodes is trie_ptr . Null trie pointers are represented by zero. To initialize the trie, we simply set $\text{trie_l}[0]$ and trie_ptr to zero. We also set $\text{trie_c}[0]$ to some arbitrary value, since the algorithm may access it.

The algorithms maintain the condition

$$\text{trie_c}[\text{trie_r}[z]] > \text{trie_c}[z] \quad \text{whenever } z \neq 0 \text{ and } \text{trie_r}[z] \neq 0;$$

in other words, sibling nodes are ordered by their c fields.

```
define trie_root ≡ trie_l[0] { root of the linked trie }
(Global variables 13) +≡
  init trie_c: packed array [trie_pointer] of packed_ASCII_code; { characters to match }
  trie_o: packed array [trie_pointer] of quarterword; { operations to perform }
  trie_l: packed array [trie_pointer] of trie_pointer; { left subtrie links }
  trie_r: packed array [trie_pointer] of trie_pointer; { right subtrie links }
  trie_ptr: trie_pointer; { the number of nodes in the trie }
  trie_hash: packed array [trie_pointer] of trie_pointer; { used to identify equivalent subtrees }
  tini
```

1125. Let us suppose that a linked trie has already been constructed. Experience shows that we can often reduce its size by recognizing common subtrees; therefore another hash table is introduced for this purpose, somewhat similar to trie_op_hash . The new hash table will be initialized to zero.

The function $\text{trie_node}(p)$ returns p if p is distinct from other nodes that it has seen, otherwise it returns the number of the first equivalent node that it has seen.

Notice that we might make subtrees equivalent even if they correspond to patterns for different languages, in which the trie ops might mean quite different things. That's perfectly all right.

```
( Declare procedures for preprocessing hyphenation patterns 1121 ) +≡
function trie_node(p : trie_pointer): trie_pointer; { converts to a canonical form }
  label exit;
  var h: trie_pointer; { trial hash location }
  q: trie_pointer; { trial trie node }
  begin h ← abs(trie_c[p] + 1009 * trie_o[p] + 2718 * trie_l[p] + 3142 * trie_r[p]) mod trie_size;
  loop begin q ← trie_hash[h];
    if q = 0 then
      begin trie_hash[h] ← p; trie_node ← p; return;
    end;
    if (trie_c[q] = trie_c[p]) ∧ (trie_o[q] = trie_o[p]) ∧ (trie_l[q] = trie_l[p]) ∧ (trie_r[q] = trie_r[p]) then
      begin trie_node ← q; return;
    end;
    if h > 0 then decr(h) else h ← trie_size;
  end;
exit: end;
```

1126. A neat recursive procedure is now able to compress a trie by traversing it and applying *trie_node* to its nodes in “bottom up” fashion. We will compress the entire trie by clearing *trie_hash* to zero and then saying ‘*trie_root* ← *compress_trie*(*trie_root*)’.

```
< Declare procedures for preprocessing hyphenation patterns 1121 > +≡
function compress_trie(p : trie_pointer): trie_pointer;
begin if p = 0 then compress_trie ← 0
else begin trie_l[p] ← compress_trie(trie_l[p]); trie_r[p] ← compress_trie(trie_r[p]);
compress_trie ← trie_node(p);
end;
end;
```

1127. The compressed trie will be packed into the *trie* array using a “top-down first-fit” procedure. This is a little tricky, so the reader should pay close attention: The *trie_hash* array is cleared to zero again and renamed *trie_ref* for this phase of the operation; later on, *trie_ref*[*p*] will be nonzero only if the linked trie node *p* is the smallest character in a family and if the characters *c* of that family have been allocated to locations *trie_ref*[*p*] + *c* in the *trie* array. Locations of *trie* that are in use will have *trie_link* = 0, while the unused holes in *trie* will be doubly linked with *trie_link* pointing to the next larger vacant location and *trie_back* pointing to the next smaller one. This double linking will have been carried out only as far as *trie_max*, where *trie_max* is the largest index of *trie* that will be needed. To save time at the low end of the trie, we maintain array entries *trie_min*[*c*] pointing to the smallest hole that is greater than *c*. Another array *trie_taken* tells whether or not a given location is equal to *trie_ref*[*p*] for some *p*; this array is used to ensure that distinct nodes in the compressed trie will have distinct *trie_ref* entries.

```
define trie_ref ≡ trie_hash { where linked trie families go into trie }
define trie_back(#) ≡ trie[#].lh { backward links in trie holes }
< Global variables 13 > +≡
init trie_taken: packed array [1 .. trie_size] of boolean; { does a family start here? }
trie_min: array [ASCII_code] of trie_pointer; { the first possible slot for each character }
trie_max: trie_pointer; { largest location used in trie }
trie_not_ready: boolean; { is the trie still in linked form? }
tini
```

1128. Each time \patterns appears, it contributes further patterns to the future trie, which will be built only when hyphenation is attempted or when a format file is dumped. The boolean variable *trie_not_ready* will change to *false* when the trie is compressed; this will disable further patterns.

```
< Initialize table entries (done by INITEX only) 182 > +≡
trie_not_ready ← true; trie_root ← 0; trie_c[0] ← si(0); trie_ptr ← 0;
```

1129. 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 1129 > ≡
< Sort the hyphenation op tables into proper order 1122 >;
for p ← 0 to trie_size do trie_hash[p] ← 0;
hyph_root ← compress_trie(hyph_root); trie_root ← compress_trie(trie_root);
{ identify equivalent subtrees }
for p ← 0 to trie_ptr do trie_ref[p] ← 0;
for p ← 0 to 255 do trie_min[p] ← p + 1;
trie_link(0) ← 1; trie_max ← 0
```

This code is used in section 1143.

1130. The *first_fit* procedure finds the smallest hole z in *trie* such that a trie family starting at a given node p will fit into vacant positions starting at z . If $c = \text{trie_c}[p]$, this means that location $z - c$ must not already be taken by some other family, and that $z - c + c'$ must be vacant for all characters c' in the family. The procedure sets *trie_ref*[p] to $z - c$ when the first fit has been found.

```
< Declare procedures for preprocessing hyphenation patterns 1121 > +≡
procedure first_fit( $p$  : trie_pointer); { packs a family into trie }
label not_found, found;
var  $h$ : trie_pointer; { candidate for trie_ref[ $p$ ] }
 $z$ : trie_pointer; { runs through holes }
 $q$ : trie_pointer; { runs through the family starting at  $p$  }
 $c$ : ASCII_code; { smallest character in the family }
 $l, r$ : trie_pointer; { left and right neighbors }
 $ll$ : 1 .. 256; { upper limit of trie_min updating }
begin  $c \leftarrow so(\text{trie\_c}[p])$ ;  $z \leftarrow \text{trie\_min}[c]$ ; { get the first conceivably good hole }
loop begin  $h \leftarrow z - c$ ;
    { Ensure that trie_max ≥  $h + 256$  1131 };
    if trie_taken[ $h$ ] then goto not_found;
    { If all characters of the family fit relative to  $h$ , then goto found, otherwise goto not_found 1132 };
    not_found:  $z \leftarrow \text{trie\_link}(z)$ ; { move to the next hole }
    end;
found: { Pack the family into trie relative to  $h$  1133 };
end;
```

1131. By making sure that *trie_max* is at least $h + 256$, we can be sure that $\text{trie_max} > z$, since $h = z - c$. It follows that location *trie_max* will never be occupied in *trie*, and we will have $\text{trie_max} \geq \text{trie_link}(z)$.

```
< Ensure that trie_max ≥  $h + 256$  1131 > ≡
if trie_max <  $h + 256$  then
begin if trie_size ≤  $h + 256$  then overflow("pattern_memory", trie_size);
repeat incr(trie_max); trie_taken[trie_max] ← false; trie_link(trie_max) ← trie_max + 1;
    trie_back(trie_max) ← trie_max - 1;
until trie_max =  $h + 256$ ;
end
```

This code is used in section 1130.

1132. { If all characters of the family fit relative to h , then goto found, otherwise goto not_found 1132 } ≡

```
 $q \leftarrow \text{trie\_r}[p]$ ;
while  $q > 0$  do
begin if trie_link( $h + so(\text{trie\_c}[q])$ ) = 0 then goto not_found;
 $q \leftarrow \text{trie\_r}[q]$ ;
end;
goto found
```

This code is used in section 1130.

1133. \langle Pack the family into *trie* relative to *h* 1133 $\rangle \equiv$

```

trie_taken[h] ← true; trie_ref[p] ← h; q ← p;
repeat z ← h + so(trie_c[q]); l ← trie_back(z); r ← trie_link(z); trie_back(r) ← l; trie_link(l) ← r;
    trie_link(z) ← 0;
if l < 256 then
    begin if z < 256 then ll ← z else ll ← 256;
    repeat trie_min[l] ← r; incr(l);
    until l = ll;
    end;
    q ← trie_r[q];
until q = 0
```

This code is used in section 1130.

1134. To pack the entire linked trie, we use the following recursive procedure.

\langle Declare procedures for preprocessing hyphenation patterns 1121 $\rangle +\equiv$

```

procedure trie_pack(p : trie_pointer); { pack subtrees of a family }
var q: trie_pointer; { a local variable that need not be saved on recursive calls }
begin repeat q ← trie_l[p];
    if (q > 0) ∧ (trie_ref[q] = 0) then
        begin first_fit(q); trie_pack(q);
        end;
    p ← trie_r[p];
until p = 0;
end;
```

1135. 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.

\langle Move the data into *trie* 1135 $\rangle \equiv$

```

h.rh ← 0; h.b0 ← min_quarterword; h.b1 ← min_quarterword;
{ trie_link ← 0, trie_op ← min_quarterword, trie_char ← qi(0) }
if trie_max = 0 then { no patterns were given }
    begin for r ← 0 to 256 do trie[r] ← h;
    trie_max ← 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 ← 0; { now we will zero out all the holes }
    repeat s ← trie_link(r); trie[r] ← h; r ← s;
    until r > trie_max;
    end;
trie_char(0) ← qi("?");
{ make trie_char(c) ≠ c for all c }
```

This code is used in section 1143.

1136. The fixing-up procedure is, of course, recursive. Since the linked trie usually has overlapping subtrees, the same data may be moved several times; but that causes no harm, and at most as much work is done as it took to build the uncompressed trie.

```
< Declare procedures for preprocessing hyphenation patterns 1121 > +≡
procedure trie_fix(p : trie_pointer); { moves p and its siblings into trie }
  var q: trie_pointer; { a local variable that need not be saved on recursive calls }
  c: ASCII_code; { another one that need not be saved }
  z: trie_pointer; { trie reference; this local variable must be saved }
begin z ← trie_ref[p]
repeat q ← trie_l[p]; c ← so(trie_c[p]); trie_link(z + c) ← trie_ref[q]; trie_char(z + c) ← qi(c);
  trie_op(z + c) ← trie_o[p];
  if q > 0 then trie_fix(q);
  p ← trie_r[p];
until p = 0;
end;
```

1137. 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 1121 > +≡
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 1138 >;
  if saving_hyph_codes > 0 then < Store hyphenation codes for current language 1855 >;
  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;
```

1138. Novices are not supposed to be using \patterns, so the error messages are terse. (Note that all error messages appear in TeX's string pool, even if they are used only by INITEX.)

⟨ Enter all of the patterns into a linked trie, until coming to a right brace 1138 ⟩ ≡

```

 $k \leftarrow 0; hyf[0] \leftarrow 0; digit\_sensed \leftarrow false;$ 
loop begin get_x_token;
  case cur_cmd of
    letter, other_char: ⟨ Append a new letter or a hyphen level 1139 ⟩;
    spacer, right_brace: begin if k > 0 then ⟨ Insert a new pattern into the linked trie 1140 ⟩;
      if cur_cmd = right_brace then goto done;
       $k \leftarrow 0; hyf[0] \leftarrow 0; digit\_sensed \leftarrow false;$ 
      end;
    othercases begin print_err("Bad_\"); print_esc("patterns"); help1("(See_\Appendix_\H.)"); error;
      end
    endcases;
  end;
done:
```

This code is used in section 1137.

1139. ⟨ Append a new letter or a hyphen level 1139 ⟩ ≡

```

if digit_sensed  $\vee$  (cur_chr < "0")  $\vee$  (cur_chr > "9") then
  begin if cur_chr = "." then cur_chr  $\leftarrow$  0 { edge-of-word delimiter }
  else begin cur_chr  $\leftarrow$  lc_code(cur_chr);
    if cur_chr = 0 then
      begin print_err("Nonletter"); help1("(See_\Appendix_\H.)"); error;
      end;
    end;
    if k < 63 then
      begin incr(k); hc[k]  $\leftarrow$  cur_chr; hyf[k]  $\leftarrow$  0; digit_sensed  $\leftarrow$  false;
      end;
    end
  else if k < 63 then
    begin hyf[k]  $\leftarrow$  cur_chr - "0"; digit_sensed  $\leftarrow$  true;
    end
```

This code is used in section 1138.

1140. When the following code comes into play, the pattern $p_1 \dots p_k$ appears in $hc[1 \dots k]$, and the corresponding sequence of numbers $n_0 \dots n_k$ appears in $hyf[0 \dots k]$.

```

⟨ Insert a new pattern into the linked trie 1140 ⟩ ≡
begin ⟨ Compute the trie op code, v, and set l ← 0 1142 ⟩;
q ← 0; hc[0] ← cur_lang;
while l ≤ k do
begin c ← hc[l]; incr(l); p ← trie_l[q]; first_child ← true;
while (p > 0) ∧ (c > so(trie_c[p])) do
begin q ← p; p ← trie_r[q]; first_child ← false;
end;
if (p = 0) ∨ (c < so(trie_c[p])) then
⟨ Insert a new trie node between q and p, and make p point to it 1141 ⟩;
q ← p; { now node q represents  $p_1 \dots p_{l-1}$  }
end;
if trie_o[q] ≠ min_quarterword then
begin print_err("Duplicate_pattern"); help1("(See Appendix H.)"); error;
end;
trie_o[q] ← v;
end

```

This code is used in section 1138.

1141. ⟨ Insert a new trie node between q and p, and make p point to it 1141 ⟩ ≡
begin if $trie_ptr = trie_size$ then overflow("pattern_memory", $trie_size$);
 $incr(trie_ptr)$; $trie_r[trie_ptr] \leftarrow p$; $p \leftarrow trie_ptr$; $trie_l[p] \leftarrow 0$;
if $first_child$ then $trie_l[q] \leftarrow p$ else $trie_r[q] \leftarrow p$;
 $trie_c[p] \leftarrow si(c)$; $trie_o[p] \leftarrow min_quarterword$;
end

This code is used in sections 1140, 1855, and 1856.

1142. ⟨ Compute the trie op code, v, and set l ← 0 1142 ⟩ ≡
if $hc[1] = 0$ then $hyf[0] \leftarrow 0$;
if $hc[k] = 0$ then $hyf[k] \leftarrow 0$;
 $l \leftarrow k$; $v \leftarrow min_quarterword$;
loop begin if $hyf[l] \neq 0$ then $v \leftarrow new_trie_op(k - l, hyf[l], v)$;
if $l > 0$ then $decr(l)$ else goto done1;
end;

done1:

This code is used in section 1140.

1143. 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 1121 ⟩ +≡
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 1129};
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 1857};
{Move the data into trie 1135};
trie_not_ready ← false;
end;
```

1144. *Breaking vertical lists into pages.* The *vsplit* procedure, which implements TeX's \vsplit operation, is considerably simpler than *line_break* because it doesn't have to worry about hyphenation, and because its mission is to discover a single break instead of an optimum sequence of breakpoints. But before we get into the details of *vsplit*, we need to consider a few more basic things.

1145. 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.

```

define discard_or_move = 60
function prune_page_top(p : pointer; s : boolean) : pointer;
  label discard_or_move; { 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 1146 ⟩;
      whatsit_node, mark_node, ins_node: begin if (type(p) = whatsit_node) ∧ ((subtype(p) =
        pdf_snappy_node) ∨ (subtype(p) = pdf_snappy_comp_node)) then
        begin print("snap_node_being_discarded"); goto discard_or_move;
        end;
      prev_p ← p; p ← link(prev_p);
      end;
      glue_node, kern_node, penalty_node: begin discard_or_move: @{print("discard_or_move: ");}
        show_node_list(p); print_ln; @} 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;

```

1146. ⟨ Insert glue for *split_top_skip* and set *p* ← *null* 1146 ⟩ ≡
begin *q* ← *new_skip_param*(*split_top_skip_code*); *link*(*prev_p*) ← *q*; *link*(*q*) ← *p*;
 { now *temp_ptr* = *glue_ptr*(*q*) }
 if *width*(*temp_ptr*) > *height*(*p*) **then** *width*(*temp_ptr*) ← *width*(*temp_ptr*) − *height*(*p*)
 else *width*(*temp_ptr*) ← 0;
 p ← *null*;
end

This code is used in section 1145.

1147. The next subroutine finds the best place to break a given vertical list so as to obtain a box of height h , with maximum depth d . A pointer to the beginning of the vertical list is given, and a pointer to the optimum breakpoint is returned. The list is effectively followed by a forced break, i.e., a penalty node with the *eject_penalty*; if the best break occurs at this artificial node, the value *null* is returned.

An array of six *scaled* distances is used to keep track of the height from the beginning of the list to the current place, just as in *line_break*. In fact, we use one of the same arrays, only changing its name to reflect its new significance.

```

define active_height ≡ active_width { new name for the six distance variables }
define cur_height ≡ active_height[1] { the natural height }
define set_height_zero(#) ≡ active_height[#] ← 0 { initialize the height to zero }
define update_heights = 90 { go here to record glue in the active_height table }

function vert_break(p : pointer; h, d : scaled): pointer; { finds optimum page break }
label done, not_found, update_heights;
var prev_p: pointer; { if p is a glue node, type(prev_p) determines whether p is a legal breakpoint }
q, r: pointer; { glue specifications }
pi: integer; { penalty value }
b: integer; { badness at a trial breakpoint }
least_cost: integer; { the smallest badness plus penalties found so far }
best_place: pointer; { the most recent break that leads to least_cost }
prev_dp: scaled; { depth of previous box in the list }
t: small_number; { type of the node following a kern }
begin prev_p ← p; { an initial glue node is not a legal breakpoint }
least_cost ← awful.bad; do_all_six(set_height_zero); prev_dp ← 0;
loop begin { If node p is a legal breakpoint, check if this break is the best known, and goto done if p is
    null or if the page-so-far is already too full to accept more stuff 1149 };
    prev_p ← p; p ← link(prev_p);
    end;
done: vert_break ← best_place;
end;

```

1148. A global variable *best_height_plus_depth* will be set to the natural size of the box that corresponds to the optimum breakpoint found by *vert_break*. (This value is used by the insertion-splitting algorithm of the page builder.)

```

⟨ Global variables 13 ⟩ +≡
best_height_plus_depth: scaled; { height of the best box, without stretching or shrinking }

```

1149. A subtle point to be noted here is that the maximum depth d might be negative, so cur_height and $prev_dp$ might need to be corrected even after a glue or kern node.

```

⟨ If node  $p$  is a legal breakpoint, check if this break is the best known, and goto  $done$  if  $p$  is null or if the
    page-so-far is already too full to accept more stuff 1149 ⟩ ≡
  if  $p = null$  then  $pi \leftarrow eject\_penalty$ 
  else ⟨ Use node  $p$  to update the current height and depth measurements; if this node is not a legal
    breakpoint, goto  $not\_found$  or  $update\_heights$ , otherwise set  $pi$  to the associated penalty at the
    break 1150 ⟩;
  ⟨ Check if node  $p$  is a new champion breakpoint; then goto  $done$  if  $p$  is a forced break or if the page-so-far
    is already too full 1151 ⟩;
  if ( $type(p) < glue\_node$ )  $\vee$  ( $type(p) > kern\_node$ ) then goto  $not\_found$ ;
update_heights: ⟨ Update the current height and depth measurements with respect to a glue or kern
    node  $p$  1153 ⟩;
not_found: if  $prev\_dp > d$  then
  begin  $cur\_height \leftarrow cur\_height + prev\_dp - d$ ;  $prev\_dp \leftarrow d$ ;
  end;
```

This code is used in section 1147.

1150. ⟨ Use node p to update the current height and depth measurements; if this node is not a legal
breakpoint, **goto** not_found or $update_heights$, otherwise set pi to the associated penalty at the
break 1150 ⟩ ≡

```

case  $type(p)$  of
  hlist_node, vlist_node, rule_node: begin
     $cur\_height \leftarrow cur\_height + prev\_dp + height(p)$ ;  $prev\_dp \leftarrow depth(p)$ ; goto  $not\_found$ ;
    end;
  whatsit_node: ⟨ Process whatsit  $p$  in  $vert\_break$  loop, goto  $not\_found$  1612 ⟩;
  glue_node: if  $precedes\_break(prev\_p)$  then  $pi \leftarrow 0$ 
    else goto  $update\_heights$ ;
  kern_node: begin if  $link(p) = null$  then  $t \leftarrow penalty\_node$ 
    else  $t \leftarrow type(link(p))$ ;
    if  $t = glue\_node$  then  $pi \leftarrow 0$  else goto  $update\_heights$ ;
    end;
  penalty_node:  $pi \leftarrow penalty(p)$ ;
  mark_node, ins_node: goto  $not\_found$ ;
  othercases confusion("vertbreak")
  endcases
```

This code is used in section 1149.

1151. `define deplorable ≡ 100000 { more than inf_bad, but less than awful_bad }`

`⟨ Check if node p is a new champion breakpoint; then goto done if p is a forced break or if the page-so-far is already too full 1151 ⟩ ≡`

```

if  $pi < inf\_penalty$  then
  begin ⟨ Compute the badness,  $b$ , using awful_bad if the box is too full 1152 ⟩;
  if  $b < awful\_bad$  then
    if  $pi \leq eject\_penalty$  then  $b \leftarrow pi$ 
    else if  $b < inf\_bad$  then  $b \leftarrow b + pi$ 
    else  $b \leftarrow deplorable$ ;
  if  $b \leq least\_cost$  then
    begin  $best\_place \leftarrow p$ ;  $least\_cost \leftarrow b$ ;  $best\_height\_plus\_depth \leftarrow cur\_height + prev\_dp$ ;
    end;
  if ( $b = awful\_bad$ )  $\vee$  ( $pi \leq eject\_penalty$ ) then goto done;
  end

```

This code is used in section 1149.

1152. ⟨ Compute the badness, b , using *awful_bad* if the box is too full 1152 ⟩ ≡

```

if  $cur\_height < h$  then
  if ( $active\_height[3] \neq 0$ )  $\vee$  ( $active\_height[4] \neq 0$ )  $\vee$  ( $active\_height[5] \neq 0$ ) then  $b \leftarrow 0$ 
  else  $b \leftarrow badness(h - cur\_height, active\_height[2])$ 
else if  $cur\_height - h > active\_height[6]$  then  $b \leftarrow awful\_bad$ 
  else  $b \leftarrow badness(cur\_height - h, active\_height[6])$ 

```

This code is used in section 1151.

1153. Vertical lists that are subject to the *vert_break* procedure should not contain infinite shrinkability, since that would permit any amount of information to “fit” on one page.

`⟨ Update the current height and depth measurements with respect to a glue or kern node p 1153 ⟩ ≡`

```

if  $type(p) = kern\_node$  then  $q \leftarrow p$ 
else begin  $q \leftarrow glue\_ptr(p)$ ;
   $active\_height[2 + stretch\_order(q)] \leftarrow active\_height[2 + stretch\_order(q)] + stretch(q)$ ;
   $active\_height[6] \leftarrow active\_height[6] + shrink(q)$ ;
  if ( $shrink\_order(q) \neq normal$ )  $\wedge$  ( $shrink(q) \neq 0$ ) then
    begin
      print_err("Infinite glue shrinkage found in box being split");
      help4("The box you are vsplitting contains some infinitely shrinkable glue, e.g., \vss or \vskip Opt minus 1fil.");
      ("Such glue doesn't belong there; but you can safely proceed,")
      ("since the offensive shrinkability has been made finite."); error; r \leftarrow new_spec(q);
       $shrink\_order(r) \leftarrow normal$ ;  $delete\_glue\_ref(q)$ ;  $glue\_ptr(p) \leftarrow r$ ;  $q \leftarrow r$ ;
    end;
  end;
   $cur\_height \leftarrow cur\_height + prev\_dp + width(q)$ ;  $prev\_dp \leftarrow 0$ 

```

This code is used in section 1149.

1154. 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 1825 ⟩
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 1155 ⟩;
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 1156 ⟩;
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;

```

1155. ⟨ Dispense with trivial cases of void or bad boxes 1155 ⟩ ≡

```

if v = null then
  begin vsplit ← null; return;
end;
if type(v) ≠ vlist_node then
  begin print_err(""); print_esc("vsplit"); print("needs a "); print_esc("vbox");
  help2("The box you are trying to split is an hbox.");
  ("I can't split such a box, so I'll leave it alone."); error; vsplit ← null; return;
end

```

This code is used in section 1154.

1156. 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 1156 ⟩ ≡

```
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 1827 ⟩
  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 1154.

1157. The page builder. When TeX appends new material to its main vlist in vertical mode, it uses a method something like *vsplit* to decide where a page ends, except that the calculations are done “on line” as new items come in. The main complication in this process is that insertions must be put into their boxes and removed from the vlist, in a more-or-less optimum manner.

We shall use the term “current page” for that part of the main vlist that is being considered as a candidate for being broken off and sent to the user’s output routine. The current page starts at *link(page_head)*, and it ends at *page_tail*. We have *page_head* = *page_tail* if this list is empty.

Utter chaos would reign if the user kept changing page specifications while a page is being constructed, so the page builder keeps the pertinent specifications frozen as soon as the page receives its first box or insertion. The global variable *page_contents* is *empty* when the current page contains only mark nodes and content-less whatsit nodes; it is *inserts_only* if the page contains only insertion nodes in addition to marks and whatsits. Glue nodes, kern nodes, and penalty nodes are discarded until a box or rule node appears, at which time *page_contents* changes to *box_there*. As soon as *page_contents* becomes non-*empty*, the current *vsize* and *max_depth* are squirreled away into *page_goal* and *page_max_depth*; the latter values will be used until the page has been forwarded to the user’s output routine. The *\topskip* adjustment is made when *page_contents* changes to *box_there*.

Although *page_goal* starts out equal to *vsize*, it is decreased by the scaled natural height-plus-depth of the insertions considered so far, and by the *\skip* corrections for those insertions. Therefore it represents the size into which the non-inserted material should fit, assuming that all insertions in the current page have been made.

The global variables *best_page_break* and *least_page_cost* correspond respectively to the local variables *best_place* and *least_cost* in the *vert_break* routine that we have already studied; i.e., they record the location and value of the best place currently known for breaking the current page. The value of *page_goal* at the time of the best break is stored in *best_size*.

```
define inserts_only = 1 {page_contents when an insert node has been contributed, but no boxes}
define box_there = 2 {page_contents when a box or rule has been contributed}

⟨Global variables 13⟩ +≡
page_tail: pointer; {the final node on the current page}
page_contents: empty .. box_there; {what is on the current page so far?}
page_max_depth: scaled; {maximum box depth on page being built}
best_page_break: pointer; {break here to get the best page known so far}
least_page_cost: integer; {the score for this currently best page}
best_size: scaled; {its page_goal}
```

1158. The page builder has another data structure to keep track of insertions. This is a list of four-word nodes, starting and ending at *page_ins_head*. That is, the first element of the list is node $r_1 = \text{link}(\text{page_ins_head})$; node r_j is followed by $r_{j+1} = \text{link}(r_j)$; and if there are n items we have $r_{n+1} = \text{page_ins_head}$. The *subtype* field of each node in this list refers to an insertion number; for example, '\insert 250' would correspond to a node whose *subtype* is *qi(250)* (the same as the *subtype* field of the relevant *ins_node*). These *subtype* fields are in increasing order, and $\text{subtype}(\text{page_ins_head}) = \text{qi}(255)$, so *page_ins_head* serves as a convenient sentinel at the end of the list. A record is present for each insertion number that appears in the current page.

The *type* field in these nodes distinguishes two possibilities that might occur as we look ahead before deciding on the optimum page break. If $\text{type}(r) = \text{inserting}$, then $\text{height}(r)$ contains the total of the height-plus-depth dimensions of the box and all its inserts seen so far. If $\text{type}(r) = \text{split_up}$, then no more insertions will be made into this box, because at least one previous insertion was too big to fit on the current page; *broken_ptr*(r) points to the node where that insertion will be split, if TeX decides to split it, *broken_ins*(r) points to the insertion node that was tentatively split, and *height*(r) includes also the natural height plus depth of the part that would be split off.

In both cases, *last_ins_ptr*(r) points to the last *ins_node* encountered for box *qo*(*subtype*(r)) that would be at least partially inserted on the next page; and *best_ins_ptr*(r) points to the last such *ins_node* that should actually be inserted, to get the page with minimum badness among all page breaks considered so far. We have *best_ins_ptr*(r) = *null* if and only if no insertion for this box should be made to produce this optimum page.

The data structure definitions here use the fact that the *height* field appears in the fourth word of a box node.

```
define page_ins_node_size = 4 { number of words for a page insertion node }
define inserting = 0 { an insertion class that has not yet overflowed }
define split_up = 1 { an overflowed insertion class }
define broken_ptr(#) ≡ link(# + 1) { an insertion for this class will break here if anywhere }
define broken_ins(#) ≡ info(# + 1) { this insertion might break at broken_ptr }
define last_ins_ptr(#) ≡ link(# + 2) { the most recent insertion for this subtype }
define best_ins_ptr(#) ≡ info(# + 2) { the optimum most recent insertion }

{ Initialize the special list heads and constant nodes 966 } +≡
  subtype(page_ins_head) ← qi(255); type(page_ins_head) ← split_up; link(page_ins_head) ← page_ins_head;
```

1159. 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 \lastnode type }
insert_penalties: integer; { sum of the penalties for insertions that were held over }
```

1160. ⟨ Put each of TeX's primitives into the hash table 244 ⟩ +≡

```
primitive("pagegoal", set_page_dimen, 0); primitive("pagetotal", set_page_dimen, 1);
primitive("pagestretch", set_page_dimen, 2); primitive("pagefilstretch", set_page_dimen, 3);
primitive("pagefillstretch", set_page_dimen, 4); primitive("pagefilllstretch", set_page_dimen, 5);
primitive("pageshrink", set_page_dimen, 6); primitive("pagedepth", set_page_dimen, 7);
```

1161. ⟨ Cases of *print.cmd.chr* for symbolic printing of primitives 245 ⟩ +≡

```
set_page_dimen: case chr_code of
  0: print_esc("pagegoal");
  1: print_esc("pagetotal");
  2: print_esc("pagestretch");
  3: print_esc("pagefilstretch");
  4: print_esc("pagefillstretch");
  5: print_esc("pagefilllstretch");
  6: print_esc("pageshrink");
  othercases print_esc("pagedepth")
endcases;
```

```

1162. define print_plus_end(#) ≡ print(#); end
  define print_plus(#) ≡
    if page_so_far[#] ≠ 0 then
      begin print("plus"); print_scaled(page_so_far[#]); print_plus_end
procedure print_totals;
  begin print_scaled(page_total); print_plus(2)( ""); print_plus(3)(fil"); print_plus(4)(fill);
print_plus(5)(fill1);
  if page_shrink ≠ 0 then
    begin print("minus"); print_scaled(page_shrink);
    end;
end;

```

```

1163. ⟨Show the status of the current page 1163⟩ ≡
if page_head ≠ page_tail then
  begin print_nl("###current_page:");
  if output_active then print("(held_over_for_next_output)");
show_box(link(page_head));
  if page_contents > empty then
    begin print_nl("totalheight"); print_totals; print_nl("goal_height");
print_scaled(page_goal); r ← link(page_ins_head);
    while r ≠ page_ins_head do
      begin print_ln; print_esc("insert"); t ← qo(subtype(r)); print_int(t); print("adds");
      if count(t) = 1000 then t ← height(r)
      else t ← x_over_n(height(r), 1000) * count(t);
print_scaled(t);
      if type(r) = split_up then
        begin q ← page_head; t ← 0;
        repeat q ← link(q);
          if (type(q) = ins_node) ∧ (subtype(q) = subtype(r)) then incr(t);
        until q = broken_ins(r);
        print("#"); print_int(t); print("might_split");
      end;
      r ← link(r);
    end;
  end;
end

```

This code is used in section 236.

1164. Here is a procedure that is called when the *page_contents* is changing from *empty* to *inserts_only* or *box_there*.

```

define set_page_so_far_zero(#) ≡ page_so_far[#] ← 0
procedure freeze_page_specs(s : small_number);
  begin page_contents ← s; page_goal ← vsize; page_max_depth ← max_depth; page_depth ← 0;
  do_all_six(set_page_so_far_zero); least_page_cost ← awful_bad;
  stat if tracing_pages > 0 then
    begin begin_diagnostic; print_nl("%goal_height"); print_scaled(page_goal);
print("max_depth"); print_scaled(page_max_depth); end_diagnostic(false);
    end; tats
  end;

```

1165. Pages are built by appending nodes to the current list in TeX's vertical mode, which is at the outermost level of the semantic nest. This vlist is split into two parts; the "current page" that we have been talking so much about already, and the "contribution list" that receives new nodes as they are created. The current page contains everything that the page builder has accounted for in its data structures, as described above, while the contribution list contains other things that have been generated by other parts of TeX but have not yet been seen by the page builder. The contribution list starts at *link(contrib_head)*, and it ends at the current node in TeX's vertical mode.

When TeX has appended new material in vertical mode, it calls the procedure *build_page*, which tries to catch up by moving nodes from the contribution list to the current page. This procedure will succeed in its goal of emptying the contribution list, unless a page break is discovered, i.e., unless the current page has grown to the point where the optimum next page break has been determined. In the latter case, the nodes after the optimum break will go back onto the contribution list, and control will effectively pass to the user's output routine.

We make *type(page_head) = glue_node*, so that an initial glue node on the current page will not be considered a valid breakpoint.

```
( Initialize the special list heads and constant nodes 966 ) +≡
  type(page_head) ← glue_node; subtype(page_head) ← normal;
```

1166. The global variable *output_active* is true during the time the user's output routine is driving TeX.

```
( Global variables 13 ) +≡
  output_active: boolean; { are we in the midst of an output routine? }
```

1167. (Set initial values of key variables 21) +≡

```
  output_active ← false; insert_penalties ← 0;
```

1168. 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 1168 ) ≡
  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 233 and 1194.

1169. At certain times box 255 is supposed to be void (i.e., *null*), or an insertion box is supposed to be ready to accept a vertical list. If not, an error message is printed, and the following subroutine flushes the unwanted contents, reporting them to the user.

```
procedure box_error(n : eight_bits);
  begin error; begin_diagnostic; print_nl("The following box has been deleted:");
  show_box(box(n)); end_diagnostic(true); flush_node_list(box(n)); box(n) ← null;
  end;
```

1170. The following procedure guarantees that a given box register does not contain an \hbox.

```
procedure ensure_vbox(n : eight_bits);
  var p: pointer; { the box register contents }
  begin p ← box(n);
  if p ≠ null then
    if type(p) = hlist_node then
      begin print_err("Insertions can only be added to a vbox");
      help3("Tut tut: You're trying to insert into a "
            "\box_register that now contains an \hbox.")
      ("Proceed, and I'll discard its present contents."); box_error(n);
      end;
    end;
  end;
```

1171. T_EX is not always in vertical mode at the time *build_page* is called; the current mode reflects what T_EX should return to, after the contribution list has been emptied. A call on *build_page* should be immediately followed by ‘**goto big_switch**’, which is T_EX’s central control point.

```
define contribute = 80 { go here to link a node into the current page }

{ Declare the procedure called fire_up 1189 }
procedure build_page; { append contributions to the current page }
  label exit, done, done1, continue, contribute, update_heights;
  var p: pointer; { the node being appended }
  q, r: pointer; { nodes being examined }
  b, c: integer; { badness and cost of current page }
  pi: integer; { penalty to be added to the badness }
  n: min_quarterword .. 255; { insertion box number }
  delta, h, w: scaled; { sizes used for insertion calculations }
begin if (link(contrib_head) = null) ∨ output_active then return;
repeat continue: p ← link(contrib_head);
  { Update the values of last_glue, last_penalty, and last_kern 1173 };
  { Move node p to the current page; if it is time for a page break, put the nodes following the break
    back onto the contribution list, and return to the user's output routine if there is one 1174 };
  until link(contrib_head) = null;
  { Make the contribution list empty by setting its tail to contrib_head 1172 };
exit: end;
```

1172. define *contrib_tail* ≡ *nest*[0].tail_field { tail of the contribution list }

{ Make the contribution list empty by setting its tail to *contrib_head* 1172 } ≡

```
if nest_ptr = 0 then tail ← contrib_head { vertical mode }
else contrib_tail ← contrib_head { other modes }
```

This code is used in section 1171.

1173. \langle Update the values of *last_glue*, *last_penalty*, and *last_kern* 1173 $\rangle \equiv$

```

if last_glue  $\neq$  max_halfword then delete_glue_ref(last_glue);
last_penalty  $\leftarrow$  0; last_kern  $\leftarrow$  0; last_node_type  $\leftarrow$  type(p) + 1;
if type(p) = glue_node then
begin last_glue  $\leftarrow$  glue_ptr(p); add_glue_ref(last_glue);
end
else begin last_glue  $\leftarrow$  max_halfword;
if type(p) = penalty_node then last_penalty  $\leftarrow$  penalty(p)
else if type(p) = kern_node then last_kern  $\leftarrow$  width(p);
end

```

This code is used in section 1171.

1174. The code here is an example of a many-way switch into routines that merge together in different places. Some people call this unstructured programming, but the author doesn't see much wrong with it, as long as the various labels have a well-understood meaning.

\langle Move node *p* to the current page; if it is time for a page break, put the nodes following the break back onto the contribution list, and return to the user's output routine if there is one 1174 $\rangle \equiv$

```

⟨ If the current page is empty and node p is to be deleted, goto done1; otherwise use node p to update
the state of the current page; if this node is an insertion, goto contribute; otherwise if this node is
not a legal breakpoint, goto contribute or update_heights; otherwise set pi to the penalty associated
with this breakpoint 1177 ⟩;
⟨ Check if node p is a new champion breakpoint; then if it is time for a page break, prepare for output,
and either fire up the user's output routine and return or ship out the page and goto done 1182 ⟩;
if (type(p) < glue_node)  $\vee$  (type(p) > kern_node) then goto contribute;
update_heights: ⟨ Update the current page measurements with respect to the glue or kern specified by
node p 1181 ⟩;
contribute: ⟨ Make sure that page_max_depth is not exceeded 1180 ⟩;
⟨ Link node p into the current page and goto done 1175 ⟩;
done1: ⟨ Recycle node p 1176 ⟩;
done:

```

This code is used in section 1171.

1175. \langle Link node *p* into the current page and goto *done* 1175 $\rangle \equiv$

```

link(page_tail)  $\leftarrow$  p; page_tail  $\leftarrow$  p; link(contrib_head)  $\leftarrow$  link(p); link(p)  $\leftarrow$  null; goto done

```

This code is used in section 1174.

1176. \langle Recycle node *p* 1176 $\rangle \equiv$

```

link(contrib_head)  $\leftarrow$  link(p); link(p)  $\leftarrow$  null;
if saving_vdiscards > 0 then
begin if page_disc = null then page_disc  $\leftarrow$  p else link(tail_page_disc)  $\leftarrow$  p;
tail_page_disc  $\leftarrow$  p;
end
else flush_node_list(p)

```

This code is used in section 1174.

1177. The title of this section is already so long, it seems best to avoid making it more accurate but still longer, by mentioning the fact that a kern node at the end of the contribution list will not be contributed until we know its successor.

```
<If the current page is empty and node p is to be deleted, goto done1; otherwise use node p to update the
state of the current page; if this node is an insertion, goto contribute; otherwise if this node is not a
legal breakpoint, goto contribute or update_heights; otherwise set pi to the penalty associated with
this breakpoint 1177 > ≡
case type(p) of
  hlist_node, vlist_node, rule_node: if page_contents < box_there then
    < Initialize the current page, insert the \topskip glue ahead of p, and goto continue 1178 >
    else < Prepare to move a box or rule node to the current page, then goto contribute 1179 >;
  whatsit_node: if (page_contents < box_there)  $\wedge$  ((subtype(p) = pdf_snapy_node)  $\vee$  (subtype(p) =
pdf_snapy_comp_node)) then
    begin print("snap_node_being_discarded"); goto done1;
    end
    else < Prepare to move whatsit p to the current page, then goto contribute 1611 >;
  glue_node: if page_contents < box_there then goto done1
    else if precedes_break(page_tail) then pi  $\leftarrow$  0
    else goto update_heights;
  kern_node: if page_contents < box_there then goto done1
    else if link(p) = null then return
      else if type(link(p)) = glue_node then pi  $\leftarrow$  0
      else goto update_heights;
  penalty_node: if page_contents < box_there then goto done1 else pi  $\leftarrow$  penalty(p);
  mark_node: goto contribute;
  ins_node: < Append an insertion to the current page and goto contribute 1185 >;
  othercases confusion("page")
endcases
```

This code is used in section 1174.

1178. < Initialize the current page, insert the \topskip glue ahead of *p*, and goto *continue* 1178 > ≡
begin if *page_contents* = empty then *freeze_page_specs(box_there)*
else *page_contents* \leftarrow *box_there*;
q \leftarrow *new_skip_param(top_skip_code)*; { now *temp_ptr* = *glue_ptr(q)* }
if *width(temp_ptr)* > *height(p)* then *width(temp_ptr)* \leftarrow *width(temp_ptr)* - *height(p)*
else *width(temp_ptr)* \leftarrow 0;
link(q) \leftarrow *p*; *link(contrib_head)* \leftarrow *q*; goto *continue*;
end

This code is used in section 1177.

1179. < Prepare to move a box or rule node to the current page, then goto *contribute* 1179 > ≡
begin *page_total* \leftarrow *page_total* + *page_depth* + *height(p)*; *page_depth* \leftarrow *depth(p)*; goto *contribute*;
end

This code is used in section 1177.

1180. < Make sure that *page_max_depth* is not exceeded 1180 > ≡
if *page_depth* > *page_max_depth* then
 begin *page_total* \leftarrow *page_total* + *page_depth* - *page_max_depth*;
 page_depth \leftarrow *page_max_depth*;
end;

This code is used in section 1174.

1181. ⟨Update the current page measurements with respect to the glue or kern specified by node p 1181⟩ ≡

```

if  $\text{type}(p) = \text{kern\_node}$  then  $q \leftarrow p$ 
else begin  $q \leftarrow \text{glue\_ptr}(p)$ ;
 $\text{page\_so\_far}[2 + \text{stretch\_order}(q)] \leftarrow \text{page\_so\_far}[2 + \text{stretch\_order}(q)] + \text{stretch}(q)$ ;
 $\text{page\_shrink} \leftarrow \text{page\_shrink} + \text{shrink}(q)$ ;
if ( $\text{shrink\_order}(q) \neq \text{normal}$ )  $\wedge$  ( $\text{shrink}(q) \neq 0$ ) then
begin
print_err("Infinite glue shrinkage found on current page");
help4("The page about to be output contains some infinitely")
("shrinkable glue, e.g., \vss' or \vskip\Opt minus 1fil'.")
("Such glue doesn't belong there; but you can safely proceed,")
("since the offensive shrinkability has been made finite."); error;
 $r \leftarrow \text{new\_spec}(q)$ ;
 $\text{shrink\_order}(r) \leftarrow \text{normal}$ ; delete_glue_ref( $q$ );  $\text{glue\_ptr}(p) \leftarrow r$ ;  $q \leftarrow r$ ;
end;
end;
 $\text{page\_total} \leftarrow \text{page\_total} + \text{page\_depth} + \text{width}(q)$ ;  $\text{page\_depth} \leftarrow 0$ 
```

This code is used in section 1174.

1182. ⟨Check if node p is a new champion breakpoint; then if it is time for a page break, prepare for output, and either fire up the user's output routine and return or ship out the page and goto done 1182⟩ ≡

```

if  $pi < \text{inf\_penalty}$  then
begin ⟨Compute the badness,  $b$ , of the current page, using  $\text{awful\_bad}$  if the box is too full 1184⟩;
if  $b < \text{awful\_bad}$  then
  if  $pi \leq \text{eject\_penalty}$  then  $c \leftarrow pi$ 
  else if  $b < \text{inf\_bad}$  then  $c \leftarrow b + pi + \text{insert\_penalties}$ 
    else  $c \leftarrow \text{deplorable}$ 
else  $c \leftarrow b$ ;
if  $\text{insert\_penalties} \geq 10000$  then  $c \leftarrow \text{awful\_bad}$ ;
stat if  $\text{tracing\_pages} > 0$  then ⟨Display the page break cost 1183⟩;
tats
if  $c \leq \text{least\_page\_cost}$  then
begin  $\text{best\_page\_break} \leftarrow p$ ;  $\text{best\_size} \leftarrow \text{page\_goal}$ ;  $\text{least\_page\_cost} \leftarrow c$ ;  $r \leftarrow \text{link}(\text{page\_ins\_head})$ ;
while  $r \neq \text{page\_ins\_head}$  do
  begin  $\text{best\_ins\_ptr}(r) \leftarrow \text{last\_ins\_ptr}(r)$ ;  $r \leftarrow \text{link}(r)$ ;
  end;
end;
if ( $c = \text{awful\_bad}$ )  $\vee$  ( $pi \leq \text{eject\_penalty}$ ) then
begin  $\text{fire\_up}(p)$ ; {output the current page at the best place}
  if  $\text{output\_active}$  then  $\text{return}$ ; {user's output routine will act}
  goto done; {the page has been shipped out by default output routine}
end;
end
```

This code is used in section 1174.

1183. \langle Display the page break cost 1183 $\rangle \equiv$

```

begin begin_diagnostic; print_nl("%"); print("t="); print_totals;
print("g="); print_scaled(page_goal);
print("b=");
if b = awful_bad then print_char("*") else print_int(b);
print("p="); print_int(pi); print("c=");
if c = awful_bad then print_char("*") else print_int(c);
if c ≤ least_page_cost then print_char("#");
end_diagnostic(false);
end

```

This code is used in section 1182.

1184. \langle Compute the badness, b , of the current page, using $awful_bad$ if the box is too full 1184 $\rangle \equiv$

```

if page_total < page_goal then
  if (page_so_far[3] ≠ 0) ∨ (page_so_far[4] ≠ 0) ∨ (page_so_far[5] ≠ 0) then b ← 0
  else b ← badness(page_goal – page_total, page_so_far[2])
else if page_total – page_goal > page_shrink then b ← awful_bad
  else b ← badness(page_total – page_goal, page_shrink)

```

This code is used in section 1182.

1185. \langle Append an insertion to the current page and goto contribute 1185 $\rangle \equiv$

```

begin if page_contents = empty then freeze_page_specs(inserts_only);
n ← subtype(p); r ← page_ins_head;
while n ≥ subtype(link(r)) do r ← link(r);
n ← qo(n);
if subtype(r) ≠ qi(n) then ⟨Create a page insertion node with  $subtype(r) = qi(n)$ , and include the glue
correction for box  $n$  in the current page state 1186⟩;
if type(r) = split_up then insert_penalties ← insert_penalties + float_cost(p)
else begin last_ins_ptr(r) ← p; delta ← page_goal – page_total – page_depth + page_shrink;
{ this much room is left if we shrink the maximum }
if count(n) = 1000 then h ← height(p)
else h ← x_over_n(height(p), 1000) * count(n); { this much room is needed }
if ((h ≤ 0) ∨ (h ≤ delta)) ∧ (height(p) + height(r) ≤ dimen(n)) then
  begin page_goal ← page_goal – h; height(r) ← height(r) + height(p);
  end
else ⟨Find the best way to split the insertion, and change type(r) to split_up 1187⟩;
end;
goto contribute;
end

```

This code is used in section 1177.

1186. We take note of the value of $\backslash skip n$ and the height plus depth of $\backslash box n$ only when the first $\backslash insert n$ node is encountered for a new page. A user who changes the contents of $\backslash box n$ after that first $\backslash insert n$ had better be either extremely careful or extremely lucky, or both.

```
<Create a page insertion node with subtype(r) = qi(n), and include the glue correction for box n in the
current page state 1186> ≡
begin q ← get_node(page_ins_node_size); link(q) ← link(r); link(r) ← q; r ← q; subtype(r) ← qi(n);
type(r) ← inserting; ensure_vbox(n);
if box(n) = null then height(r) ← 0
else height(r) ← height(box(n)) + depth(box(n));
best.ins_ptr(r) ← null;
q ← skip(n);
if count(n) = 1000 then h ← height(r)
else h ← x_over_n(height(r), 1000) * count(n);
page_goal ← page_goal - h - width(q);
page_so_far[2 + stretch_order(q)] ← page_so_far[2 + stretch_order(q)] + stretch(q);
page_shrink ← page_shrink + shrink(q);
if (shrink_order(q) ≠ normal) ∧ (shrink(q) ≠ 0) then
begin print_err("Infinite glue shrinkage inserted from"); print_esc("skip"); print_int(n);
help3("The correction glue for page breaking with insertions")
("must have finite shrinkability. But you may proceed, ")
("since the offensive shrinkability has been made finite."); error;
end;
end
```

This code is used in section 1185.

1187. Here is the code that will split a long footnote between pages, in an emergency. The current situation deserves to be recapitulated: Node p is an insertion into box n ; the insertion will not fit, in its entirety, either because it would make the total contents of box n greater than $\backslash dimen n$, or because it would make the incremental amount of growth h greater than the available space $delta$, or both. (This amount h has been weighted by the insertion scaling factor, i.e., by $\backslash count n$ over 1000.) Now we will choose the best way to break the vlist of the insertion, using the same criteria as in the $\backslash vsplit$ operation.

```
<Find the best way to split the insertion, and change type(r) to split_up 1187> ≡
begin if count(n) ≤ 0 then w ← max_dimen
else begin w ← page_goal - page_total - page_depth;
if count(n) ≠ 1000 then w ← x_over_n(w, count(n)) * 1000;
end;
if w > dimen(n) - height(r) then w ← dimen(n) - height(r);
q ← vert_break(ins_ptr(p), w, depth(p)); height(r) ← height(r) + best_height_plus_depth;
stat if tracing_pages > 0 then <Display the insertion split cost 1188>;
tats
if count(n) ≠ 1000 then best_height_plus_depth ← x_over_n(best_height_plus_depth, 1000) * count(n);
page_goal ← page_goal - best_height_plus_depth; type(r) ← split_up; broken_ptr(r) ← q;
broken_ins(r) ← p;
if q = null then insert_penalties ← insert_penalties + eject_penalty
else if type(q) = penalty_node then insert_penalties ← insert_penalties + penalty(q);
end
```

This code is used in section 1185.

```
1188. <Display the insertion split cost 1188> ≡
begin begin_diagnostic; print_nl("%\u00d7split"); print_int(n); print("to"); print_scaled(w);
print_char(","); print_scaled(best_height_plus_depth);
print("\u00d7p=");
if q = null then print_int(eject_penalty)
else if type(q) = penalty_node then print_int(penalty(q))
else print_char("0");
end_diagnostic(false);
end
```

This code is used in section 1187.

1189. 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* ≠ *null* implies *first_mark* ≠ *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 1189 > ≡
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 1190 >;
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 1191 >;
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 1201 >
    else < Fire up the user’s output routine and return 1202 >;
  < Perform the default output routine 1200 >;
exit: end;
```

This code is used in section 1171.

1190. *< Set the value of output_penalty 1190 >* ≡
if type(best_page_break) = penalty_node **then**
 begin geq_word_define(int_base + output_penalty_code, penalty(best_page_break));
 penalty(best_page_break) ← inf_penalty;
 end
else geq_word_define(int_base + output_penalty_code, inf_penalty)

This code is used in section 1189.

1191. 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 1191 > ≡
if c = best_page_break **then** best_page_break ← null; { *c* not yet linked in }
< Ensure that box 255 is empty before output 1192 >
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 1195 };
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 1197 };
 end
 else if type(*p*) = mark_node **then**
 if mark_class(*p*) ≠ 0 **then** { Update the current marks for fire_up 1830 }
 else { Update the values of first_mark and bot_mark 1193 };
 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 1194 };
< Delete the page-insertion nodes 1196 >

This code is used in section 1189.

1192. *< Ensure that box 255 is empty before output 1192 >* ≡

if box(255) ≠ null **then**
 begin print_err(""); print_esc("box"); print("255 is not void");
 help2("You shouldn't use \box255 except in \output routines.")
 ("Proceed, and I'll discard its present contents."); box_error(255);
end

This code is used in section 1191.

1193. *< Update the values of first_mark and bot_mark 1193 >* ≡

begin if first_mark = null **then**
 begin first_mark ← mark_ptr(*p*); add_token_ref(first_mark);
 end;
if bot_mark ≠ null **then** delete_token_ref(bot_mark);
 bot_mark ← mark_ptr(*p*); add_token_ref(bot_mark);
end

This code is used in section 1191.

1194. When the following code is executed, the current page runs from node $\text{link}(\text{page_head})$ to node prev_p , and the nodes from p to page_tail are to be placed back at the front of the contribution list. Furthermore the heldover insertions appear in a list from $\text{link}(\text{hold_head})$ to q ; we will put them into the current page list for safekeeping while the user's output routine is active. We might have $q = \text{hold_head}$; and $p = \text{null}$ if and only if $\text{prev_p} = \text{page_tail}$. Error messages are suppressed within vpackage , since the box might appear to be overfull or underfull simply because the stretch and shrink from the $\backslash\text{skip}$ registers for inserts are not actually present in the box.

```
< Break the current page at node  $p$ , put it in box 255, and put the remaining nodes on the contribution
list 1194 > ≡
if  $p \neq \text{null}$  then
begin if  $\text{link}(\text{contrib\_head}) = \text{null}$  then
  if  $\text{nest\_ptr} = 0$  then  $\text{tail} \leftarrow \text{page\_tail}$ 
  else  $\text{contrib\_tail} \leftarrow \text{page\_tail}$ ;
   $\text{link}(\text{page\_tail}) \leftarrow \text{link}(\text{contrib\_head})$ ;  $\text{link}(\text{contrib\_head}) \leftarrow p$ ;  $\text{link}(\text{prev\_p}) \leftarrow \text{null}$ ;
end;
 $\text{save\_vbadness} \leftarrow \text{vbadness}$ ;  $\text{vbadness} \leftarrow \text{inf\_bad}$ ;  $\text{save\_vfuzz} \leftarrow \text{vfuzz}$ ;  $\text{vfuzz} \leftarrow \text{max\_dimen}$ ;
{ inhibit error messages }
 $\text{box}(255) \leftarrow \text{vpackage}(\text{link}(\text{page\_head}), \text{best\_size}, \text{exactly}, \text{page\_max\_depth})$ ;  $\text{vbadness} \leftarrow \text{save\_vbadness}$ ;
 $\text{vfuzz} \leftarrow \text{save\_vfuzz}$ ;
if  $\text{last\_glue} \neq \text{max\_halfword}$  then  $\text{delete\_glue\_ref}(\text{last\_glue})$ ;
< Start a new current page 1168 >; { this sets  $\text{last\_glue} \leftarrow \text{max\_halfword}$  }
if  $q \neq \text{hold\_head}$  then
begin  $\text{link}(\text{page\_head}) \leftarrow \text{link}(\text{hold\_head})$ ;  $\text{page\_tail} \leftarrow q$ ;
end
```

This code is used in section 1191.

1195. If many insertions are supposed to go into the same box, we want to know the position of the last node in that box, so that we don't need to waste time when linking further information into it. The last_ins_ptr fields of the page insertion nodes are therefore used for this purpose during the packaging phase.

```
< Prepare all the boxes involved in insertions to act as queues 1195 > ≡
begin  $r \leftarrow \text{link}(\text{page\_ins\_head})$ ;
while  $r \neq \text{page\_ins\_head}$  do
begin if  $\text{best\_ins\_ptr}(r) \neq \text{null}$  then
  begin  $n \leftarrow \text{qo}(\text{subtype}(r))$ ;  $\text{ensure\_vbox}(n)$ ;
  if  $\text{box}(n) = \text{null}$  then  $\text{box}(n) \leftarrow \text{new\_null\_box}$ ;
   $p \leftarrow \text{box}(n) + \text{list\_offset}$ ;
  while  $\text{link}(p) \neq \text{null}$  do  $p \leftarrow \text{link}(p)$ ;
   $\text{last\_ins\_ptr}(r) \leftarrow p$ ;
  end;
   $r \leftarrow \text{link}(r)$ ;
end;
end;
```

This code is used in section 1191.

1196. < Delete the page-insertion nodes 1196 > ≡

```
 $r \leftarrow \text{link}(\text{page\_ins\_head})$ ;
while  $r \neq \text{page\_ins\_head}$  do
begin  $q \leftarrow \text{link}(r)$ ;  $\text{free\_node}(r, \text{page\_ins\_node\_size})$ ;  $r \leftarrow q$ ;
end;
 $\text{link}(\text{page\_ins\_head}) \leftarrow \text{page\_ins\_head}$ 
```

This code is used in section 1191.

1197. We will set $\text{best_ins_ptr} \leftarrow \text{null}$ and package the box corresponding to insertion node r , just after making the final insertion into that box. If this final insertion is '*split_up*', the remainder after splitting and pruning (if any) will be carried over to the next page.

```
<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 1197> ≡
begin  $r \leftarrow \text{link}(\text{page\_ins\_head})$ ;
while  $\text{subtype}(r) \neq \text{subtype}(p)$  do  $r \leftarrow \text{link}(r)$ ;
if  $\text{best\_ins\_ptr}(r) = \text{null}$  then  $\text{wait} \leftarrow \text{true}$ 
else begin  $\text{wait} \leftarrow \text{false}$ ;  $s \leftarrow \text{last\_ins\_ptr}(r)$ ;  $\text{link}(s) \leftarrow \text{ins\_ptr}(p)$ ;
if  $\text{best\_ins\_ptr}(r) = p$  then <Wrap up the box specified by node  $r$ , splitting node  $p$  if called for; set
     $\text{wait} \leftarrow \text{true}$  if node  $p$  holds a remainder after splitting 1198>
else begin while  $\text{link}(s) \neq \text{null}$  do  $s \leftarrow \text{link}(s)$ ;
     $\text{last\_ins\_ptr}(r) \leftarrow s$ ;
end;
end;
<Either append the insertion node  $p$  after node  $q$ , and remove it from the current page, or delete
    node( $p$ ) 1199>;
end
```

This code is used in section 1191.

1198. <Wrap up the box specified by node r , splitting node p if called for; set $\text{wait} \leftarrow \text{true}$ if node p holds a remainder after splitting 1198> ≡

```
begin if  $\text{type}(r) = \text{split\_up}$  then
if  $(\text{broken\_ins}(r) = p) \wedge (\text{broken\_ptr}(r) \neq \text{null})$  then
begin while  $\text{link}(s) \neq \text{broken\_ptr}(r)$  do  $s \leftarrow \text{link}(s)$ ;
 $\text{link}(s) \leftarrow \text{null}$ ;  $\text{split\_top\_skip} \leftarrow \text{split\_top\_ptr}(p)$ ;  $\text{ins\_ptr}(p) \leftarrow \text{prune\_page\_top}(\text{broken\_ptr}(r), \text{false})$ ;
if  $\text{ins\_ptr}(p) \neq \text{null}$  then
begin  $\text{temp\_ptr} \leftarrow \text{vpack}(\text{ins\_ptr}(p), \text{natural})$ ;  $\text{height}(p) \leftarrow \text{height}(\text{temp\_ptr}) + \text{depth}(\text{temp\_ptr})$ ;
 $\text{free\_node}(\text{temp\_ptr}, \text{box\_node\_size})$ ;  $\text{wait} \leftarrow \text{true}$ ;
end;
end;
 $\text{best\_ins\_ptr}(r) \leftarrow \text{null}$ ;  $n \leftarrow \text{qo}(\text{subtype}(r))$ ;  $\text{temp\_ptr} \leftarrow \text{list\_ptr}(\text{box}(n))$ ;
 $\text{free\_node}(\text{box}(n), \text{box\_node\_size})$ ;  $\text{box}(n) \leftarrow \text{vpack}(\text{temp\_ptr}, \text{natural})$ ;
end
```

This code is used in section 1197.

1199. <Either append the insertion node p after node q , and remove it from the current page, or delete
 node(p) 1199> ≡

```
 $\text{link}(\text{prev\_p}) \leftarrow \text{link}(p)$ ;  $\text{link}(p) \leftarrow \text{null}$ ;
if  $\text{wait}$  then
begin  $\text{link}(q) \leftarrow p$ ;  $q \leftarrow p$ ;  $\text{incr}(\text{insert\_penalties})$ ;
end
else begin  $\text{delete\_glue\_ref}(\text{split\_top\_ptr}(p))$ ;  $\text{free\_node}(p, \text{ins\_node\_size})$ ;
end;
 $p \leftarrow \text{prev\_p}$ 
```

This code is used in section 1197.

1200. 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 1200> ≡
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 1189.

1201. <Explain that too many dead cycles have occurred in a row 1201> ≡

```
begin print_err("Output_loop---"); print_int(dead_cycles); print("unconsecutive_dead_cycles");
help3("I've_concluded_that_your_output_is_awry;_it_never_does_a")
("shipout,so_I_m_shipping_box255_out_myself._Next_time")
("increase_maxdeadcycles_if_you_want_me_to_be_more_patient!"); error;
end
```

This code is used in section 1189.

1202. <Fire up the user's output routine and return 1202> ≡

```
begin output_active ← true; incr(dead_cycles); push_nest; mode ← -vmode;
prev_depth ← pdf_ignored_dimen; mode_line ← -line; begin_token_list(output_routine, output_text);
new_save_level(output_group); normal_paragraph; scan_left_brace; return;
end
```

This code is used in section 1189.

1203. When the user's output routine finishes, it has constructed a vlist in internal vertical mode, and T_EX will do the following:

```
<Resume the page builder after an output routine has come to an end 1203> ≡
begin if (loc ≠ null) ∨ ((token_type ≠ output_text) ∧ (token_type ≠ backed_up)) then
  (Recover from an unbalanced output routine 1204);
  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 1205};
  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 1278.

1204. *< Recover from an unbalanced output routine 1204 >* ≡
begin *print_err("Unbalanced_output_routine");*
help2("Your_sneaky_output_routine_has_problematic_{`s_and/or_{`s."})
("I_can't_handle_that_very_well;_good_luck."); error;
repeat *get_token;*
until *loc = null;*
end { loops forever if reading from a file, since $null = min_halfword \leq 0$ }

This code is used in section 1203.

1205. *< Ensure that box 255 is empty after output 1205 >* ≡
if *box(255) ≠ null* **then**
begin *print_err("Output_routine_didn't_use_all_of_"); print_esc("box"); print_int(255);*
help3("Your\\output_commands_should\\empty\\box255,")
("e.g.,_by_saying_`\\shipout\\box255'.")
("Proceed;_I'll_discard_its_present_contents."); box_error(255);
end

This code is used in section 1203.

1206. The chief executive. We come now to the *main_control* routine, which contains the master switch that causes all the various pieces of T_EX to do their things, in the right order.

In a sense, this is the grand climax of the program: It applies all the tools that we have worked so hard to construct. In another sense, this is the messiest part of the program: It necessarily refers to other pieces of code all over the place, so that a person can't fully understand what is going on without paging back and forth to be reminded of conventions that are defined elsewhere. We are now at the hub of the web, the central nervous system that touches most of the other parts and ties them together.

The structure of *main_control* itself is quite simple. There's a label called *big_switch*, at which point the next token of input is fetched using *get_x_token*. Then the program branches at high speed into one of about 100 possible directions, based on the value of the current mode and the newly fetched command code; the sum $\text{abs}(\text{mode}) + \text{cur_cmd}$ indicates what to do next. For example, the case '*vmode* + *letter*' arises when a letter occurs in vertical mode (or internal vertical mode); this case leads to instructions that initialize a new paragraph and enter horizontal mode.

The big **case** statement that contains this multiway switch has been labeled *reswitch*, so that the program can **goto** *reswitch* when the next token has already been fetched. Most of the cases are quite short; they call an "action procedure" that does the work for that case, and then they either **goto** *reswitch* or they "fall through" to the end of the **case** statement, which returns control back to *big_switch*. Thus, *main_control* is not an extremely large procedure, in spite of the multiplicity of things it must do; it is small enough to be handled by Pascal compilers that put severe restrictions on procedure size.

One case is singled out for special treatment, because it accounts for most of T_EX's activities in typical applications. The process of reading simple text and converting it into *char_node* records, while looking for ligatures and kerns, is part of T_EX's "inner loop"; the whole program runs efficiently when its inner loop is fast, so this part has been written with particular care.

1207. We shall concentrate first on the inner loop of *main_control*, deferring consideration of the other cases until later.

```

define big_switch = 60 { go here to branch on the next token of input }
define main_loop = 70 { go here to typeset a string of consecutive characters }
define main_loop_wrapup = 80 { go here to finish a character or ligature }
define main_loop_move = 90 { go here to advance the ligature cursor }
define main_loop_move_lig = 95 { same, when advancing past a generated ligature }
define main_loop_lookahead = 100 { go here to bring in another character, if any }
define main_lig_loop = 110 { go here to check for ligatures or kerning }
define append_normal_space = 120 { go here to append a normal space between words }

{ Declare action procedures for use by main_control 1221 }
{ Declare the procedure called handle_right_brace 1246 }
procedure main_control; { governs TEX's activities }
  label big_switch, reswitch, main_loop, main_loop_wrapup, main_loop_move, main_loop_move + 1,
        main_loop_move + 2, main_loop_move_lig, main_loop_lookahead, main_loop_lookahead + 1,
        main_lig_loop, main_lig_loop + 1, main_lig_loop + 2, append_normal_space, exit;
  var t: integer; { general-purpose temporary variable }
  tmp_k1, tmp_k2: pointer; { for testing whether an auto kern should be inserted }
  begin if every_job ≠ null then begin_token_list(every_job, every_job_text);
big_switch: get_x_token;
reswitch: { Give diagnostic information, if requested 1208 };
  case abs(mode) + cur_cmd of
    hmode + letter, hmode + other_char, hmode + char_given: goto main_loop;
    hmode + char_num: begin scan_char_num; cur_chr ← cur_val; goto main_loop; end;
    hmode + no_boundary: begin get_x_token;
      if (cur_cmd = letter) ∨ (cur_cmd = other_char) ∨ (cur_cmd = char_given) ∨ (cur_cmd = char_num)
        then cancel_boundary ← true;
      goto reswitch;
    end;
    hmode + spacer: if (space_factor = 1000) ∨ (pdf_adjust_interword_glue > 0) then
      goto append_normal_space
    else app_space;
    hmode + ex_space, mmode + ex_space: goto append_normal_space;
  { Cases of main_control that are not part of the inner loop 1223 }
  end; { of the big case statement }
  goto big_switch;
main_loop: { Append character cur_chr and the following characters (if any) to the current hlist in the
           current font; goto reswitch when a non-character has been fetched 1211 };
append_normal_space: { Append a normal inter-word space to the current list, then goto big_switch 1219 };
exit: end;

```

1208. When a new token has just been fetched at *big_switch*, we have an ideal place to monitor T_EX's activity.

```

{ Give diagnostic information, if requested 1208 } ≡
  if interrupt ≠ 0 then
    if OK_to_interrupt then
      begin back_input; check_interrupt; goto big_switch;
    end;
  debug if panicking then check_mem(false); gubed
  if tracing_commands > 0 then show_cur_cmd_chr

```

This code is used in section 1207.

1209. The following part of the program was first written in a structured manner, according to the philosophy that “premature optimization is the root of all evil.” Then it was rearranged into pieces of spaghetti so that the most common actions could proceed with little or no redundancy.

The original unoptimized form of this algorithm resembles the *reconstitute* procedure, which was described earlier in connection with hyphenation. Again we have an implied “cursor” between characters *cur_l* and *cur_r*. The main difference is that the *lig_stack* can now contain a charnode as well as pseudo-ligatures; that stack is now usually nonempty, because the next character of input (if any) has been appended to it. In *main_control* we have

$$cur_r = \begin{cases} character(lig_stack), & \text{if } lig_stack > null; \\ font_bchar[cur_font], & \text{otherwise;} \end{cases}$$

except when *character(lig_stack)* = *font_false_bchar[cur_font]*. Several additional global variables are needed.

⟨ Global variables 13 ⟩ +≡

```
main_f: internal_font_number; { the current font }
main_i: four_quarters; { character information bytes for cur_l }
main_j: four_quarters; { ligature/kern command }
main_k: font_index; { index into font_info }
main_p: pointer; { temporary register for list manipulation }
main_s: integer; { space factor value }
bchar: halfword; { boundary character of current font, or non_char }
false_bchar: halfword; { nonexistent character matching bchar, or non_char }
cancel_boundary: boolean; { should the left boundary be ignored? }
ins_disc: boolean; { should we insert a discretionary node? }
```

1210. The boolean variables of the main loop are normally false, and always reset to false before the loop is left. That saves us the extra work of initializing each time.

⟨ Set initial values of key variables 21 ⟩ +≡

```
ligature_present ← false; cancel_boundary ← false; lft_hit ← false; rt_hit ← false; ins_disc ← false;
```

1211. We leave the *space_factor* unchanged if $sf_code(cur_chr) = 0$; otherwise we set it equal to $sf_code(cur_chr)$, except that it should never change from a value less than 1000 to a value exceeding 1000. The most common case is $sf_code(cur_chr) = 1000$, so we want that case to be fast.

The overall structure of the main loop is presented here. Some program labels are inside the individual sections.

```

define adjust_space_factor ≡
  main_s ← sf_code(cur_chr);
  if main_s = 1000 then space_factor ← 1000
  else if main_s < 1000 then
    begin if main_s > 0 then space_factor ← main_s;
    end
    else if space_factor < 1000 then space_factor ← 1000
    else space_factor ← main_s

⟨ Append character cur_chr and the following characters (if any) to the current hlist in the current font;
  goto reswitch when a non-character has been fetched 1211 ⟩ ≡
  adjust_space_factor;
  save_tail ← null; main_f ← cur_font; bchar ← font_bchar[main_f];
  false_bchar ← font_false_bchar[main_f];
  if mode > 0 then
    if language ≠ clang then fix_language;
    fast_get_avail(lig_stack); font(lig_stack) ← main_f; cur_l ← qi(cur_chr); character(lig_stack) ← cur_l;
    cur_q ← tail; tmp_k1 ← get_auto_kern(main_f, non_char, cur_l);
    ⟨ If tmp_k1 is not null then append that kern 1217 ⟩;
  if cancel_boundary then
    begin cancel_boundary ← false; main_k ← non_address;
    end
  else main_k ← bchar_label[main_f];
  if main_k = non_address then goto main_loop_move + 2; { no left boundary processing }
  cur_r ← cur_l; cur_l ← non_char; goto main_lig_loop + 1; { begin with cursor after left boundary }

main_loop_wrapup: ⟨ Make a ligature node, if ligature_present; insert a null discretionary, if
  appropriate 1212 ⟩;
main_loop_move: ⟨ If the cursor is immediately followed by the right boundary, goto reswitch; if it's
  followed by an invalid character, goto big_switch; otherwise move the cursor one step to the right
  and goto main_lig_loop 1213 ⟩;
main_loop_lookahead: ⟨ Look ahead for another character, or leave lig_stack empty if there's none there 1215 ⟩;
main_lig_loop: ⟨ If there's a ligature/kern command relevant to cur_l and cur_r, adjust the text
  appropriately; exit to main_loop_wrapup 1216 ⟩;
main_loop_move_lig: ⟨ Move the cursor past a pseudo-ligature, then goto main_loop_lookahead or
  main_lig_loop 1214 ⟩

```

This code is used in section 1207.

1212. If *link(cur_q)* is nonnull when *wrapup* is invoked, *cur_q* points to the list of characters that were consumed while building the ligature character *cur_l*.

A discretionary break is not inserted for an explicit hyphen when we are in restricted horizontal mode. In particular, this avoids putting discretionary nodes inside of other discretionaries.

```

define pack_lig(#) ≡ { the parameter is either rt_hit or false }
  begin main_p ← new_ligature(main_f, cur_l, link(cur_q));
  if lft_hit then
    begin subtype(main_p) ← 2; lft_hit ← false;
    end;
  if # then
    if lig_stack = null then
      begin incr(subtype(main_p)); rt_hit ← false;
      end;
    if pdf_prepend_kern > 0 then tmp_k2 ← get_auto_kern(main_f, non_char, cur_l)
    else tmp_k2 ← null;
    if tmp_k2 = null then
      begin link(cur_q) ← main_p; tail ← main_p; ligature_present ← false;
      end
    else begin link(cur_q) ← tmp_k2; link(tmp_k2) ← main_p; tail ← main_p;
      ligature_present ← false;
      end
    end
end

define wrapup(#) ≡
  if cur_l < non_char then
    begin if link(cur_q) > null then
      if character(tail) = qi(hyphen_char[main_f]) then ins_disc ← true;
      if ligature_present then pack_lig(#);
      if ins_disc then
        begin ins_disc ← false;
        if mode > 0 then tail_append(new_disc);
        end;
      end
    end
```

⟨ Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1212 ⟩ ≡
wrapup(rt_hit)

This code is used in section 1211.

1213. ⟨ If the cursor is immediately followed by the right boundary, **goto** *reswitch*; if it's followed by an invalid character, **goto** *big_switch*; otherwise move the cursor one step to the right and **goto** *main_lig_loop* 1213 ⟩ ≡

```

if lig_stack = null then goto reswitch;
cur_q ← tail; cur_l ← character(lig_stack);
main_loop_move + 1: if  $\neg$ is_char_node(lig_stack) then goto main_loop_move_lig;
main_loop_move + 2: if (cur_chr < font_bc[main_f])  $\vee$  (cur_chr > font_ec[main_f]) then
  begin char_warning(main_f, cur_chr); free_avail(lig_stack); goto big_switch;
  end;
main_i ← char_info(main_f)(cur_l);
if  $\neg$ char_exists(main_i) then
  begin char_warning(main_f, cur_chr); free_avail(lig_stack); goto big_switch;
  end;
link(tail) ← lig_stack; tail ← lig_stack { main_loop_lookahead is next }
```

This code is used in section 1211.

1214. Here we are at *main_loop_move_lig*. When we begin this code we have *cur_q* = *tail* and *cur_l* = *character(lig_stack)*.

```
(Move the cursor past a pseudo-ligature, then goto main_loop_lookahead or main_lig_loop 1214) ≡
  main_p ← lig_ptr(lig_stack);
  if main_p > null then tail_append(main_p); {append a single character}
  temp_ptr ← lig_stack; lig_stack ← link(temp_ptr); free_node(temp_ptr, small_node_size);
  main_i ← char_info(main_f)(cur_l); ligature_present ← true;
  if lig_stack = null then
    if main_p > null then goto main_loop_lookahead
    else cur_r ← bchar
  else cur_r ← character(lig_stack);
  goto main_lig_loop
```

This code is used in section 1211.

1215. The result of \char can participate in a ligature or kern, so we must look ahead for it.

```
(Look ahead for another character, or leave lig_stack empty if there's none there 1215) ≡
  get_next; {set only cur_cmd and cur_chr, for speed}
  if cur_cmd = letter then goto main_loop_lookahead + 1;
  if cur_cmd = other_char then goto main_loop_lookahead + 1;
  if cur_cmd = char_given then goto main_loop_lookahead + 1;
  x_token; {now expand and set cur_cmd, cur_chr, cur_tok}
  if cur_cmd = letter then goto main_loop_lookahead + 1;
  if cur_cmd = other_char then goto main_loop_lookahead + 1;
  if cur_cmd = char_given then goto main_loop_lookahead + 1;
  if cur_cmd = char_num then
    begin scan_char_num; cur_chr ← cur_val; goto main_loop_lookahead + 1;
    end;
  if cur_cmd = no_boundary then bchar ← non_char;
  cur_r ← bchar; lig_stack ← null; goto main_lig_loop;
main_loop_lookahead + 1: adjust_space_factor; fast_get_avail(lig_stack); font(lig_stack) ← main_f;
  cur_r ← qi(cur_chr); character(lig_stack) ← cur_r;
  if cur_r = false_bchar then cur_r ← non_char {this prevents spurious ligatures}
```

This code is used in section 1211.

1216. Even though comparatively few characters have a lig/kern program, several of the instructions here count as part of TeX's inner loop, since a potentially long sequential search must be performed. For example, tests with Computer Modern Roman showed that about 40 per cent of all characters actually encountered in practice had a lig/kern program, and that about four lig/kern commands were investigated for every such character.

At the beginning of this code we have $\text{main_i} = \text{char_info}(\text{main_f})(\text{cur_l})$.

(If there's a ligature/kern command relevant to cur_l and cur_r, adjust the text appropriately; exit to main_loop_wrapup 1216) \equiv

```

tmp_k1  $\leftarrow$  get_auto_kern(main_f, cur_l, cur_r); (If tmp_k1 is not null then append that kern 1217);
if char_tag(main_i)  $\neq$  lig_tag then goto main_loop_wrapup;
if cur_r = non_char then goto main_loop_wrapup;
main_k  $\leftarrow$  lig_kern_start(main_f)(main_i); main_j  $\leftarrow$  font_info[main_k].qqqq;
if skip_byte(main_j)  $\leq$  stop_flag then goto main_lig_loop + 2;
main_k  $\leftarrow$  lig_kern_restart(main_f)(main_j);
main_lig_loop + 1: main_j  $\leftarrow$  font_info[main_k].qqqq;
main_lig_loop + 2: if next_char(main_j) = cur_r then
    if skip_byte(main_j)  $\leq$  stop_flag then (Do ligature or kern command, returning to main_lig_loop or
        main_loop_wrapup or main_loop_move 1218);
    if skip_byte(main_j) = qi(0) then incr(main_k)
    else begin if skip_byte(main_j)  $\geq$  stop_flag then goto main_loop_wrapup;
        main_k  $\leftarrow$  main_k + qo(skip_byte(main_j)) + 1;
        end;
    goto main_lig_loop + 1

```

This code is used in section 1211.

1217. *(If tmp_k1 is not null then append that kern 1217) \equiv*

```

if tmp_k1  $\neq$  null then
begin wrapup(rt.hit); { Note: wrapup might insert a null discretionary }
save_tail  $\leftarrow$  tail; { insert auto-kern before a null discretionary inserted by wrapup if appropriate }
if ( $\neg$ is_char_node(tail))  $\wedge$  (type(tail) = disc_node)  $\wedge$  (replace_count(tail) = 0)  $\wedge$  (pre_break(tail) =
    null)  $\wedge$  (post_break(tail) = null)  $\wedge$  (link(prev_tail) = tail) then
begin insert_before_tail(tmp_k1);
end
else tail_append(tmp_k1);
goto main_loop_move;
end

```

This code is used in sections 1211 and 1216.

1218. When a ligature or kern instruction matches a character, we know from *read_font_info* that the character exists in the font, even though we haven't verified its existence in the normal way.

This section could be made into a subroutine, if the code inside *main_control* needs to be shortened.

```
< Do ligature or kern command, returning to main_lig_loop or main_loop_wrapup or main_loop_move 1218 > ≡
begin if op_byte(main_j) ≥ kern_flag then
  begin wrapup(rt_hit); tail_append(new_kern(char_kern(main_f)(main_j))); goto main_loop_move;
  end;
  if cur_l = non_char then lft_hit ← true
  else if lig_stack = null then rt_hit ← true;
  check_interrupt; { allow a way out in case there's an infinite ligature loop }
  case op_byte(main_j) of
    qi(1), qi(5): begin cur_l ← rem_byte(main_j); { =:|, =:|> }
      main_i ← char_info(main_f)(cur_l); ligature_present ← true;
      end;
    qi(2), qi(6): begin cur_r ← rem_byte(main_j); { |=:, |=:> }
      if lig_stack = null then { right boundary character is being consumed }
        begin lig_stack ← new_lig_item(cur_r); bchar ← non_char;
        end
      else if is_char_node(lig_stack) then { link(lig_stack) = null }
        begin main_p ← lig_stack; lig_stack ← new_lig_item(cur_r); lig_ptr(lig_stack) ← main_p;
        end
      else character(lig_stack) ← cur_r;
      end;
    qi(3): begin cur_r ← rem_byte(main_j); { |=:| }
      main_p ← lig_stack; lig_stack ← new_lig_item(cur_r); link(lig_stack) ← main_p;
      end;
    qi(7), qi(11): begin wrapup(false); { |=:|>, |=:|>> }
      cur_q ← tail; cur_l ← rem_byte(main_j); main_i ← char_info(main_f)(cur_l);
      ligature_present ← true;
      end;
  othercases begin cur_l ← rem_byte(main_j); ligature_present ← true; { =: }
    if lig_stack = null then goto main_loop_wrapup
    else goto main_loop_move + 1;
    end
  endcases;
  if op_byte(main_j) > qi(4) then
    if op_byte(main_j) ≠ qi(7) then goto main_loop_wrapup;
    if cur_l < non_char then goto main_lig_loop;
    main_k ← bchar_label[main_f]; goto main_lig_loop + 1;
  end
```

This code is used in section 1216.

1219. The occurrence of blank spaces is almost part of TeX's inner loop, since we usually encounter about one space for every five non-blank characters. Therefore *main_control* gives second-highest priority to ordinary spaces.

When a glue parameter like `\spaceskip` is set to '0pt', we will see to it later that the corresponding glue specification is precisely *zero_glue*, not merely a pointer to some specification that happens to be full of zeroes. Therefore it is simple to test whether a glue parameter is zero or not.

```
< Append a normal inter-word space to the current list, then goto big_switch 1219 > ≡
  if space_skip = zero_glue then
    begin (Find the glue specification, main_p, for text spaces in the current font 1220);
      temp_ptr ← new_glue(main_p);
    end
  else temp_ptr ← new_param_glue(space_skip_code);
  if pdf_adjust_interword_glue > 0 then adjust_interword_glue(tail, temp_ptr);
  link(tail) ← temp_ptr; tail ← temp_ptr; goto big_switch
```

This code is used in section 1207.

1220. Having *font_glue* allocated for each text font saves both time and memory. If any of the three spacing parameters are subsequently changed by the use of `\fontdimen`, the *find_font_dimen* procedure deallocates the *font_glue* specification allocated here.

```
< Find the glue specification, main_p, for text spaces in the current font 1220 > ≡
  begin main_p ← font_glue[cur_font];
  if main_p = null then
    begin main_p ← new_spec(zero_glue); main_k ← param_base[cur_font] + space_code;
      width(main_p) ← font_info[main_k].sc; { that's space(cur_font) }
      stretch(main_p) ← font_info[main_k + 1].sc; { and space_stretch(cur_font) }
      shrink(main_p) ← font_info[main_k + 2].sc; { and space_shrink(cur_font) }
      font_glue[cur_font] ← main_p;
    end;
  end
```

This code is used in sections 1219 and 1221.

1221. < Declare action procedures for use by *main_control* 1221 > ≡

```
procedure app_space; { handle spaces when space_factor ≠ 1000 }
  var q: pointer; { glue node }
  begin if (space_factor ≥ 2000) ∧ (xspace_skip ≠ zero_glue) then q ← new_param_glue(xspace_skip_code)
  else begin if space_skip ≠ zero_glue then main_p ← space_skip
  else < Find the glue specification, main_p, for text spaces in the current font 1220 >;
    main_p ← new_spec(main_p);
    < Modify the glue specification in main_p according to the space factor 1222 >;
    q ← new_glue(main_p); glue_ref_count(main_p) ← null;
  end;
  link(tail) ← q; tail ← q;
end;
```

See also sections 1225, 1227, 1228, 1229, 1232, 1238, 1239, 1242, 1247, 1248, 1253, 1257, 1262, 1264, 1269, 1271, 1273, 1274, 1277, 1279, 1281, 1283, 1288, 1291, 1295, 1297, 1301, 1305, 1307, 1309, 1313, 1314, 1316, 1320, 1329, 1333, 1337, 1338, 1341, 1343, 1350, 1352, 1354, 1359, 1369, 1372, 1378, 1389, 1448, 1453, 1457, 1466, 1471, 1480, 1528, and 1624.

This code is used in section 1207.

1222. ⟨Modify the glue specification in *main_p* according to the space factor 1222⟩ ≡
if *space_factor* ≥ 2000 **then** *width(main_p)* ← *width(main_p)* + *extra_space(cur_font)*;
stretch(main_p) ← *xn_over_d(stretch(main_p), space_factor, 1000)*;
shrink(main_p) ← *xn_over_d(shrink(main_p), 1000, space_factor)*

This code is used in section 1221.

1223. Whew—that covers the main loop. We can now proceed at a leisurely pace through the other combinations of possibilities.

```
define any_mode(#) ≡ vmode + #, hmode + #, mmode + # { for mode-independent commands }
⟨Cases of main_control that are not part of the inner loop 1223⟩ ≡
any_mode(relax), vmode + spacer, mmode + spacer, mmode + no_boundary: do_nothing;
any_mode(ignore_spaces): begin if cur_chr = 0 then
begin (Get the next non-blank non-call token 432);
goto reswitch;
end
else begin t ← scanner_status; scanner_status ← normal; get_next; scanner_status ← t;
if cur_cs < hash_base then cur_cs ← prim_lookup(cur_cs - single_base)
else cur_cs ← prim_lookup(text(cur_cs));
if cur_cs ≠ undefined_primitive then
begin cur_cmd ← prim_eq_type(cur_cs); cur_chr ← prim_equiv(cur_cs);
cur_tok ← cs_token_flag + prim_eqtb_base + cur_cs; goto reswitch;
end;
end;
end;
vmode + stop: if its_all_over then return; { this is the only way out }
⟨Forbidden cases detected in main_control 1226⟩ any_mode(mac_param): report_illegal_case;
⟨Math-only cases in non-math modes, or vice versa 1224⟩: insert_dollar_sign;
⟨Cases of main_control that build boxes and lists 1234⟩
⟨Cases of main_control that don't depend on mode 1388⟩
⟨Cases of main_control that are for extensions to TeX 1527⟩
```

This code is used in section 1207.

1224. Here is a list of cases where the user has probably gotten into or out of math mode by mistake. TeX will insert a dollar sign and rescan the current token.

```
define non_math(#) ≡ vmode + #, hmode + #
⟨Math-only cases in non-math modes, or vice versa 1224⟩ ≡
non_math(sup_mark), non_math(sub_mark), non_math(math_char_num), non_math(math_given),
non_math(math_comp), non_math(delim_num), non_math(left_right), non_math(above),
non_math(radical), non_math(math_style), non_math(math_choice), non_math(vccenter),
non_math(non_script), non_math(mkern), non_math(limit_switch), non_math(mskip),
non_math(math_accent), mmode + endv, mmode + par_end, mmode + stop, mmode + vskip,
mmode + un_vbox, mmode + valign, mmode + hrule
```

This code is used in section 1223.

1225. ⟨Declare action procedures for use by *main_control* 1221⟩ +≡
procedure insert_dollar_sign;

```
begin back_input; cur_tok ← math_shift_token + "$"; print_err("Missing $ inserted");
help2("I've inserted a begin-math/end-math symbol since I think"
("you left one out. Proceed, with fingers crossed."); ins_error;
end;
```

1226. When erroneous situations arise, TeX usually issues an error message specific to the particular error. For example, ‘\noalign’ should not appear in any mode, since it is recognized by the *align_peek* routine in all of its legitimate appearances; a special error message is given when ‘\noalign’ occurs elsewhere. But sometimes the most appropriate error message is simply that the user is not allowed to do what he or she has attempted. For example, ‘\moveleft’ is allowed only in vertical mode, and ‘\lower’ only in non-vertical modes. Such cases are enumerated here and in the other sections referred to under ‘See also . . .’

⟨Forbidden cases detected in *main_control* 1226⟩ ≡

```
vmode + vmove, hmode + hmove, mmode + hmove, any_mode(last_item),
```

See also sections 1276, 1289, and 1322.

This code is used in section 1223.

1227. The ‘*you_cant*’ procedure prints a line saying that the current command is illegal in the current mode; it identifies these things symbolically.

⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡
procedure *you_cant*;

```
begin print_err("You\u201ccan't\u201duse\u201d"); print_cmd_chr(cur_cmd, cur_chr); print("\u201e\u201c\u201d\u201c\u201d");
print_mode(mode);
end;
```

1228. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡
procedure *report_illegal_case*;

```
begin you_cant; help4("Sorry,\u201cbut I'm not programmed to handle this case;\u201d")
("I'll just pretend that you didn't ask for it.")
("If you're in the wrong mode,\u201cyou might be able to")
("return to the right one by typing `I}\u201e\u201e\u201e` or `I$\u201e\u201e\u201e` or `I\par`.\u201d");
error;
end;
```

1229. Some operations are allowed only in privileged modes, i.e., in cases that *mode* > 0. The *privileged* function is used to detect violations of this rule; it issues an error message and returns *false* if the current *mode* is negative.

⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡
function *privileged*: boolean;
begin **if** *mode* > 0 **then** *privileged* ← true
else begin *report_illegal_case*; *privileged* ← false;
end;
end;

1230. Either \dump or \end will cause *main_control* to enter the endgame, since both of them have ‘*stop*’ as their command code.

⟨ Put each of TeX’s primitives into the hash table 244 ⟩ +≡

```
primitive("end", stop, 0);
primitive("dump", stop, 1);
```

1231. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 245 ⟩ +≡
stop: **if** *chr_code* = 1 **then** *print_esc*("dump") **else** *print_esc*("end");

1232. We don't want to leave *main_control* immediately when a *stop* command is sensed, because it may be necessary to invoke an *\output* routine several times before things really grind to a halt. (The output routine might even say '*\gdef\end{...}*', to prolong the life of the job.) Therefore *its_all_over* is *true* only when the current page and contribution list are empty, and when the last output was not a "dead cycle."

```
< Declare action procedures for use by main_control 1221 > +≡
function its_all_over: boolean; { do this when \end or \dump occurs }
label exit;
begin if privileged then
  begin if (page_head = page_tail) ∧ (head = tail) ∧ (dead_cycles = 0) then
    begin its_all_over ← true; return;
    end;
  back_input; { we will try to end again after ejecting residual material }
  tail_append(new_null_box); width(tail) ← hsize; tail_append(new_glue(fill_glue));
  tail_append(new_penalty(−'10000000000));
  build_page; { append \hbox to \hsize{} \vfill \penalty-'10000000000 }
  end;
  its_all_over ← false;
exit: end;
```

1233. Building boxes and lists. The most important parts of *main_control* are concerned with TeX's chief mission of box-making. We need to control the activities that put entries on vlists and hlists, as well as the activities that convert those lists into boxes. All of the necessary machinery has already been developed; it remains for us to "push the buttons" at the right times.

1234. As an introduction to these routines, let's consider one of the simplest cases: What happens when '\hrule' occurs in vertical mode, or '\vrule' in horizontal mode or math mode? The code in *main_control* is short, since the *scan_rule_spec* routine already does most of what is required; thus, there is no need for a special action procedure.

Note that baselineskip calculations are disabled after a rule in vertical mode, by setting *prev_depth* \leftarrow *pdf_ignored_dimen*.

```
<Cases of main_control that build boxes and lists 1234> ≡
vmode + hrule, hmode + vrule, mmode + vrule: begin tail_append(scan_rule_spec);
  if abs(mode) = vmode then prev_depth ← pdf_ignored_dimen
  else if abs(mode) = hmode then space_factor ← 1000;
  end;
```

See also sections 1235, 1241, 1245, 1251, 1268, 1270, 1272, 1275, 1280, 1282, 1287, 1290, 1294, 1300, 1304, 1308, 1312, 1315, 1318, 1328, 1332, 1336, 1340, 1342, 1345, 1349, 1353, 1358, 1368, and 1371.

This code is used in section 1223.

1235. The processing of things like \hskip and \vskip is slightly more complicated. But the code in *main_control* is very short, since it simply calls on the action routine *append_glue*. Similarly, \kern activates *append_kern*.

```
<Cases of main_control that build boxes and lists 1234> +≡
vmode + vskip, hmode + hskip, mmode + hskip, mmode + mskip: append_glue;
any_mode(kern), mmode + mkern: append_kern;
```

1236. The *hskip* and *vskip* command codes are used for control sequences like \hss and \vfil as well as for \hskip and \vskip. The difference is in the value of *cur_chr*.

```
define fil_code = 0 { identifies \hfil and \vfil }
define fill_code = 1 { identifies \hfill and \vfill }
define ss_code = 2 { identifies \hss and \vss }
define fil_neg_code = 3 { identifies \hfilneg and \vfilneg }
define skip_code = 4 { identifies \hskip and \vskip }
define mskip_code = 5 { identifies \mskip }

<Put each of TeX's primitives into the hash table 244> +≡
primitive("hskip", hskip, skip_code);
primitive("hfil", hskip, fil_code); primitive("hfill", hskip, fill_code);
primitive("hss", hskip, ss_code); primitive("hfilneg", hskip, fil_neg_code);
primitive("vskip", vskip, skip_code);
primitive("vfil", vskip, fil_code); primitive("vfill", vskip, fill_code);
primitive("vss", vskip, ss_code); primitive("vfilneg", vskip, fil_neg_code);
primitive("mskip", mskip, mskip_code);
primitive("kern", kern, explicit); primitive("mkern", mkern, mu_glue);
```

1237. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡

```

hskip: case chr_code of
  skip_code: print_esc("hskip");
  fil_code: print_esc("hfil");
  fill_code: print_esc("hfill");
  ss_code: print_esc("hss");
  othercases print_esc("hfilneg")
endcases;

vskip: case chr_code of
  skip_code: print_esc("vskip");
  fil_code: print_esc("vfil");
  fill_code: print_esc("vfill");
  ss_code: print_esc("vss");
  othercases print_esc("vfilneg")
endcases;

mskip: print_esc("mskip");
kern: print_esc("kern");
mkern: print_esc("mkern");

```

1238. All the work relating to glue creation has been relegated to the following subroutine. It does not call *build_page*, because it is used in at least one place where that would be a mistake.

⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡

```

procedure append_glue;
  var s: small_number; { modifier of skip command }
  begin s ← cur_chr;
  case s of
    fil_code: cur_val ← fil_glue;
    fill_code: cur_val ← fill_glue;
    ss_code: cur_val ← ss_glue;
    fil_neg_code: cur_val ← fil_neg_glue;
    skip_code: scan_glue(glue_val);
    mskip_code: scan_glue(mu_val);
  end; { now cur_val points to the glue specification }
  tail_append(new_glue(cur_val));
  if s ≥ skip_code then
    begin decr(glue_ref_count(cur_val));
    if s > skip_code then subtype(tail) ← mu_glue;
    end;
  end;

```

1239. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡

```

procedure append_kern;
  var s: quarterword; { subtype of the kern node }
  begin s ← cur_chr; scan_dimen(s = mu_glue, false, false); tail_append(new_kern(cur_val));
  subtype(tail) ← s;
end;

```

1240. Many of the actions related to box-making are triggered by the appearance of braces in the input. For example, when the user says ‘`\hbox to 100pt{\hlist}`’ in vertical mode, the information about the box size (100pt, *exactly*) is put onto `save_stack` with a level boundary word just above it, and `cur_group ← adjusted_hbox_group`; T_EX enters restricted horizontal mode to process the `hlist`. The right brace eventually causes `save_stack` to be restored to its former state, at which time the information about the box size (100pt, *exactly*) is available once again; a box is packaged and we leave restricted horizontal mode, appending the new box to the current list of the enclosing mode (in this case to the current list of vertical mode), followed by any vertical adjustments that were removed from the box by `hpack`.

The next few sections of the program are therefore concerned with the treatment of left and right curly braces.

1241. If a left brace occurs in the middle of a page or paragraph, it simply introduces a new level of grouping, and the matching right brace will not have such a drastic effect. Such grouping affects neither the mode nor the current list.

```
(Cases of main_control that build boxes and lists 1234) +≡
non_math(left_brace): new_save_level(simple_group);
any_mode(begin_group): new_save_level(semi_simple_group);
any_mode(end_group): if cur_group = semi_simple_group then unsave
else off_save;
```

1242. We have to deal with errors in which braces and such things are not properly nested. Sometimes the user makes an error of commission by inserting an extra symbol, but sometimes the user makes an error of omission. T_EX can't always tell one from the other, so it makes a guess and tries to avoid getting into a loop.

The `off_save` routine is called when the current group code is wrong. It tries to insert something into the user's input that will help clean off the top level.

```
(Declare action procedures for use by main_control 1221) +≡
procedure off_save;
  var p: pointer; { inserted token }
  begin if cur_group = bottom_level then <Drop current token and complain that it was unmatched 1244>
  else begin back_input; p ← get_avail; link(temp_head) ← p; print_err("Missing");
  {Prepare to insert a token that matches cur_group, and print what it is 1243};
  print(" inserted"); ins_list(link(temp_head));
  help5("I've inserted something that you may have forgotten.")
  ("(See the <inserted_text> above.)")
  ("With luck, this will get me unwedged. But if you")
  ("really didn't forget anything, try typing `2' now; then")
  ("my insertion and my current dilemma will both disappear."); error;
  end;
end;
```

1243. At this point, $link(temp_head) = p$, a pointer to an empty one-word node.

```
< Prepare to insert a token that matches cur_group, and print what it is 1243 > ≡
case cur_group of
  semi_simple_group: begin info(p) ← cs_token_flag + frozen_end_group; print_esc("endgroup");
    end;
  math_shift_group: begin info(p) ← math_shift_token + "$"; print_char("$");
    end;
  math_left_group: begin info(p) ← cs_token_flag + frozen_right; link(p) ← get_avail; p ← link(p);
    info(p) ← other_token + "."; print_esc("right.");
    end;
  othercases begin info(p) ← right_brace_token + "}"; print_char("}");
    end
  endcases
```

This code is used in section 1242.

1244. < Drop current token and complain that it was unmatched 1244 > ≡

```
begin print_err("Extra"); print_cmd_chr(cur_cmd, cur_chr);
  help1("Things are pretty mixed up, but I think the worst is over.");
  error;
end
```

This code is used in section 1242.

1245. The routine for a *right_brace* character branches into many subcases, since a variety of things may happen, depending on *cur_group*. Some types of groups are not supposed to be ended by a right brace; error messages are given in hopes of pinpointing the problem. Most branches of this routine will be filled in later, when we are ready to understand them; meanwhile, we must prepare ourselves to deal with such errors.

< Cases of *main_control* that build boxes and lists 1234 > +≡
any_mode(*right_brace*): *handle_right_brace*;

1246. < Declare the procedure called *handle_right_brace* 1246 > ≡

```
procedure handle_right_brace;
  var p, q: pointer; { for short-term use }
    d: scaled; { holds split_max_depth in insert_group }
    f: integer; { holds floating_penalty in insert_group }
  begin case cur_group of
    simple_group: unsafe;
    bottom_level: begin print_err("Too many }'s");
      help2("You've closed more groups than you opened.");
      ("Such boobooos are generally harmless, so keep going."); error;
    end;
    semi_simple_group, math_shift_group, math_left_group: extra_right_brace;
    < Cases of handle_right_brace where a right_brace triggers a delayed action 1263 >
    othercases confusion("rightbrace")
    endcases;
  end;
```

This code is used in section 1207.

1247. \langle Declare action procedures for use by *main_control* 1221 $\rangle +\equiv$

```
procedure extra_right_brace;
begin print_err("Extra } , or forgotten ");
case cur_group of
semi_simple_group: print_esc("endgroup");
math_shift_group: print_char("$");
math_left_group: print_esc("right");
end;
help5("I've deleted a group-closing symbol because it seems to be")
("spurious, as in `\$x}`. But perhaps the } is legitimate and")
("you forgot something else, as in `\\hbox{x}`. In such cases")
("the way to recover is to insert both the forgotten and the")
("deleted material, e.g., by typing `\$`."); error; incr(align_state);
end;
```

1248. 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.

\langle Declare action procedures for use by *main_control* 1221 $\rangle +\equiv$

```
procedure normal_paragraph;
begin if looseness ≠ 0 then eq_word_define(int_base + looseness_code, 0);
if hang_indent ≠ 0 then eq_word_define(dimen_base + hang_indent_code, 0);
if hang_after ≠ 1 then eq_word_define(int_base + hang_after_code, 1);
if par_shape_ptr ≠ null then eq_define(par_shape_loc, shape_ref, null);
if inter_line_penalties_ptr ≠ null then eq_define(inter_line_penalties_loc, shape_ref, null);
end;
```

1249. 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 $\geq \text{box_flag}$, where $\text{box_flag} = 2^{30}$. Codes box_flag through $\text{global_box_flag} - 1$ represent '`\setbox0`' through '`\setbox32767`'; codes global_box_flag through $\text{ship_out_flag} - 1$ represent '`\global\setbox0`' through '`\global\setbox32767`'; code ship_out_flag represents '`\shipout`'; and codes leader_flag through $\text{leader_flag} + 2$ 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 ≡ '100000000000 { 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 244 ⟩ +≡
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);
```

1250. *<Cases of print_cmd_chr for symbolic printing of primitives 245 > +≡*

```

hmove: if chr_code = 1 then print_esc("moveleft") else print_esc("moveright");
vmove: if chr_code = 1 then print_esc("raise") else print_esc("lower");
make_box: case chr_code of
  box_code: print_esc("box");
  copy_code: print_esc("copy");
  last_box_code: print_esc("lastbox");
  vsplit_code: print_esc("vsplit");
  vtop_code: print_esc("vtop");
  vtop_code + vmode: print_esc("vbox");
  othercases print_esc("hbox")
endcases;
leader_ship: if chr_code = a_leaders then print_esc("leaders")
  else if chr_code = c_leaders then print_esc("cleaders")
  else if chr_code = x_leaders then print_esc("xleaders")
  else print_esc("shipout");

```

1251. Constructions that require a box are started by calling *scan_box* with a specified context code. The *scan_box* routine verifies that a *make_box* command comes next and then it calls *begin_box*.

<Cases of main_control that build boxes and lists 1234 > +≡

```

vmode + hmove, hmode + vmove, mmode + vmove: begin t ← cur_chr; scan_normal_dimen;
  if t = 0 then scan_box(cur_val) else scan_box(-cur_val);
end;
any_mode(leader_ship): scan_box(leader_flag - a_leaders + cur_chr);
any_mode(make_box): begin_box(0);

```

1252. The global variable *cur_box* will point to a newly made box. If the box is void, we will have *cur_box* = *null*. Otherwise we will have *type(cur_box)* = *hlist_node* or *vlist_node* or *rule_node*; the *rule_node* case can occur only with leaders.

<Global variables 13 > +≡

```

cur_box: pointer; { box to be placed into its context }

```

1253. 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 1221 > +≡

```

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 1254 }
  else if box_context < ship_out_flag then { Store cur_box in a box register 1255 }
  else if cur_box ≠ null then
    if box_context > ship_out_flag then { Append a new leader node that uses cur_box 1256 }
    else ship_out(cur_box);
end;

```

1254. The global variable *adjust_tail* will be non-null if and only if the current box might include adjustments that should be appended to the current vertical list.

```
(Append box cur_box to the current list, shifted by box_context 1254) ≡
begin if cur_box ≠ null then
  begin shift_amount(cur_box) ← box_context;
  if abs(mode) = vmode then
    begin if pre-adjust_tail ≠ null then
      begin if pre-adjust_head ≠ pre-adjust_tail then append_list(pre-adjust_head)(pre-adjust_tail);
      pre-adjust_tail ← null;
      end;
    append_to_vlist(cur_box);
    if adjust_tail ≠ null then
      begin if adjust_head ≠ adjust_tail then append_list(adjust_head)(adjust_tail);
      adjust_tail ← null;
      end;
    if mode > 0 then build_page;
    end
  else begin if abs(mode) = hmode then space_factor ← 1000
  else begin p ← new_noad; math_type(nucleus(p)) ← sub_box; info(nucleus(p)) ← cur_box;
  cur_box ← p;
  end;
  link(tail) ← cur_box; tail ← cur_box;
  end;
  end;
end
```

This code is used in section 1253.

1255. ⟨Store *cur_box* in a box register 1255⟩ ≡

```
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 1253.

1256. ⟨Append a new leader node that uses *cur_box* 1256⟩ ≡

```
begin ⟨Get the next non-blank non-relax non-call token 430⟩;
if ((cur_cmd = hskip) ∧ (abs(mode) ≠ vmode)) ∨ ((cur_cmd = vskip) ∧ (abs(mode) = vmode)) then
  begin append_glue; subtype(tail) ← box_context − (leader_flag − a_leaders);
  leader_ptr(tail) ← cur_box;
  end
else begin print_err("Leaders not followed by proper glue");
  help3("You should say `leaders<box or rule><hskip or vskip>' .")
  ("I found the <box or rule>, but there's no suitable")
  ("<hskip or vskip>, so I'm ignoring these leaders."); back_error; flush_node_list(cur_box);
  end;
end
```

This code is used in section 1253.

1257. 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 1221 > +≡
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 1258 };
vsplit_code: { Split off part of a vertical box, make cur_box point to it 1260 };
othercases { Initiate the construction of an hbox or vbox, then return 1261 }
endcases;
box_end(box_context); { in simple cases, we use the box immediately }
exit: end;
```

1258. Note that the condition $\neg \text{is_char_node}(\text{tail})$ implies that $\text{head} \neq \text{tail}$, since head is a one-word node.

```

define fetch_effective_tail_eTeX(#) ≡ { extract tx, drop \begin{M} \end{M} pair }
  q ← head; p ← null;
  repeat r ← p; p ← q; fm ← false;
    if  $\neg \text{is\_char\_node}(q)$  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)  $\wedge$  (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 1258 ≡
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)  $\wedge$  (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  $\neg \text{is\_char\_node}(tx)$  then
    if (type(tx) = hlist_node)  $\vee$  (type(tx) = vlist_node) then
      ⟨ Remove the last box, unless it's part of a discretionary 1259 ⟩;
    done: end;
end
```

This code is used in section 1257.

1259. ⟨ Remove the last box, unless it's part of a discretionary 1259 ⟩ ≡

```

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

This code is used in section 1258.

1260. Here we deal with things like ‘\vsplit 13 to 100pt’.

```
< Split off part of a vertical box, make cur_box point to it 1260 > ≡
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 1257.

1261. Here is where we enter restricted horizontal mode or internal vertical mode, in order to make a box.

```
< Initiate the construction of an hbox or vbox, then return 1261 > ≡
begin k ← cur_chr – vtop_code; saved(0) ← box_context;
if k = hmode then
  if (box_context < box_flag) ∧ (abs(mode) = vmode) then scan_spec(adjusted_hbox_group, true)
  else scan_spec(hbox_group, true)
else begin if k = vmode then scan_spec(vbox_group, true)
  else begin scan_spec(vtop_group, true); k ← vmode;
  end;
normal_paragraph;
end;
push_nest; mode ← –k;
if k = vmode then
  begin prev_depth ← pdf_ignored_dimen;
    if every_vbox ≠ null then begin_token_list(every_vbox, every_vbox_text);
  end
else begin space_factor ← 1000;
  if every_hbox ≠ null then begin_token_list(every_hbox, every_hbox_text);
end;
return;
end
```

This code is used in section 1257.

1262. < Declare action procedures for use by *main_control* 1221 > +≡

```
procedure scan_box(box_context : integer); { the next input should specify a box or perhaps a rule }
begin < Get the next non-blank non-relax non-call token 430 >;
if cur_cmd = make_box then begin_box(box_context)
else if (box_context ≥ leader_flag) ∧ ((cur_cmd = hrule) ∨ (cur_cmd = vrule)) then
  begin cur_box ← scan_rule_spec; box_end(box_context);
  end
else begin
  print_err("A <box> was supposed to be here");
  help3("I was expecting to see \\hbox or \\vbox or \\copy or \\box or"
  ("something like that. So you might find something missing in")
  ("your output. But keep trying; you can fix this later."); back_error;
  end;
end;
```

1263. When the right brace occurs at the end of an `\hbox` or `\vbox` or `\vtop` construction, the *package* routine comes into action. We might also have to finish a paragraph that hasn't ended.

```
(Cases of handle_right_brace where a right_brace triggers a delayed action 1263) ≡
hbox_group: package(0);
adjusted_hbox_group: begin adjust_tail ← adjust_head; pre_adjust_tail ← pre_adjust_head; package(0);
end;
vbox_group: begin end_graf; package(0);
end;
vtop_group: begin end_graf; package(vtop_code);
end;
```

See also sections 1278, 1296, 1310, 1311, 1346, 1351, and 1364.

This code is used in section 1246.

1264. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡

```
procedure package(c : small_number);
  var h: scaled; {height of box}
  p: pointer; {first node in a box}
  d: scaled; {max depth}
  begin d ← box_max_depth; unsafe; save_ptr ← save_ptr - 3;
  if mode = -hmode then cur_box ← hpack(link(head), saved(2), saved(1))
  else begin cur_box ← vpackage(link(head), saved(2), saved(1), d);
    if c = vtop_code then ⟨ Readjust the height and depth of cur_box, for \vtop 1265 ⟩;
  end;
  pop_nest; box_end(saved(0));
end;
```

1265. The height of a '`\vtop`' box is inherited from the first item on its list, if that item is an *hlist_node*, *vlist_node*, or *rule_node*; otherwise the `\vtop` height is zero.

```
(Readjust the height and depth of cur_box, for \vtop 1265) ≡
  begin h ← 0; p ← list_ptr(cur_box);
  if p ≠ null then
    if type(p) ≤ rule_node then h ← height(p);
    depth(cur_box) ← depth(cur_box) - h + height(cur_box); height(cur_box) ← h;
  end
```

This code is used in section 1264.

1266. Here is a really small patch to add a new primitive called `\quitvmode`. In vertical modes, it is identical to `\indent`, but in horizontal and math modes it is really a no-op (as opposed to `\indent`, which executes the *indent_in_hmode* procedure).

A paragraph begins when horizontal-mode material occurs in vertical mode, or when the paragraph is explicitly started by '`\quitvmode`', '`\indent`' or '`\noindent`'.

```
(Put each of TeX's primitives into the hash table 244) +≡
  primitive("\indent", start_par, 1); primitive("\noindent", start_par, 0);
  primitive("\quitvmode", start_par, 2);
```

1267. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 245 ⟩ +≡

```
start_par: if chr_code = 0 then print_esc("\noindent") else if chr_code = 1 then
  print_esc("\indent") else print_esc("\quitvmode");
```

1268. \langle Cases of *main_control* that build boxes and lists 1234 $\rangle + \equiv$
vmode + *start_par*: *new_graf* (*cur_chr* > 0);
vmode + *letter*, *vmode* + *other_char*, *vmode* + *char_num*, *vmode* + *char_given*, *vmode* + *math_shift*,
vmode + *un_hbox*, *vmode* + *vrule*, *vmode* + *accent*, *vmode* + *discretionary*, *vmode* + *hskip*,
vmode + *valign*, *vmode* + *ex_space*, *vmode* + *no_boundary*:
begin *back_input*; *new_graf* (true);
end;

1269. \langle Declare action procedures for use by *main_control* 1221 $\rangle + \equiv$
function *norm_min* (*h* : integer): *small_number*;
begin *if* *h* \leq 0 **then** *norm_min* \leftarrow 1 **else if** *h* \geq 63 **then** *norm_min* \leftarrow 63 **else** *norm_min* \leftarrow *h*;
end;
procedure *new_graf* (*indented* : boolean);
begin *prev_graf* \leftarrow 0;
if (*mode* = *vmode*) \vee (*head* \neq *tail*) **then** *tail_append* (*new_param_glue* (*par_skip_code*));
push_nest; *mode* \leftarrow *hmode*; *space_factor* \leftarrow 1000; *set_cur_lang*; *clang* \leftarrow *cur_lang*;
prev_graf \leftarrow (*norm_min* (*left_hyphen_min*) * '100 + *norm_min* (*right_hyphen_min*)) * '200000 + *cur_lang*;
if *indented* **then**
begin *tail* \leftarrow *new_null_box*; *link* (*head*) \leftarrow *tail*; *width* (*tail*) \leftarrow *par_indent*; **end**;
if *every_par* \neq null **then** *begin_token_list* (*every_par*, *every_par.text*);
if *nest_ptr* = 1 **then** *build_page*; { put *par_skip* glue on current page }
end;

1270. \langle Cases of *main_control* that build boxes and lists 1234 $\rangle + \equiv$
hmode + *start_par*, *memode* + *start_par*: **if** *cur_chr* \neq 2 **then** *indent_in_hmode*;

1271. \langle Declare action procedures for use by *main_control* 1221 $\rangle + \equiv$
procedure *indent_in_hmode*;
var *p, q*: pointer;
begin *if* *cur_chr* > 0 **then** { \indent }
begin *p* \leftarrow *new_null_box*; *width* (*p*) \leftarrow *par_indent*;
if *abs(mode)* = *hmode* **then** *space_factor* \leftarrow 1000
else begin *q* \leftarrow *new_noad*; *math_type* (*nucleus* (*q*)) \leftarrow *sub_box*; *info* (*nucleus* (*q*)) \leftarrow *p*; *p* \leftarrow *q*;
end;
tail_append (*p*);
end;
end;

1272. A paragraph ends when a *par_end* command is sensed, or when we are in horizontal mode when reaching the right brace of vertical-mode routines like *\vbox*, *\insert*, or *\output*.

\langle Cases of *main_control* that build boxes and lists 1234 $\rangle + \equiv$
vmode + *par_end*: **begin** *normal_paragraph*;
if *mode* > 0 **then** *build_page*;
end;
hmode + *par_end*: **begin** **if** *align_state* < 0 **then** *off_save*;
{ this tries to recover from an alignment that didn't end properly }
end_graf; { this takes us to the enclosing mode, if *mode* > 0 }
if *mode* = *vmode* **then** *build_page*;
end;
hmode + *stop*, *hmode* + *vskip*, *hmode* + *hrule*, *hmode* + *un_vbox*, *hmode* + *halign*: *head_for_vmode*;

1273. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡

```

procedure head_for_vmode;
begin if mode < 0 then
  if cur_cmd ≠ hrule then off_save
  else begin print_err("You can't use `"); print_esc("hrule");
    print("` here except with leaders");
    help2("To put a horizontal rule in an hbox or an alignment,");
    ("you should use \leaders or \hrulefill (see The TeXbook)."); error;
  end
else begin back_input; cur_tok ← par_token; back_input; token_type ← inserted;
  end;
end;
```

1274. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡

```

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;
```

1275. Insertion and adjustment and mark nodes are constructed by the following pieces of the program.
⟨ Cases of *main_control* that build boxes and lists 1234 ⟩ +≡
any_mode(insert), *hmode + vadjust*, *mmode + vadjust*: *begin_insert_or_adjust*;
any_mode(mark): *make_mark*;

1276. ⟨ Forbidden cases detected in *main_control* 1226 ⟩ +≡
vmode + vadjust,

1277. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡

```

procedure begin_insert_or_adjust;
begin if cur_cmd = vadjust then cur_val ← 255
else begin scan_eight_bit_int;
  if cur_val = 255 then
    begin print_err("You can't "); print_esc("insert"); print_int(255);
    help1("I'm changing to \insert0; box 255 is special."); error; cur_val ← 0;
  end;
  end;
  saved(0) ← cur_val;
  if (cur_cmd = vadjust) ∧ scan_keyword("pre") then saved(1) ← 1
  else saved(1) ← 0;
  save_ptr ← save_ptr + 2; new_save_level(insert_group); scan_left_brace; normal_paragraph; push_nest;
  mode ← -vmode; prev_depth ← pdf_ignored_dimen;
end;
```

1278. ⟨Cases of `handle_right_brace` where a `right_brace` triggers a delayed action 1263⟩ +≡
`insert_group: begin end_graf; q ← split_top_skip; add_glue_ref(q); d ← split_max_depth;`
`f ← floating_penalty; unsave; save_ptr ← save_ptr - 2;`
`{ now saved(0) is the insertion number, or 255 for vadjust }`
`p ← vpack(link(head), natural); pop_nest;`
`if saved(0) < 255 then`
`begin tail_append(get_node(ins_node_size)); type(tail) ← ins_node; subtype(tail) ← qi(saved(0));`
`height(tail) ← height(p) + depth(p); ins_ptr(tail) ← list_ptr(p); split_top_ptr(tail) ← q;`
`depth(tail) ← d; float_cost(tail) ← f;`
`end`
`else begin tail_append(get_node(small_node_size)); type(tail) ← adjust_node;`
`adjust_pre(tail) ← saved(1); { the subtype is used for adjust_pre }`
`adjust_ptr(tail) ← list_ptr(p); delete_glue_ref(q);`
`end;`
`free_node(p, box_node_size);`
`if nest_ptr = 0 then build_page;`
`end;`
`output_group: ⟨ Resume the page builder after an output routine has come to an end 1203 ⟩;`

1279. ⟨Declare action procedures for use by `main_control` 1221⟩ +≡

```
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;
```

1280. Penalty nodes get into a list via the `break_penalty` command.

⟨Cases of `main_control` that build boxes and lists 1234⟩ +≡
`any_mode(break_penalty): append_penalty;`

1281. ⟨Declare action procedures for use by `main_control` 1221⟩ +≡

```
procedure append_penalty;
begin scan_int; tail_append(new_penalty(cur_val));
if mode = vmode then build_page;
end;
```

1282. The `remove_item` command removes a penalty, kern, or glue node if it appears at the tail of the current list, using a brute-force linear scan. Like `\lastbox`, this command is not allowed in vertical mode (except internal vertical mode), since the current list in vertical mode is sent to the page builder. But if we happen to be able to implement it in vertical mode, we do.

⟨Cases of `main_control` that build boxes and lists 1234⟩ +≡
`any_mode(remove_item): delete_last;`

1283. 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 1221 > +≡
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 1284 }
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;
```

1284. { Apologize for inability to do the operation now, unless \unskip follows non-glue 1284 } ≡

```
begin if (cur_chr ≠ glue_node) ∨ (last_glue ≠ max_halfword) then
  begin you_cant; help2("Sorry... I usually can't take things from the current page.")
    ("Try \vskip-\lastskip instead.");
  if cur_chr = kern_node then help_line[0] ← ("Try \kern-\lastkern instead.")
  else if cur_chr ≠ glue_node then
    help_line[0] ← ("Perhaps you can make the output routine do it.");
  error;
end;
end
```

This code is used in section 1283.

1285. { Put each of TeX's primitives into the hash table 244 } +≡

```
primitive("unpenalty", remove_item, penalty_node);
primitive("unkern", remove_item, kern_node);
primitive("unskip", remove_item, glue_node);
primitive("unhbox", un_hbox, box_code);
primitive("unhcopy", un_hbox, copy_code);
primitive("unvbox", un_vbox, box_code);
primitive("unvcopy", un_vbox, copy_code);
```

1286. { Cases of print_cmd_chr for symbolic printing of primitives 245 } +≡

```
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 1862 }
  else print_esc("unvbox");
```

1287. The *un_hbox* and *un_vbox* commands unwrap one of the 256 current boxes.

```
{ Cases of main_control that build boxes and lists 1234 } +≡
vmode + un_vbox, hmode + un_hbox, mmode + un_hbox: unpackage;
```

1288. \langle Declare action procedures for use by *main_control* 1221 $\rangle +\equiv$

```

procedure unpackage;
  label done, exit;
  var p: pointer; { the box }
    r: pointer; { to remove marginal kern nodes }
    c: box_code .. copy_code; { should we copy? }
  begin if cur_chr > copy_code then {Handle saved items and goto done 1863};
    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
  begin r ← link(tail);
  if ¬is_char_node(r) ∧ (type(r) = margin_kern_node) then
    begin link(tail) ← link(r); free_avail(margin_char(r)); free_node(r, margin_kern_node_size);
    end;
  tail ← link(tail);
  end;
exit: end;
```

1289. \langle Forbidden cases detected in *main_control* 1226 $\rangle +\equiv$
vmode + italic_corr,

1290. Italic corrections are converted to kern nodes when the *italic_corr* command follows a character. In math mode the same effect is achieved by appending a kern of zero here, since italic corrections are supplied later.

\langle Cases of *main_control* that build boxes and lists 1234 $\rangle +\equiv$
hmode + italic_corr: append_italic_correction;
mmode + italic_corr: tail_append(new_kern(0));

1291. \langle Declare action procedures for use by *main_control* 1221 $\rangle +\equiv$

```

procedure append_italic_correction;
  label exit;
  var p: pointer; { char_node at the tail of the current list }
    f: internal_font_number; { the font in the char_node }
  begin if tail ≠ head then
    begin if is_char_node(tail) then p ← tail
    else if type(tail) = ligature_node then p ← lig_char(tail)
      else return;
    f ← font(p); tail_append(new_kern(char_italic(f)(char_info(f)(character(p))))));
    subtype(tail) ← explicit;
  end;
exit: end;
```

1292. Discretionary nodes are easy in the common case '\-', but in the general case we must process three braces full of items.

`<Put each of TeX's primitives into the hash table 244> +≡
primitive("-", discretionary, 1); primitive("discretionary", discretionary, 0);`

1293. `<Cases of print_cmd_chr for symbolic printing of primitives 245> +≡
discretionary: if chr_code = 1 then print_esc("-") else print_esc("discretionary");`

1294. `<Cases of main_control that build boxes and lists 1234> +≡
hmode + discretionary, mmode + discretionary: append_discretionary;`

1295. The space factor does not change when we append a discretionary node, but it starts out as 1000 in the subsidiary lists.

`<Declare action procedures for use by main_control 1221> +≡
procedure append_discretionary;
var c: integer; { hyphen character }
app_kern, pre_kern, p, c_node: pointer;
begin tail_append(new_disc);
if cur_chr = 1 then
begin c ← hyphen_char[cur_font];
if c ≥ 0 then
if c < 256 then
begin pre_kern ← get_auto_kern(cur_font, non_char, c);
app_kern ← get_auto_kern(cur_font, c, non_char); c_node ← new_character(cur_font, c);
if (app_kern = null) ∧ (pre_kern = null) then { no auto-kern }
pre_break(tail) ← c_node
else begin if pre_kern = null then pre_break(tail) ← c_node
else begin pre_break(tail) ← pre_kern; link(pre_kern) ← c_node;
end;
if app_kern ≠ null then link(c_node) ← app_kern;
end;
end;
end;
end
else begin incr(save_ptr); saved(-1) ← 0; new_save_level(disc_group); scan_left_brace; push_nest;
mode ← -hmode; space_factor ← 1000;
end;
end;`

1296. The three discretionary lists are constructed somewhat as if they were hboxes. A subroutine called `build_discretionary` handles the transitions. (This is sort of fun.)

`<Cases of handle_right_brace where a right_brace triggers a delayed action 1263> +≡
disc_group: build_discretionary;`

1297. *< Declare action procedures for use by `main_control` 1221 > +≡*

```

procedure build_discretionary;
  label done, exit;
  var p, q: pointer; { for link manipulation }
  n: integer; { length of discretionary list }
begin unsave;
  { Prune the current list, if necessary, until it contains only char_node, kern_node, hlist_node, vlist_node, rule_node, and ligature_node items; set n to the length of the list, and set q to the list's tail 1299 };
  p ← link(head); pop_nest;
  case saved(-1) of
    0: pre_break(tail) ← p;
    1: post_break(tail) ← p;
    2: { Attach list p to the current list, and record its length; then finish up and return 1298 };
  end; { there are no other cases }
  incr(saved(-1)); new_save_level(disc_group); scan_left_brace; push_nest; mode ← -hmode;
  space_factor ← 1000;
exit: end;
```

1298. *< Attach list `p` to the current list, and record its length; then finish up and return 1298 > ≡*

```

begin if (n > 0) ∧ (abs(mode) = mmode) then
  begin print_err("Illegal_math"); print_esc("discretionary");
  help2("Sorry: The third part of a discretionary break must be")
  ("empty, in math formulas. I had to delete your third part."); flush_node_list(p); n ← 0;
  error;
  end
else link(tail) ← p;
if n ≤ max_quarterword then replace_count(tail) ← n
else begin print_err("Discretionary list is too long");
  help2("Wow---I never thought anybody would tweak me here.")
  ("You can't seriously need such a huge discretionary list?"); error;
end;
if n > 0 then tail ← q;
decr(save_ptr); return;
end
```

This code is used in section 1297.

1299. During this loop, $p = \text{link}(q)$ and there are n items preceding p .

```
< Prune the current list, if necessary, until it contains only char_node, kern_node, hlist_node, vlist_node,  

  rule_node, and ligature_node items; set  $n$  to the length of the list, and set  $q$  to the list's tail 1299 > ≡  

 $q \leftarrow \text{head}; p \leftarrow \text{link}(q); n \leftarrow 0;$   

while  $p \neq \text{null}$  do  

  begin if  $\neg \text{is\_char\_node}(p)$  then  

    if  $\text{type}(p) > \text{rule\_node}$  then  

      if  $\text{type}(p) \neq \text{kern\_node}$  then  

        if  $\text{type}(p) \neq \text{ligature\_node}$  then  

          begin print_err("Improper_discretionary_list");  

          help1("Discretionary_lists_must_contain_only_boxes_and_kerns.");  

          error; begin_diagnostic;  

          print_nl("The_following_discretionary_sublist_has_been_deleted:"); show_box(p);  

          end_diagnostic(true); flush_node_list(p); link(q) ← null; goto done;  

        end;  

         $q \leftarrow p; p \leftarrow \text{link}(q); \text{incr}(n);$   

      end;  

  done:
```

This code is used in section 1297.

1300. We need only one more thing to complete the horizontal mode routines, namely the \accent primitive.

```
< Cases of main_control that build boxes and lists 1234 > +≡  

hmode + accent: make Accent;
```

1301. The positioning of accents is straightforward but tedious. Given an accent of width a , designed for characters of height x and slant s ; and given a character of width w , height h , and slant t : We will shift the accent down by $x - h$, and we will insert kern nodes that have the effect of centering the accent over the character and shifting the accent to the right by $\delta = \frac{1}{2}(w - a) + h \cdot t - x \cdot s$. If either character is absent from the font, we will simply use the other, without shifting.

```
< Declare action procedures for use by main_control 1221 > +≡  

procedure make Accent;  

  var  $s, t$ : real; { amount of slant }  

   $p, q, r$ : pointer; { character, box, and kern nodes }  

   $f$ : internal_font_number; { relevant font }  

   $a, h, x, w, delta$ : scaled; { heights and widths, as explained above }  

   $i$ : four_quarters; { character information }  

  begin scan_char_num;  $f \leftarrow \text{cur\_font}$ ;  $p \leftarrow \text{new\_character}(f, \text{cur\_val})$ ;  

  if  $p \neq \text{null}$  then  

    begin  $x \leftarrow \text{x\_height}(f)$ ;  $s \leftarrow \text{slant}(f)/\text{float\_constant}(65536)$ ;  

     $a \leftarrow \text{char\_width}(f)(\text{char\_info}(f)(\text{character}(p)))$ ;  

    do_assignments;  

    < Create a character node  $q$  for the next character, but set  $q \leftarrow \text{null}$  if problems arise 1302 >;  

    if  $q \neq \text{null}$  then < Append the accent with appropriate kerns, then set  $p \leftarrow q$  1303 >;  

     $\text{link}(\text{tail}) \leftarrow p$ ;  $\text{tail} \leftarrow p$ ;  $\text{space\_factor} \leftarrow 1000$ ;  

    end;  

  end;
```

1302. \langle Create a character node q for the next character, but set $q \leftarrow null$ if problems arise 1302 $\rangle \equiv$

```

 $q \leftarrow null; f \leftarrow cur\_font;$ 
 $\text{if } (cur\_cmd = letter) \vee (cur\_cmd = other\_char) \vee (cur\_cmd = char\_given) \text{ then}$ 
 $q \leftarrow new\_character(f, cur\_chr)$ 
 $\text{else if } cur\_cmd = char\_num \text{ then}$ 
 $\quad \begin{array}{l} \text{begin } scan\_char\_num; q \leftarrow new\_character(f, cur\_val); \\ \text{end} \end{array}$ 
 $\text{else } back\_input$ 

```

This code is used in section 1301.

1303. The kern nodes appended here must be distinguished from other kerns, lest they be wiped away by the hyphenation algorithm or by a previous line break.

The two kerns are computed with (machine-dependent) *real* arithmetic, but their sum is machine-independent; the net effect is machine-independent, because the user cannot remove these nodes nor access them via `\lastkern`.

\langle Append the accent with appropriate kerns, then set $p \leftarrow q$ 1303 $\rangle \equiv$

```

 $\begin{array}{l} \text{begin } t \leftarrow slant(f)/float\_constant(65536); i \leftarrow char\_info(f)(character(q)); w \leftarrow char\_width(f)(i); \\ h \leftarrow char\_height(f)(height\_depth(i)); \end{array}$ 
 $\text{if } h \neq x \text{ then } \{ \text{the accent must be shifted up or down} \}$ 
 $\quad \begin{array}{l} \text{begin } p \leftarrow hpack(p, natural); shift\_amount(p) \leftarrow x - h; \\ \text{end}; \end{array}$ 
 $\quad delta \leftarrow round((w - a)/float\_constant(2) + h * t - x * s); r \leftarrow new\_kern(delta); subtype(r) \leftarrow acc\_kern;$ 
 $\quad link(tail) \leftarrow r; link(r) \leftarrow p; tail \leftarrow new\_kern(-a - delta); subtype(tail) \leftarrow acc\_kern; link(p) \leftarrow tail;$ 
 $\quad p \leftarrow q;$ 
 $\quad \text{end}$ 

```

This code is used in section 1301.

1304. When ‘\cr’ or ‘\span’ or a tab mark comes through the scanner into *main_control*, it might be that the user has foolishly inserted one of them into something that has nothing to do with alignment. But it is far more likely that a left brace or right brace has been omitted, since *get_next* takes actions appropriate to alignment only when ‘\cr’ or ‘\span’ or tab marks occur with *align_state* = 0. The following program attempts to make an appropriate recovery.

\langle Cases of *main_control* that build boxes and lists 1234 $\rangle + \equiv$

```

any_mode(car_ret), any_mode(tab_mark): align_error;
any_mode(no_align): no_align_error;
any_mode(omit): omit_error;

```

1305. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡

```

procedure align_error;
begin if abs(align_state) > 2 then
  ⟨ Express consternation over the fact that no alignment is in progress 1306 ⟩
else begin back_input;
  if align_state < 0 then
    begin print_err("Missing { inserted"); incr(align_state); cur_tok ← left_brace_token + "{";
    end
  else begin print_err("Missing } inserted"); decr(align_state); cur_tok ← right_brace_token + "}";
    end;
  help3("I've put in what seems to be necessary to fix")
    ("the current column of the current alignment.")
    ("Try to go on, since this might almost work."); ins_error;
  end;
end;
```

1306. ⟨ Express consternation over the fact that no alignment is in progress 1306 ⟩ ≡

```

begin print_err("Misplaced ");
print_cmd_chr(cur_cmd, cur_chr);
if cur_tok = tab_token + "&" then
  begin help6("I can't figure out why you would want to use a tab mark")
    ("here. If you just want an ampersand, the remedy is")
    ("simple: Just type ` I & ` now. But if some right brace")
    ("up above has ended a previous alignment prematurely,")
    ("you're probably due for more error messages, and you")
    ("might try typing ` S ` now just to see what is salvageable.");
  end
else begin help5("I can't figure out why you would want to use a tab mark")
  ("or \cr or \span just now. If something like a right brace")
  ("up above has ended a previous alignment prematurely,")
  ("you're probably due for more error messages, and you")
  ("might try typing ` S ` now just to see what is salvageable.");
  end;
error;
end
```

This code is used in section 1305.

1307. The help messages here contain a little white lie, since `\noalign` and `\omit` are allowed also after `\noalign{...}`.

```

⟨ Declare action procedures for use by main_control 1221 ⟩ +≡
procedure no_align_error;
begin print_err("Misplaced ");
print_esc("noalign");
help2("I expect to see \noalign only after the \cr of")
  ("an alignment. Proceed, and I'll ignore this case."); error;
end;

procedure omit_error;
begin print_err("Misplaced ");
print_esc("omit");
help2("I expect to see \omit only after tab marks or the \cr of")
  ("an alignment. Proceed, and I'll ignore this case."); error;
end;
```

1308. 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 1234 > +≡
vmode + halign: init_align;
hmode + valign: < Cases of main_control for hmode + valign 1703 >
    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;
```

1309. An `align_group` code is supposed to remain on the `save_stack` during an entire alignment, until `fin_align` removes it.

A devious user might force an `endv` command to occur just about anywhere; we must defeat such hacks.

```
< Declare action procedures for use by main_control 1221 > +≡
procedure do_endv;
begin base_ptr ← input_ptr; input_stack[base_ptr] ← cur_input;
while (input_stack[base_ptr].index_field ≠ v_template) ∧ (input_stack[base_ptr].loc_field =
    null) ∧ (input_stack[base_ptr].state_field = token_list) do decr(base_ptr);
if (input_stack[base_ptr].index_field ≠ v_template) ∨ (input_stack[base_ptr].loc_field ≠
    null) ∨ (input_stack[base_ptr].state_field ≠ token_list) then
    fatal_error("(interwoven_alignment_preambles_are_not_allowed)");
if cur_group = align_group then
    begin end_graf;
    if fin_col then fin_row;
    end
else off_save;
end;
```

1310. < Cases of handle_right_brace where a `right_brace` triggers a delayed action 1263 > +≡
`align_group: begin back_input; cur_tok ← cs_token_flag + frozen_cr; print_err("Missing");`
`print_esc("cr"); print(" inserted");`
`help1("I'm guessing that you meant to end an alignment here."); ins_error;`
`end;`

1311. < Cases of handle_right_brace where a `right_brace` triggers a delayed action 1263 > +≡
`no_align_group: begin end_graf; unsave; align_peek;`
`end;`

1312. Finally, `\endcsname` is not supposed to get through to `main_control`.

```
< Cases of main_control that build boxes and lists 1234 > +≡
any_mode(end_cs_name): cs_error;
```

1313. < Declare action procedures for use by main_control 1221 > +≡
procedure cs_error;

```
begin print_err("Extra"); print_esc("endcsname");
help1("I'm ignoring this, since I wasn't doing a \csname."); error;
end;
```

1314. Building math lists. The routines that TeX uses to create mlists are similar to those we have just seen for the generation of hlists and vlists. But it is necessary to make “noads” as well as nodes, so the reader should review the discussion of math mode data structures before trying to make sense out of the following program.

Here is a little routine that needs to be done whenever a subformula is about to be processed. The parameter is a code like *math_group*.

```
< Declare action procedures for use by main_control 1221 > +≡
procedure push_math(c : group_code);
begin push_nest; mode ← -mmode; incompleat_noad ← null; new_save_level(c);
end;
```

1315. We get into math mode from horizontal mode when a ‘\$’ (i.e., a *math_shift* character) is scanned. We must check to see whether this ‘\$’ is immediately followed by another, in case display math mode is called for.

```
< Cases of main_control that build boxes and lists 1234 > +≡
hmode + math_shift: init_math;
```

1316. < Declare action procedures for use by main_control 1221 > +≡
< Declare subprocedures for init_math 1733 >

```
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) ∧ (mode > 0) then < Go into display math mode 1323 >
else begin back_input; < Go into ordinary math mode 1317 >;
end;
end;
```

1317. < Go into ordinary math mode 1317 > ≡

```
begin push_math(math_shift_group); eq_word_define(int_base + cur_fam_code, -1);
if every_math ≠ null then begin_token_list(every_math, every_math_text);
end
```

This code is used in sections 1316 and 1320.

1318. We get into ordinary math mode from display math mode when ‘\eqno’ or ‘\leqno’ appears. In such cases *cur_chr* will be 0 or 1, respectively; the value of *cur_chr* is placed onto *save_stack* for safe keeping.

< Cases of main_control that build boxes and lists 1234 > +≡

```
mmode + eq_no: if privileged then
  if cur_group = math_shift_group then start_eq_no
  else off_save;
```

1319. ⟨ Put each of T_EX's primitives into the hash table 244 ⟩ +≡
primitive("eqno", eq_no, 0); primitive("leqno", eq_no, 1);

1320. When T_EX is in display math mode, *cur_group* = *math_shift_group*, so it is not necessary for the *start_eq_no* procedure to test for this condition.

⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡
procedure *start_eq_no*;
 begin *saved*(0) ← *cur_chr*; *incr*(*save_ptr*); ⟨ Go into ordinary math mode 1317 ⟩;
 end;

1321. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 245 ⟩ +≡
eq_no: if chr_code = 1 then print_esc("leqno") else print_esc("eqno");

1322. ⟨ Forbidden cases detected in *main_control* 1226 ⟩ +≡
non_math(eq_no),

1323. 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 1323 ⟩ ≡
 begin *j* ← null; *w* ← -*max_dimen*;
 if *head* = *tail* **then** { '\noindent\$\$' or '\$\$ \$\$' }
 ⟨ Prepare for display after an empty paragraph 1732 ⟩
 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* ← *max_dimen* if the non-blank information on that line is
 affected by stretching or shrinking 1324 ⟩;
 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 1327 ⟩;
 push_math(*math_shift_group*); *mode* ← *mmode*; *eq_word_define*(*int_base* + *cur_fam_code*, -1);
 eq_word_define(*dimen_base* + *pre_display_size_code*, *w*); *LR_box* ← *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* ≠ null **then** *begin_token_list*(*every_display*, *every_display_text*);
 if *nest_ptr* = 1 **then** *build_page*;
 end

This code is used in section 1316.

1324. ⟨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 1324⟩ ≡
 ⟨ Prepare for display after a non-empty paragraph 1734 ⟩;
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 $v \leftarrow max_dimen$ 1325⟩;
 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 1735⟩

This code is used in section 1323.

1325. ⟨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 $v \leftarrow max_dimen$ 1325⟩ ≡
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* 826⟩;
margin_kern_node: $d \leftarrow width(p)$;
kern_node: $d \leftarrow width(p)$;
⟨ Cases of ‘Let d be the natural width’ that need special treatment 1736 ⟩
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 1326⟩;
whatsit_node: ⟨Let d be the width of the whatsit p 1608⟩;
othercases $d \leftarrow 0$
endcases

This code is used in section 1324.

1326. We need to be careful that w , v , and d do not depend on any *glue_set* values, since such values are subject to system-dependent rounding. System-dependent numbers are not allowed to infiltrate parameters like *pre_display_size*, since TeX82 is supposed to make the same decisions on all machines.

\langle Let d be the natural width of this glue; if stretching or shrinking, set $v \leftarrow \text{max_dimen}$; **goto found** in the case of leaders 1326 $\rangle \equiv$

```
begin  $q \leftarrow \text{glue\_ptr}(p)$ ;  $d \leftarrow \text{width}(q)$ ;
if  $\text{glue\_sign}(\text{just\_box}) = \text{stretching}$  then
  begin if  $(\text{glue\_order}(\text{just\_box}) = \text{stretch\_order}(q)) \wedge (\text{stretch}(q) \neq 0)$  then  $v \leftarrow \text{max\_dimen}$ ;
  end
else if  $\text{glue\_sign}(\text{just\_box}) = \text{shrinking}$  then
  begin if  $(\text{glue\_order}(\text{just\_box}) = \text{shrink\_order}(q)) \wedge (\text{shrink}(q) \neq 0)$  then  $v \leftarrow \text{max\_dimen}$ ;
  end;
if  $\text{subtype}(p) \geq a\_leaders$  then goto found;
end
```

This code is used in section 1325.

1327. A displayed equation is considered to be three lines long, so we calculate the length and offset of line number *prev_graf* + 2.

\langle Calculate the length, l , and the shift amount, s , of the display lines 1327 $\rangle \equiv$

```
if  $\text{par\_shape\_ptr} = \text{null}$  then
  if  $(\text{hang\_indent} \neq 0) \wedge (((\text{hang\_after} \geq 0) \wedge (\text{prev\_graf} + 2 > \text{hang\_after})) \vee$ 
      $(\text{prev\_graf} + 1 < -\text{hang\_after}))$  then
    begin  $l \leftarrow \text{hsize} - \text{abs}(\text{hang\_indent})$ ;
    if  $\text{hang\_indent} > 0$  then  $s \leftarrow \text{hang\_indent}$  else  $s \leftarrow 0$ ;
    end
  else begin  $l \leftarrow \text{hsize}$ ;  $s \leftarrow 0$ ;
  end
else begin  $n \leftarrow \text{info}(\text{par\_shape\_ptr})$ ;
  if  $\text{prev\_graf} + 2 \geq n$  then  $p \leftarrow \text{par\_shape\_ptr} + 2 * n$ 
  else  $p \leftarrow \text{par\_shape\_ptr} + 2 * (\text{prev\_graf} + 2)$ ;
   $s \leftarrow \text{mem}[p - 1].sc$ ;  $l \leftarrow \text{mem}[p].sc$ ;
end
```

This code is used in section 1323.

1328. Subformulas of math formulas cause a new level of math mode to be entered, on the semantic nest as well as the save stack. These subformulas arise in several ways: (1) A left brace by itself indicates the beginning of a subformula that will be put into a box, thereby freezing its glue and preventing line breaks. (2) A subscript or superscript is treated as a subformula if it is not a single character; the same applies to the nucleus of things like *\underline*. (3) The *\left* primitive initiates a subformula that will be terminated by a matching *\right*. The group codes placed on *save_stack* in these three cases are *math_group*, *math_group*, and *math_left_group*, respectively.

Here is the code that handles case (1); the other cases are not quite as trivial, so we shall consider them later.

\langle Cases of *main_control* that build boxes and lists 1234 $\rangle + \equiv$

```
memode + left_brace: begin tail_append(new_noad); back_input; scan_math(nucleus(tail));
end;
```

1329. Recall that the *nucleus*, *subscr*, and *supscr* fields in a node are broken down into subfields called *math_type* and either *info* or (*fam*, *character*). The job of *scan_math* is to figure out what to place in one of these principal fields; it looks at the subformula that comes next in the input, and places an encoding of that subformula into a given word of *mem*.

```
define fam_in_range ≡ ((cur_fam ≥ 0) ∧ (cur_fam < 16))
⟨ Declare action procedures for use by main_control 1221 ⟩ +≡
procedure scan_math(p : pointer);
label restart, reswitch, exit;
var c: integer; { math character code }
begin restart: ⟨ Get the next non-blank non-relax non-call token 430 ⟩;
reswitch: case cur_cmd of
letter, other_char, char_given: begin c ← ho(math_code(cur_chr));
if c = '100000 then
begin ⟨ Treat cur_chr as an active character 1330 ⟩;
goto restart;
end;
end;
char_num: begin scan_char_num; cur_chr ← cur_val; cur_cmd ← char_given; goto reswitch;
end;
math_char_num: begin scan_fifteen_bit_int; c ← cur_val;
end;
math_given: c ← cur_chr;
delim_num: begin scan_twenty_seven_bit_int; c ← cur_val div '10000;
end;
othercases ⟨ Scan a subformula enclosed in braces and return 1331 ⟩
endcases;
math_type(p) ← math_char; character(p) ← qi(c mod 256);
if (c ≥ var_code) ∧ fam_in_range then fam(p) ← cur_fam
else fam(p) ← (c div 256) mod 16;
exit: end;
```

1330. An active character that is an *outer_call* is allowed here.

```
⟨ Treat cur_chr as an active character 1330 ⟩ ≡
begin cur_cs ← cur_chr + active_base; cur_cmd ← eq_type(cur_cs); cur_chr ← equiv(cur_cs); x_token;
back_input;
end
```

This code is used in sections 1329 and 1333.

1331. The pointer *p* is placed on *save_stack* while a complex subformula is being scanned.

```
⟨ Scan a subformula enclosed in braces and return 1331 ⟩ ≡
begin back_input; scan_left_brace;
saved(0) ← p; incr(save_ptr); push_math(math_group); return;
end
```

This code is used in section 1329.

1332. The simplest math formula is, of course, '\$ \$', when no noads are generated. The next simplest cases involve a single character, e.g., '\$x\$'. Even though such cases may not seem to be very interesting, the reader can perhaps understand how happy the author was when '\$x\$' was first properly typeset by TeX. The code in this section was used.

```
<Cases of main_control that build boxes and lists 1234> +≡
mmode + letter, mode + other_char, mode + char_given: set_math_char(ho(math_code(cur_chr)));
mode + char_num: begin scan_char_num; cur_chr ← cur_val; set_math_char(ho(math_code(cur_chr)));
end;
mode + math_char_num: begin scan_fifteen_bit_int; set_math_char(cur_val);
end;
mode + math_given: set_math_char(cur_chr);
mode + delim_num: begin scan_twenty_seven_bit_int; set_math_char(cur_val div `10000);
end;
```

1333. The *set_math_char* procedure creates a new noad appropriate to a given math code, and appends it to the current mlist. However, if the math code is sufficiently large, the *cur_chr* is treated as an active character and nothing is appended.

```
<Declare action procedures for use by main_control 1221> +≡
procedure set_math_char(c: integer);
  var p: pointer; { the new noad }
begin if c ≥ `100000 then <Treat cur_chr as an active character 1330>
else begin p ← new_noad; math_type(nucleus(p)) ← math_char;
character(nucleus(p)) ← qi(c mod 256); fam(nucleus(p)) ← (c div 256) mod 16;
if c ≥ var_code then
  begin if fam_in_range then fam(nucleus(p)) ← cur_fam;
  type(p) ← ord_noad;
  end
else type(p) ← ord_noad + (c div `10000);
link(tail) ← p; tail ← p;
end;
end;
```

1334. Primitive math operators like \mathop and \underline are given the command code *math_comp*, supplemented by the noad type that they generate.

```
<Put each of TeX's primitives into the hash table 244> +≡
primitive("mathord", math_comp, ord_noad); primitive("mathop", math_comp, op_noad);
primitive("mathbin", math_comp, bin_noad); primitive("mathrel", math_comp, rel_noad);
primitive("mathopen", math_comp, open_noad); primitive("mathclose", math_comp, close_noad);
primitive("mathpunct", math_comp, punct_noad); primitive("mathinner", math_comp, inner_noad);
primitive("underline", math_comp, under_noad); primitive("overline", math_comp, over_noad);
primitive("displaylimits", limit_switch, normal); primitive("limits", limit_switch, limits);
primitive("nolimits", limit_switch, no_limits);
```

1335. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡

```
math_comp: case chr_code 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");
  under_noad: print_esc("underline");
  othercases print_esc("overline")
endcases;
limit_switch: if chr_code = limits then print_esc("limits")
  else if chr_code = no_limits then print_esc("nolimits")
  else print_esc("displaylimits");
```

1336. ⟨Cases of *main_control* that build boxes and lists 1234⟩ +≡

```
mmode + math_comp: begin tail_append(new_noad); type(tail) ← cur_chr; scan_math(nucleus(tail));
end;
mmode + limit_switch: math_limit_switch;
```

1337. ⟨Declare action procedures for use by *main_control* 1221⟩ +≡

```
procedure math_limit_switch;
label exit;
begin if head ≠ tail then
  if type(tail) = op_noad then
    begin subtype(tail) ← cur_chr; return;
    end;
  print_err("Limit_controls_must_follow_a_math_operator");
  help1("I'm_ignoring_this_misplaced_limits_or_nolimits_command.");
exit: end;
```

1338. Delimiter fields of noads are filled in by the *scan_delimiter* routine. The first parameter of this procedure is the *mem* address where the delimiter is to be placed; the second tells if this delimiter follows \radical or not.

⟨Declare action procedures for use by *main_control* 1221⟩ +≡

```
procedure scan_delimiter(p: pointer; r: boolean);
begin if r then scan_twenty_seven_bit_int
else begin ⟨Get the next non-blank non-relax non-call token 430⟩;
  case cur_cmd of
    letter, other_char: cur_val ← del_code(cur_chr);
    delim_num: scan_twenty_seven_bit_int;
    othercases cur_val ← -1
  endcases;
  end;
  if cur_val < 0 then
    ⟨Report that an invalid delimiter code is being changed to null; set cur_val ← 0 1339⟩;
    small_fam(p) ← (cur_val div '4000000) mod 16; small_char(p) ← qi((cur_val div '10000) mod 256);
    large_fam(p) ← (cur_val div 256) mod 16; large_char(p) ← qi(cur_val mod 256);
  end;
```

1339. ⟨ Report that an invalid delimiter code is being changed to null; set $cur_val \leftarrow 0$ 1339 ⟩ ≡

```
begin print_err("Missing delimiter (. inserted)");
help6("I was expecting to see something like (` or `\\{` or `\\}` here.\nIf you typed, e.g., `(` instead of `\\{` , you")
("should probably delete the `(` by typing `1` now, so that")
("braces don't get unbalanced.\nOtherwise just proceed.")
("Acceptable delimiters are characters whose \\delcode is")
("nonnegative, or you can use `\\delimiter<delimiter_code>`.");
back_error; cur_val ← 0;
end;
```

This code is used in section 1338.

1340. ⟨ Cases of $main_control$ that build boxes and lists 1234 ⟩ +≡
 $mmode + radical: math_radical;$

1341. ⟨ Declare action procedures for use by $main_control$ 1221 ⟩ +≡
procedure $math_radical$;

```
begin tail_append(get_node(radical_noad_size)); type(tail) ← radical_noad; subtype(tail) ← normal;
mem[nucleus(tail)].hh ← empty_field; mem[subscr(tail)].hh ← empty_field;
mem[supscr(tail)].hh ← empty_field; scan_delimiter(left_delimiter(tail), true); scan_math(nucleus(tail));
end;
```

1342. ⟨ Cases of $main_control$ that build boxes and lists 1234 ⟩ +≡
 $mmode + accent, mmode + math_accent: math_ac;$

1343. ⟨ Declare action procedures for use by $main_control$ 1221 ⟩ +≡
procedure $math_ac$;

```
begin if cur_cmd = accent then ⟨ Complain that the user should have said \\mathaccent 1344 ⟩;
tail_append(get_node(accent_noad_size)); type(tail) ← accent_noad; subtype(tail) ← normal;
mem[nucleus(tail)].hh ← empty_field; mem[subscr(tail)].hh ← empty_field;
mem[supscr(tail)].hh ← empty_field; math_type(accent_chr(tail)) ← math_char; scan_fifteen_bit_int;
character(accent_chr(tail)) ← qi(cur_val mod 256);
if (cur_val ≥ var_code) ∧ fam_in_range then fam(accent_chr(tail)) ← cur_fam
else fam(accent_chr(tail)) ← (cur_val div 256) mod 16;
scan_math(nucleus(tail));
end;
```

1344. ⟨ Complain that the user should have said \\mathaccent 1344 ⟩ ≡

```
begin print_err("Please use "); print_esc("mathaccent"); print(" for accents in math mode");
help2("I'm changing \\accent to \\mathaccent here; wish me luck.")
("Accents are not the same in formulas as they are in text."); error;
end
```

This code is used in section 1343.

1345. ⟨ Cases of $main_control$ that build boxes and lists 1234 ⟩ +≡
 $mmode + vcenter: begin scan_spec(vcenter_group, false); normal_paragraph; push_nest; mode ← -vmode;$
 $prev_depth ← pdf_ignored_dimen;$
 $if every_vbox ≠ null then begin_token_list(every_vbox, every_vbox_text);$
end;

1346. \langle Cases of `handle_right_brace` where a `right_brace` triggers a delayed action 1263 $\rangle + \equiv$
`vcenter_group: begin end_graf; unsave; save_ptr ← save_ptr - 2;`
`p ← vpack(link(head), saved(1), saved(0)); pop_nest; tail_append(new_noad); type(tail) ← vcenter_noad;`
`math_type(nucleus(tail)) ← sub_box; info(nucleus(tail)) ← p;`
`end;`

1347. The routine that inserts a `style_node` holds no surprises.

\langle Put each of TeX's primitives into the hash table 244 $\rangle + \equiv$
`primitive("displaystyle", math_style, display_style); primitive("textstyle", math_style, text_style);`
`primitive("scriptstyle", math_style, script_style);`
`primitive("scriptscriptstyle", math_style, script_script_style);`

1348. \langle Cases of `print_cmd_chr` for symbolic printing of primitives 245 $\rangle + \equiv$
`math_style: print_style(chr_code);`

1349. \langle Cases of `main_control` that build boxes and lists 1234 $\rangle + \equiv$
`mmode + math_style: tail_append(new_style(cur_chr));`
`mmode + non_script: begin tail_append(new_glue(zero_glue)); subtype(tail) ← cond_math_glue;`
`end;`
`mmode + math_choice: append_choices;`

1350. The routine that scans the four mlists of a `\mathchoice` is very much like the routine that builds discretionary nodes.

\langle Declare action procedures for use by `main_control` 1221 $\rangle + \equiv$
`procedure append_choices;`
`begin tail_append(new_choice); incr(save_ptr); saved(-1) ← 0; push_math(math_choice_group);`
`scan_left_brace;`
`end;`

1351. \langle Cases of `handle_right_brace` where a `right_brace` triggers a delayed action 1263 $\rangle + \equiv$
`math_choice_group: build_choices;`

1352. \langle Declare action procedures for use by `main_control` 1221 $\rangle + \equiv$
 \langle Declare the function called `fin_mlist` 1362 \rangle
`procedure build_choices;`
`label exit;`
`var p: pointer; { the current mlist }`
`begin unsave; p ← fin_mlist(null);`
`case saved(-1) of`
`0: display_mlist(tail) ← p;`
`1: text_mlist(tail) ← p;`
`2: script_mlist(tail) ← p;`
`3: begin script_script_mlist(tail) ← p; decr(save_ptr); return;`
`end;`
`end; { there are no other cases }`
`incr(saved(-1)); push_math(math_choice_group); scan_left_brace;`
`exit: end;`

1353. Subscripts and superscripts are attached to the previous nucleus by the action procedure called *sub_sup*. We use the facts that *sub_mark* = *sup_mark* + 1 and *subscr*(*p*) = *supscr*(*p*) + 1.

$\langle \text{Cases of } \text{main_control} \text{ that build boxes and lists 1234} \rangle + \equiv$
mmode + sub_mark, mmode + sup_mark: sub_sup;

1354. $\langle \text{Declare action procedures for use by } \text{main_control 1221} \rangle + \equiv$
procedure *sub_sup*;

```
var t: small_number; { type of previous sub/superscript }
p: pointer; { field to be filled by scan_math }
begin t  $\leftarrow$  empty; p  $\leftarrow$  null;
if tail  $\neq$  head then
  if scripts_allowed(tail) then
    begin p  $\leftarrow$  supscr(tail) + cur_cmd - sup_mark; { supscr or subscr }
    t  $\leftarrow$  math_type(p);
    end;
  if (p = null)  $\vee$  (t  $\neq$  empty) then <Insert a dummy noad to be sub/superscripted 1355>;
  scan_math(p);
end;
```

1355. <Insert a dummy noad to be sub/superscripted 1355> \equiv
begin tail_append(*new_noad*); *p* \leftarrow supscr(*tail*) + cur_cmd - sup_mark; { supscr or subscr }
if *t* \neq empty then
 begin if cur_cmd = sup_mark then
 begin print_err("Double-superscript");
 help1("I_treat_x^1^2_essentially_like_x^1{}^2.");
 end
 else begin print_err("Double-subscript");
 help1("I_treat_x_1_2_essentially_like_x_1{}_2.");
 end;
 error;
 end;
end

This code is used in section 1354.

1356. An operation like ‘\over’ causes the current mlist to go into a state of suspended animation: *incompleat_noad* points to a *fraction_noad* that contains the mlist-so-far as its numerator, while the denominator is yet to come. Finally when the mlist is finished, the denominator will go into the incompleat fraction noad, and that noad will become the whole formula, unless it is surrounded by ‘\left’ and ‘\right’ delimiters.

```
define above_code = 0 { '\above' }
define over_code = 1 { '\over' }
define atop_code = 2 { '\atop' }
define delimited_code = 3 { '\abovewithdelims', etc. }
```

$\langle \text{Put each of } \text{\TeX}'\text{s primitives into the hash table 244} \rangle + \equiv$
primitive("above", above, above_code);
primitive("over", above, over_code);
primitive("atop", above, atop_code);
primitive("abovewithdelims", above, delimited_code + above_code);
primitive("overwithdelims", above, delimited_code + over_code);
primitive("atopwithdelims", above, delimited_code + atop_code);

1357. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡
above: **case** *chr_code* **of**

```
over_code: print_esc("over");
atop_code: print_esc("atop");
delimited_code + above_code: print_esc("abovewithdelims");
delimited_code + over_code: print_esc("overwithdelims");
delimited_code + atop_code: print_esc("atopwithdelims");
othercases print_esc("above")
endcases;
```

1358. ⟨Cases of *main_control* that build boxes and lists 1234⟩ +≡
mmode + *above*: *math_fraction*;

1359. ⟨Declare action procedures for use by *main_control* 1221⟩ +≡
procedure *math_fraction*;

```
var c: small_number; { the type of generalized fraction we are scanning }
begin c ← cur_chr;
if incompleat_noad ≠ null then
    ⟨Ignore the fraction operation and complain about this ambiguous case 1361⟩
else begin incompleat_noad ← get_node(fraction_noad_size); type(incompleat_noad) ← fraction_noad;
    subtype(incompleat_noad) ← normal; math_type(numerator(incompleat_noad)) ← sub_mlist;
    info(numerator(incompleat_noad)) ← link(head);
    mem[denominator(incompleat_noad)].hh ← empty_field;
    mem[left_delimiter(incompleat_noad)].qqqq ← null_delimiter;
    mem[right_delimiter(incompleat_noad)].qqqq ← null_delimiter;
    link(head) ← null; tail ← head; ⟨Use code c to distinguish between generalized fractions 1360⟩;
end;
end;
```

1360. ⟨Use code *c* to distinguish between generalized fractions 1360⟩ ≡

```
if c ≥ delimited_code then
    begin scan_delimiter(left_delimiter(incompleat_noad), false);
    scan_delimiter(right_delimiter(incompleat_noad), false);
    end;
case c mod delimited_code of
    above_code: begin scan_normal_dimen; thickness(incompleat_noad) ← cur_val;
    end;
    over_code: thickness(incompleat_noad) ← default_code;
    atop_code: thickness(incompleat_noad) ← 0;
end { there are no other cases }
```

This code is used in section 1359.

1361. *⟨ Ignore the fraction operation and complain about this ambiguous case 1361 ⟩* ≡

```

begin if c ≥ delimited_code then
  begin scan_delimiter(garbage, false); scan_delimiter(garbage, false);
  end;
  if c mod delimited_code = above_code then scan_normal_dimen;
  print_err("Ambiguous; you need another { and }");
  help3("I'm ignoring this fraction specification, since I don't")
  ("know whether a construction like `x\over y\over z`")
  ("means `x\over y\over z` or `x\over y\over z`."); error;
end

```

This code is used in section 1359.

1362. At the end of a math formula or subformula, the *fin_mlist* routine is called upon to return a pointer to the newly completed mlist, and to pop the nest back to the enclosing semantic level. The parameter to *fin_mlist*, if not null, points to a *right_noad* that ends the current mlist; this *right_noad* has not yet been appended.

⟨ Declare the function called fin_mlist 1362 ⟩ ≡

```

function fin_mlist(p : pointer): pointer;
  var q: pointer; { the mlist to return }
  begin if incompleat_noad ≠ null then ⟨ Compleat the incompleat noad 1363 ⟩
  else begin link(tail) ← p; q ← link(head);
  end;
  pop_nest; fin_mlist ← q;
end;

```

This code is used in section 1352.

1363. *⟨ Compleat the incompleat noad 1363 ⟩* ≡

```

begin math_type(denominator(incompleat_noad)) ← sub_mlist;
info(denominator(incompleat_noad)) ← link(head);
if p = null then q ← incompleat_noad
else begin q ← info(numerator(incompleat_noad));
  if (type(q) ≠ left_noad) ∨ (delim_ptr = null) then confusion("right");
  info(numerator(incompleat_noad)) ← link(delim_ptr); link(delim_ptr) ← incompleat_noad;
  link(incompleat_noad) ← p;
  end;
end

```

This code is used in section 1362.

1364. Now at last we're ready to see what happens when a right brace occurs in a math formula. Two special cases are simplified here: Braces are effectively removed when they surround a single Ord without sub/superscripts, or when they surround an accent that is the nucleus of an Ord atom.

```
<Cases of handle_right_brace where a right_brace triggers a delayed action 1263> +≡
math_group: begin unsave; decr(save_ptr);
  math_type(saved(0)) ← sub_mlist; p ← fin_mlist(null); info(saved(0)) ← p;
  if p ≠ null then
    if link(p) = null then
      if type(p) = ord_noad then
        begin if math_type(subscr(p)) = empty then
          if math_type(supscr(p)) = empty then
            begin mem[saved(0)].hh ← mem[nucleus(p)].hh; free_node(p, noad_size);
            end;
          end
        else if type(p) = accent_noad then
          if saved(0) = nucleus(tail) then
            if type(tail) = ord_noad then <Replace the tail of the list by p 1365>;
        end;
      end;
    end;
```

1365. <Replace the tail of the list by p 1365> ≡

```
begin q ← head;
while link(q) ≠ tail do q ← link(q);
link(q) ← p; free_node(tail, noad_size); tail ← p;
end
```

This code is used in section 1364.

1366. We have dealt with all constructions of math mode except '\left' and '\right', so the picture is completed by the following sections of the program.

```
<Put each of TeX's primitives into the hash table 244> +≡
primitive("left", left_right, left_noad); primitive("right", left_right, right_noad);
text(frozen_right) ← "right"; eqtb[frozen_right] ← eqtb[cur_val];
```

1367. <Cases of print_cmd_chr for symbolic printing of primitives 245> +≡
`left_right: if chr_code = left_noad then print_esc("left")
 <Cases of left_right for print_cmd_chr 1698>
 else print_esc("right");`

1368. <Cases of main_control that build boxes and lists 1234> +≡
`memode + left_right: math_left_right;`

1369. \langle Declare action procedures for use by *main_control* 1221 $\rangle +\equiv$

```

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 1370}
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;
```

1370. { Try to recover from mismatched \right 1370 } \equiv

```

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 1369.

1371. Here is the only way out of math mode.

\langle Cases of *main_control* that build boxes and lists 1234 $\rangle +\equiv$

```

mmode + math_shift: if cur_group = math_shift_group then after_math
  else off_save;
```

1372. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡
 ⟨ Declare subprocedures for *after_math* 1744 ⟩

```

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 1376 }

begin danger ← false; { Retrieve the prototype box 1742 };
{ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set
  danger ← true 1373 };
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 1375 };
  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 1742 };
{ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and
  set danger ← true 1373 };
m ← mode; p ← fin_mlist(null);
end
else a ← null;
if m < 0 then { Finish math in text 1374 }
else begin if a = null then { Check that another $ follows 1375 };
  { Finish displayed math 1377 };
end;
end;
```

1373. ⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← true 1373 ⟩ ≡

```

if (font_params[fam_fnt(2 + text_size)] < total_mathsy_params) ∨
  (font_params[fam_fnt(2 + script_size)] < total_mathsy_params) ∨
  (font_params[fam_fnt(2 + script_script_size)] < total_mathsy_params) then
  begin print_err("Math_formula_deleted:_Insufficient_symbol_fonts");
  help3("Sorry, but I can't typeset math unless \textfont 2")
  ("and \scriptfont 2 and \scriptscriptfont 2 have all")
  ("the \fontdimen values needed in math symbol fonts."); error; flush_math; danger ← true;
  end
else if (font_params[fam_fnt(3 + text_size)] < total_mathx_params) ∨
  (font_params[fam_fnt(3 + script_size)] < total_mathx_params) ∨
  (font_params[fam_fnt(3 + script_script_size)] < total_mathx_params) then
  begin print_err("Math_formula_deleted:_Insufficient_extension_fonts");
  help3("Sorry, but I can't typeset math unless \textfont 3")
  ("and \scriptfont 3 and \scriptscriptfont 3 have all")
  ("the \fontdimen values needed in math extension fonts."); error; flush_math;
  danger ← true;
  end
```

This code is used in sections 1372 and 1372.

1374. The *unsafe* is done after everything else here; hence an appearance of ‘`\mathsurround`’ inside of ‘`$...$`’ affects the spacing at these particular `$`’s. This is consistent with the conventions of ‘`$$...$$`’, since ‘`\abovedisplayskip`’ inside a display affects the space above that display.

`<Finish math in text 1374>` ≡

```
begin tail_append(new_math(math_surround, before)); cur_mlist ← p; cur_style ← text_style;
  mlist_penalties ← (mode > 0); mlist_to_hlist; link(tail) ← link(temp_head);
  while link(tail) ≠ null do tail ← link(tail);
  tail_append(new_math(math_surround, after)); space_factor ← 1000; unsafe;
end
```

This code is used in section 1372.

1375. TeX gets to the following part of the program when the first ‘`$`’ ending a display has been scanned.

`<Check that another $ follows 1375>` ≡

```
begin get_x_token;
if cur_cmd ≠ math_shift then
  begin print_err("Display_math_should_end_with_$$");
    help2("The `'$' that I just saw supposedly matches a previous `$$'.");
    ("So I shall assume that you typed `$$' both times."); back_error;
  end;
end
```

This code is used in sections 1372, 1372, and 1384.

1376. We have saved the worst for last: The fussiest part of math mode processing occurs when a displayed formula is being centered and placed with an optional equation number.

`<Local variables for finishing a displayed formula 1376>` ≡

```
b: pointer; { box containing the equation }
w: scaled; { width of the equation }
z: scaled; { width of the line }
e: scaled; { width of equation number }
q: scaled; { width of equation number plus space to separate from equation }
d: scaled; { displacement of equation in the line }
s: scaled; { move the line right this much }
g1, g2: small_number; { glue parameter codes for before and after }
r: pointer; { kern node used to position the display }
t: pointer; { tail of adjustment list }
pre_t: pointer; { tail of pre-adjustment list }
```

See also section 1741.

This code is used in section 1372.

1377. 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 1377> ==
  cur_mlist <- p; cur_style <- display_style; mlist_penalties <- false; mlist_to_hlist; p <- link(temp_head);
  adjust_tail <- adjust_head; pre_adjust.tail <- pre_adjust_head; b <- hpack(p, natural); p <- list_ptr(b);
  t <- adjust_tail; adjust_tail <- null;
  pre_t <- pre_adjust.tail; pre_adjust.tail <- null;
  w <- width(b); z <- display_width; s <- display_indent;
  if pre_display_direction < 0 then s <- -s - z;
  if (a = null) ∨ danger then
    begin e <- 0; q <- 0;
    end
  else begin e <- width(a); q <- 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 <- 0 1379>;
  <Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming
    that l = false 1380>;
  <Append the glue or equation number preceding the display 1381>;
  <Append the display and perhaps also the equation number 1382>;
  <Append the glue or equation number following the display 1383>;
  <Flush the prototype box 1743>;
  resume_after_display
```

This code is used in section 1372.

1378. < Declare action procedures for use by *main_control* 1221 > +≡

```
procedure resume_after_display;
begin if cur_group ≠ math_shift_group then confusion("display");
  unsave; prev_graf <- prev_graf + 3; push_nest; mode <- hmode; space_factor <- 1000; set_cur_lang;
  clang <- cur_lang;
  prev_graf <- (norm_min(left_hyphen_min) * '100 + norm_min(right_hyphen_min)) * '200000 + cur_lang;
  <Scan an optional space 469>;
  if nest_ptr = 1 then build_page;
end;
```

1379. The user can force the equation number to go on a separate line by causing its width to be zero.

```
<Squeeze the equation as much as possible; if there is an equation number that should go on a separate line
  by itself, set e <- 0 1379> ≡
begin if (e ≠ 0) ∧ ((w - total_shrink[normal] + q ≤ z) ∨
  (total_shrink[fil] ≠ 0) ∨ (total_shrink[fill] ≠ 0) ∨ (total_shrink[filll] ≠ 0)) then
  begin free_node(b, box_node_size); b <- hpack(p, z - q, exactly);
  end
else begin e <- 0;
  if w > z then
    begin free_node(b, box_node_size); b <- hpack(p, z, exactly);
    end;
  end;
w <- width(b);
end
```

This code is used in section 1377.

1380. 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 $l = \text{false}$) \equiv

```
set_box_lr(b)(dlist); d ← half(z - w);
if (e > 0) ∧ (d < 2 * e) then { too close }
begin d ← half(z - w - e);
if p ≠ null then
  if ¬is_char_node(p) then
    if type(p) = glue_node then d ← 0;
end
```

This code is used in section 1377.

1381. 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) \equiv

```
tail_append(new_penalty(pre_display_penalty));
if (d + s ≤ pre_display_size) ∨ 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 ∧ (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 1377.

1382. *(Append the display and perhaps also the equation number)* \equiv

```
if e ≠ 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 1377.

1383. \langle Append the glue or equation number following the display 1383 $\rangle \equiv$

```

if ( $a \neq null$ )  $\wedge$  ( $e = 0$ )  $\wedge$   $\neg l$  then
  begin tail_append(new_penalty(inf_penalty)); app_display(j, a, z - width(a)); g2  $\leftarrow 0$ ;
end;
if  $t \neq adjust\_head$  then { migrating material comes after equation number }
  begin link(tail)  $\leftarrow$  link(adjust_head); tail  $\leftarrow t$ ;
end;
if  $pre\_t \neq pre\_adjust\_head$  then
  begin link(tail)  $\leftarrow$  link(pre_adjust_head); tail  $\leftarrow pre\_t$ ;
end;
tail_append(new_penalty(post_display_penalty));
if  $g2 > 0$  then tail_append(new_param_glue(g2))

```

This code is used in section 1377.

1384. 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.

\langle Finish an alignment in a display 1384 $\rangle \equiv$

```

begin do_assignments;
if cur_cmd  $\neq$  math_shift then ⟨ Pontificate about improper alignment in display 1385 ⟩
else ⟨ Check that another $ follows 1375 ⟩;
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)  $\leftarrow p$ ;
if  $p \neq null$  then tail  $\leftarrow q$ ;
tail_append(new_penalty(post_display_penalty)); tail_append(new_param_glue(below_display_skip_code));
prev_depth  $\leftarrow aux\_save.sc$ ; resume_after_display;
end

```

This code is used in section 988.

1385. \langle Pontificate about improper alignment in display 1385 $\rangle \equiv$

```

begin print_err("Missing $$ inserted");
help2("Displays can use special alignments (like \\eqalignno)");
("only if nothing but the alignment itself is between $$'s."); back_error;
end

```

This code is used in section 1384.

1386. 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 244> +≡

```
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);
```

1387. <Cases of *print_cmd_chr* for symbolic printing of primitives 245> +≡

```
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 1771>
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");
```

1388. Every prefix, and every command code that might or might not be prefixed, calls the action procedure *prefixed_command*. This routine accumulates a sequence of prefixes until coming to a non-prefix, then it carries out the command.

*<Cases of *main_control* that don’t depend on mode 1388>* ≡

```
any_mode(toks_register), any_mode(assign_toks), any_mode(assign_int), any_mode(assign_dimen),
any_mode(assign_glue), any_mode(assign_mu_glue), any_mode(assign_font_dimen),
any_mode(assign_font_int), any_mode(set_aux), any_mode(set_prev_graf), any_mode(set_page_dimen),
any_mode(set_page_int), any_mode(set_box_dimen), any_mode(set_shape), any_mode(def_code),
any_mode(def_family), any_mode(set_font), any_mode(def_font), any_mode(letterspace_font),
any_mode(pdf_copy_font), any_mode(register), any_mode(advance), any_mode(multiply),
any_mode(divide), any_mode(prefix), any_mode(let), any_mode(shorthand_def), any_mode(read_to_cs),
any_mode(def), any_mode(set_box), any_mode(hyph_data), any_mode(set_interaction):
prefixed_command;
```

See also sections 1446, 1449, 1452, 1454, 1463, and 1468.

This code is used in section 1223.

1389. If the user says, e.g., ‘\global\global’, the redundancy is silently accepted.

```

⟨ Declare action procedures for use by main_control 1221 ⟩ +≡
⟨ Declare subprocedures for prefixed_command 1393 ⟩
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 ← 0;
while cur_cmd = prefix do
  begin if  $\neg$ odd(a div cur_chr) then a ← a + cur_chr;
  ⟨ Get the next non-blank non-relax non-call token 430 ⟩;
  if cur_cmd ≤ max_non_prefixed_command then ⟨ Discard erroneous prefixes and return 1390 ⟩;
  if tracing_commands > 2 then
    if eTeX_ex then show_cur_cmd_chr;
    end;
  ⟨ Discard the prefixes \long and \outer if they are irrelevant 1391 ⟩;
  ⟨ Adjust for the setting of \globaldefs 1392 ⟩;
  case cur_cmd of
    ⟨ Assignments 1395 ⟩
    othercases confusion("prefix")
    endcases;
done: ⟨ Insert a token saved by \afterassignment, if any 1447 ⟩;
exit: end;
```

1390. ⟨ Discard erroneous prefixes and return 1390 ⟩ ≡

```

begin print_err("You\u2019can\u2019t\u2019use\u2019a\u2019prefix\u2019with\u2019"); print_cmd_chr(cur_cmd, cur_chr);
print_char(" "); help1("I\u2019ll\u2019pretend\u2019you\u2019didn\u2019t\u2019say\u2019\long\u2019or\u2019\outer\u2019or\u2019\global\u2019.");
if eTeX_ex then
  help_line[0] ← "I\u2019ll\u2019pretend\u2019you\u2019didn\u2019t\u2019say\u2019\long\u2019or\u2019\outer\u2019or\u2019\global\u2019or\u2019\protected\u2019.";
  back_error; return;
end
```

This code is used in section 1389.

1391. \langle Discard the prefixes `\long` and `\outer` if they are irrelevant 1391 $\rangle \equiv$

```

if  $a \geq 8$  then
  begin  $j \leftarrow \text{protected\_token}$ ;  $a \leftarrow a - 8$ ;
  end
else  $j \leftarrow 0$ ;
if ( $\text{cur\_cmd} \neq \text{def}$ )  $\wedge ((a \bmod 4 \neq 0) \vee (j \neq 0))$  then
  begin  $\text{print\_err}(\text{"You can't use "})$ ;  $\text{print\_esc}(\text{"long"})$ ;  $\text{print}(\text{" or "})$ ;  $\text{print\_esc}(\text{"outer"})$ ;
     $\text{help1}(\text{"I'll pretend you didn't say \long or \outer here."})$ ;
    if  $eTeX_{\text{ex}}$  then
      begin  $\text{help\_line}[0] \leftarrow \text{"I'll pretend you didn't say \long or \outer or \protected here. "}$ ;
         $\text{print}(\text{" or "})$ ;  $\text{print\_esc}(\text{"protected"})$ ;
        end;
       $\text{print}(\text{" with "})$ ;  $\text{print\_cmd\_chr}(\text{cur\_cmd}, \text{cur\_chr})$ ;  $\text{print\_char}(\text{" "})$ ;  $\text{error}$ ;
    end
  end

```

This code is used in section 1389.

1392. The previous routine does not have to adjust a so that $a \bmod 4 = 0$, since the following routines test for the `\global` prefix as follows.

```

define global  $\equiv (a \geq 4)$ 
define define(#)  $\equiv$ 
  if global then  $\text{geq\_define}(#)$  else  $\text{eq\_define}(#)$ 
define word_define(#)  $\equiv$ 
  if global then  $\text{geq\_word\_define}(#)$  else  $\text{eq\_word\_define}(#)$ 

 $\langle$  Adjust for the setting of \globaldefs 1392  $\rangle \equiv$ 
if  $\text{global\_defs} \neq 0$  then
  if  $\text{global\_defs} < 0$  then
    begin if global then  $a \leftarrow a - 4$ ;
    end
  else begin if  $\neg \text{global}$  then  $a \leftarrow a + 4$ ;
  end

```

This code is used in section 1389.

1393. When a control sequence is to be defined, by `\def` or `\let` or something similar, the `get_r_token` routine will substitute a special control sequence for a token that is not redefinable.

```
< Declare subprocedures for prefixed_command 1393 > ≡
procedure get_r_token;
label restart;
begin restart: repeat get_token;
until cur_tok ≠ space_token;
if (cur_cs = 0) ∨ (cur_cs > frozen_control_sequence) then
begin print_err("Missing control sequence inserted");
help5("Please don't say ``\\def cs{...}'', say ``\\def\\cs{...}''.")
("I've inserted an inaccessible control sequence so that your"
("definition will be completed without mixing me up too badly.")
("You can recover graciously from this error, if you're")
("careful; see exercise 27.2 in The TeXbook.");
if cur_cs = 0 then back_input;
cur_tok ← cs_token_flag + frozen_protection; ins_error; goto restart;
end;
end;
```

See also sections 1407, 1414, 1421, 1422, 1423, 1424, 1425, 1435, and 1443.

This code is used in section 1389.

1394. < Initialize table entries (done by INITEX only) 182 > +≡
`text(frozen_protection) ← "inaccessible";`

1395. Here's an example of the way many of the following routines operate. (Unfortunately, they aren't all as simple as this.)

```
< Assignments 1395 > ≡
set_font: define(cur_font_loc, data, cur_chr);
```

See also sections 1396, 1399, 1402, 1403, 1404, 1406, 1410, 1412, 1413, 1419, 1420, 1426, 1430, 1431, 1434, and 1442.

This code is used in section 1389.

1396. When a `def` command has been scanned, `cur_chr` is odd if the definition is supposed to be global, and $cur_chr \geq 2$ if the definition is supposed to be expanded.

```
< Assignments 1395 > +≡
def: begin if odd(cur_chr) ∧ ¬global ∧ (global_defs ≥ 0) then a ← a + 4;
e ← (cur_chr ≥ 2); get_r_token; p ← cur_cs; q ← scan_toks(true, e);
if j ≠ 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;
```

1397. Both `\let` and `\futurelet` share the command code `let`.

```
< Put each of TeX's primitives into the hash table 244 > +≡
primitive("let", let, normal);
primitive("futurelet", let, normal + 1);
```

1398. < Cases of `print_cmd_chr` for symbolic printing of primitives 245 > +≡
`let: if chr_code ≠ normal then print_esc("futurelet") else print_esc("let");`

1399. $\langle \text{Assignments } 1395 \rangle + \equiv$

```

let: begin  $n \leftarrow \text{cur\_chr}$ ;  $\text{get\_r\_token}$ ;  $p \leftarrow \text{cur\_cs}$ ;
  if  $n = \text{normal}$  then
    begin repeat  $\text{get\_token}$ ;
    until  $\text{cur\_cmd} \neq \text{spacer}$ ;
    if  $\text{cur\_tok} = \text{other\_token} + "="$  then
      begin  $\text{get\_token}$ ;
      if  $\text{cur\_cmd} = \text{spacer}$  then  $\text{get\_token}$ ;
      end;
    end
  else begin  $\text{get\_token}$ ;  $q \leftarrow \text{cur\_tok}$ ;  $\text{get\_token}$ ;  $\text{back\_input}$ ;  $\text{cur\_tok} \leftarrow q$ ;  $\text{back\_input}$ ;
    { look ahead, then back up }
  end; { note that  $\text{back\_input}$  doesn't affect  $\text{cur\_cmd}$ ,  $\text{cur\_chr}$  }
  if  $\text{cur\_cmd} \geq \text{call}$  then  $\text{add\_token\_ref}(\text{cur\_chr})$ 
  else if ( $\text{cur\_cmd} = \text{register}$ )  $\vee$  ( $\text{cur\_cmd} = \text{toks\_register}$ ) then
    if ( $\text{cur\_chr} < \text{mem\_bot}$ )  $\vee$  ( $\text{cur\_chr} > \text{lo\_mem\_stat\_max}$ ) then  $\text{add\_sa\_ref}(\text{cur\_chr})$ ;
  define ( $p$ ,  $\text{cur\_cmd}$ ,  $\text{cur\_chr}$ );
  end;

```

1400. A $\backslash\text{chardef}$ creates a control sequence whose cmd is char_given ; a $\backslash\text{mathchardef}$ creates a control sequence whose cmd is math_given ; and the corresponding chr is the character code or math code. A $\backslash\text{countdef}$ or $\backslash\text{dimendef}$ or $\backslash\text{skipdef}$ or $\backslash\text{muskipdef}$ creates a control sequence whose cmd is assign_int or ... or assign_mu_glue , and the corresponding chr is the eqtb location of the internal register in question.

```

define  $\text{char\_def\_code} = 0$  { shorthand_def for  $\backslash\text{chardef}$  }
define  $\text{math\_char\_def\_code} = 1$  { shorthand_def for  $\backslash\text{mathchardef}$  }
define  $\text{count\_def\_code} = 2$  { shorthand_def for  $\backslash\text{countdef}$  }
define  $\text{dimen\_def\_code} = 3$  { shorthand_def for  $\backslash\text{dimendef}$  }
define  $\text{skip\_def\_code} = 4$  { shorthand_def for  $\backslash\text{skipdef}$  }
define  $\text{mu\_skip\_def\_code} = 5$  { shorthand_def for  $\backslash\text{muskipdef}$  }
define  $\text{toks\_def\_code} = 6$  { shorthand_def for  $\backslash\text{toksdef}$  }

```

$\langle \text{Put each of TeX's primitives into the hash table } 244 \rangle + \equiv$

```

primitive("chardef", shorthand_def, char_def_code);
primitive("mathchardef", shorthand_def, math_char_def_code);
primitive("countdef", shorthand_def, count_def_code);
primitive("dimendef", shorthand_def, dimen_def_code);
primitive("skipdef", shorthand_def, skip_def_code);
primitive("muskipdef", shorthand_def, mu_skip_def_code);
primitive("toksdef", shorthand_def, toks_def_code);

```

1401. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡

```

shorthand_def: case chr_code of
  char_def_code: print_esc("chardef");
  math_char_def_code: print_esc("mathchardef");
  count_def_code: print_esc("countdef");
  dimen_def_code: print_esc("dimendef");
  skip_def_code: print_esc("skipdef");
  mu_skip_def_code: print_esc("muskipdef");
  othercases print_esc("toksdef")
endcases;

char_given: begin print_esc("char"); print_hex(chr_code);
end;
math_given: begin print_esc("mathchar"); print_hex(chr_code);
end;
```

1402. 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 1395⟩ +≡
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;
```

1403. $\langle \text{Assignments } 1395 \rangle + \equiv$

```

read_to_cs: begin  $j \leftarrow \text{cur\_chr}$ ;  $\text{scan\_int} \leftarrow \text{cur\_val}$ ;
  if  $\neg \text{scan\_keyword("to")}$  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;  $p \leftarrow \text{cur\_cs}$ ; read_toks( $n, p, j$ ); define( $p, \text{call}, \text{cur\_val}$ );
end;
```

1404. 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 1395 ⟩ + ≡
toks_register, assign_toks: begin  $q \leftarrow \text{cur\_cs}$ ;  $e \leftarrow \text{false}$ ;
  { just in case, will be set true for sparse array elements }
  if  $\text{cur\_cmd} = \text{toks\_register}$  then
    if  $\text{cur\_chr} = \text{mem\_bot}$  then
      begin scan_register_num;
      if  $\text{cur\_val} > 255$  then
        begin find_sa_element( $\text{tok\_val}, \text{cur\_val}, \text{true}$ );  $\text{cur\_chr} \leftarrow \text{cur\_ptr}$ ;  $e \leftarrow \text{true}$ ;
        end
      else  $\text{cur\_chr} \leftarrow \text{toks\_base} + \text{cur\_val}$ ;
      end
    else  $e \leftarrow \text{true}$ ;
     $p \leftarrow \text{cur\_chr}$ ; {  $p = \text{every\_par\_loc}$  or  $\text{output\_routine\_loc}$  or ... }
    scan_optional_equals; { Get the next non-blank non-relax non-call token 430 };
    if  $\text{cur\_cmd} \neq \text{left\_brace}$  then { If the right-hand side is a token parameter or token register, finish the
      assignment and goto done 1405 };
    back_input;  $\text{cur\_cs} \leftarrow q$ ;  $q \leftarrow \text{scan\_toks}(\text{false}, \text{false})$ ;
    if link( $\text{def\_ref}$ ) = null then { empty list: revert to the default }
      begin sa_define( $p, \text{null}$ )( $p, \text{undefined\_cs}, \text{null}$ ); free_avail( $\text{def\_ref}$ );
      end
    else begin if ( $p = \text{output\_routine\_loc}$ )  $\wedge \neg e$  then { enclose in curly braces }
      begin link( $q \leftarrow \text{get\_avail}$ ;  $q \leftarrow \text{link}(q)$ ; info( $q \leftarrow \text{right\_brace\_token} + "}"$ );  $q \leftarrow \text{get\_avail}$ ;
      info( $q \leftarrow \text{left\_brace\_token} + "{"$ ; link( $q \leftarrow \text{link}(\text{def\_ref})$ ; link( $\text{def\_ref} \leftarrow q$ );
      end;
      sa_define( $p, \text{def\_ref}$ )( $p, \text{call}, \text{def\_ref}$ );
    end;
  end;
```

1405. *<If the right-hand side is a token parameter or token register, finish the assignment and goto done 1405>* \equiv

```

if (cur_cmd = toks_register)  $\vee$  (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  $\leftarrow$  equiv(toks_base + cur_val)
    else begin find_sa_element(tok_val, cur_val, false);
      if cur_ptr = null then q  $\leftarrow$  null
      else q  $\leftarrow$  sa_ptr(cur_ptr);
      end;
    end
  else q  $\leftarrow$  sa_ptr(cur_chr)
  else q  $\leftarrow$  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 1404.

1406. Similar routines are used to assign values to the numeric parameters.

```

⟨ Assignments 1395 ⟩  $\equiv$ 
assign_int: begin p  $\leftarrow$  cur_chr; scan_optional_equals; scan_int; word_define(p, cur_val);
end;
assign_dimen: begin p  $\leftarrow$  cur_chr; scan_optional_equals; scan_normal_dimen; word_define(p, cur_val);
end;
assign_glue, assign_mu_glue: begin p  $\leftarrow$  cur_chr; n  $\leftarrow$  cur_cmd; scan_optional_equals;
  if n = assign_mu_glue then scan_glue(mu_val) else scan_glue(glue_val);
  trap_zero_glue; define(p, glue_ref, cur_val);
end;

```

1407. When a glue register or parameter becomes zero, it will always point to *zero_glue* because of the following procedure. (Exception: The tabskip glue isn't trapped while preambles are being scanned.)

```

⟨ Declare subprocedures for prefixed_command 1393 ⟩  $\equiv$ 
procedure trap_zero_glue;
begin if (width(cur_val) = 0)  $\wedge$  (stretch(cur_val) = 0)  $\wedge$  (shrink(cur_val) = 0) then
  begin add_glue_ref(zero_glue); delete_glue_ref(cur_val); cur_val  $\leftarrow$  zero_glue;
  end;
end;

```

1408. The various character code tables are changed by the *def_code* commands, and the font families are declared by *def_family*.

```

⟨ Put each of TeX's primitives into the hash table 244 ⟩  $\equiv$ 
primitive("catcode", def_code, cat_code_base); primitive("mathcode", def_code, math_code_base);
primitive("lccode", def_code, lc_code_base); primitive("uccode", def_code, uc_code_base);
primitive("sfcode", def_code, sf_code_base); primitive("delcode", def_code, del_code_base);
primitive("textfont", def_family, math_font_base);
primitive("scriptfont", def_family, math_font_base + script_size);
primitive("scriptscriptfont", def_family, math_font_base + script_script_size);

```

1409. \langle Cases of *print_cmd_chr* for symbolic printing of primitives 245 $\rangle +\equiv$

```
def_code: if chr_code = cat_code_base then print_esc("catcode")
  else if chr_code = math_code_base then print_esc("mathcode")
  else if chr_code = lc_code_base then print_esc("lccode")
  else if chr_code = uc_code_base then print_esc("uccode")
  else if chr_code = sf_code_base then print_esc("sfcode")
  else print_esc("delcode");
def_family: print_size(chr_code - math_font_base);
```

1410. The different types of code values have different legal ranges; the following program is careful to check each case properly.

\langle Assignments 1395 $\rangle +\equiv$

```
def_code: begin < Let n be the largest legal code value, based on cur_chr 1411 >
  p ← cur_chr; scan_char_num; p ← p + cur_val; scan_optional_equals; scan_int;
  if ((cur_val < 0) ∧ (p < del_code_base)) ∨ (cur_val > n) then
    begin print_err("Invalid_code("); print_int(cur_val);
    if p < del_code_base then print("), should_be_in_the_range_0..")
    else print("), should_be_at_most_");
    print_int(n); help1("I'm going to use 0 instead of that illegal code value.");
    error; cur_val ← 0;
    end;
  if p < math_code_base then define(p, data, cur_val)
  else if p < del_code_base then define(p, data, hi(cur_val))
  else word_define(p, cur_val);
end;
```

1411. \langle Let n be the largest legal code value, based on cur_chr 1411 $\rangle \equiv$

```
if cur_chr = cat_code_base then n ← max_char_code
else if cur_chr = math_code_base then n ← '100000
else if cur_chr = sf_code_base then n ← '77777
else if cur_chr = del_code_base then n ← '77777777
else n ← 255
```

This code is used in section 1410.

1412. \langle Assignments 1395 $\rangle +\equiv$

```
def_family: begin p ← cur_chr; scan_four_bit_int; p ← p + cur_val; scan_optional_equals; scan_font_ident;
  define(p, data, cur_val);
end;
```

1413. Next we consider changes to TeX's numeric registers.

\langle Assignments 1395 $\rangle +\equiv$

```
register, advance, multiply, divide: do_register_command(a);
```

1414. We use the fact that *register* < *advance* < *multiply* < *divide*.

```
< Declare subprocedures for prefixed_command 1393 > +≡
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 1415 };
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 1416 }
else { Compute result of multiply or divide, put it in cur_val 1418 };
if arith_error then
begin print_err("Arithmetic_overflow");
help2("Ican'tcarryothatmultiplicationordivision,")
("sincetheresultisotufrange.");
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;
```

1415. 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 1415 ⟩ ≡
begin if q ≠ register then
  begin get_x_token;
    if (cur_cmd ≥ assign_int) ∧ (cur_cmd ≤ assign_mu_glue) then
      begin l ← cur_chr; p ← cur_cmd − assign_int; goto found;
      end;
    if cur_cmd ≠ register then
      begin print_err("You\u201ccan't\u201duse\u201d"); print_cmd_chr(cur_cmd, cur_chr); print("`\u201d\u201a\u201d");
      print_cmd_chr(q, 0); help1("I'm\u201dforgetting\u201dwhat\u201dyou\u201dsaid\u201dand\u201dnot\u201dchanging\u201danything.\u201d");
      error; return;
      end;
    end;
  if (cur_chr < mem_bot) ∨ (cur_chr > lo_mem_stat_max) then
    begin l ← cur_chr; p ← sa_type(l); e ← true;
    end
  else begin p ← cur_chr − mem_bot; scan_register_num;
    if cur_val > 255 then
      begin find_sa_element(p, cur_val, true); l ← cur_ptr; e ← true;
      end
    else case p of
      int_val: l ← cur_val + count_base;
      dimen_val: l ← cur_val + scaled_base;
      glue_val: l ← cur_val + skip_base;
      mu_val: l ← cur_val + mu_skip_base;
      end; { there are no other cases }
    end;
  end;
found: if p < glue_val then if e then w ← sa_int(l) else w ← eqtb[l].int
  else if e then s ← sa_ptr(l) else s ← equiv(l)

```

This code is used in section 1414.

1416. ⟨ Compute result of *register* or *advance*, put it in *cur_val* 1416 ⟩ ≡

```

if p < glue_val then
  begin if p = int_val then scan_int else scan_normal_dimen;
    if q = advance then cur_val ← cur_val + w;
    end
  else begin scan_glue(p);
    if q = advance then ⟨ Compute the sum of two glue specs 1417 ⟩;
    end

```

This code is used in section 1414.

1417. \langle Compute the sum of two glue specs 1417 $\rangle \equiv$

```

begin  $q \leftarrow new\_spec(cur\_val)$ ;  $r \leftarrow s$ ;  $delete\_glue\_ref(cur\_val)$ ;  $width(q) \leftarrow width(q) + width(r)$ ;
if  $stretch(q) = 0$  then  $stretch\_order(q) \leftarrow normal$ ;
if  $stretch\_order(q) = stretch\_order(r)$  then  $stretch(q) \leftarrow stretch(q) + stretch(r)$ 
else if  $(stretch\_order(q) < stretch\_order(r)) \wedge (stretch(r) \neq 0)$  then
begin  $stretch(q) \leftarrow stretch(r)$ ;  $stretch\_order(q) \leftarrow stretch\_order(r)$ ;
end;
if  $shrink(q) = 0$  then  $shrink\_order(q) \leftarrow normal$ ;
if  $shrink\_order(q) = shrink\_order(r)$  then  $shrink(q) \leftarrow shrink(q) + shrink(r)$ 
else if  $(shrink\_order(q) < shrink\_order(r)) \wedge (shrink(r) \neq 0)$  then
begin  $shrink(q) \leftarrow shrink(r)$ ;  $shrink\_order(q) \leftarrow shrink\_order(r)$ ;
end;
 $cur\_val \leftarrow q$ ;
end

```

This code is used in section 1416.

1418. \langle Compute result of multiply or divide, put it in cur_val 1418 $\rangle \equiv$

```

begin  $scan\_int$ ;
if  $p < glue\_val$  then
if  $q = multiply$  then
if  $p = int\_val$  then  $cur\_val \leftarrow mult\_integers(w, cur\_val)$ 
else  $cur\_val \leftarrow nx\_plus\_y(w, cur\_val, 0)$ 
else  $cur\_val \leftarrow x\_over\_n(w, cur\_val)$ 
else begin  $r \leftarrow new\_spec(s)$ ;
if  $q = multiply$  then
begin  $width(r) \leftarrow nx\_plus\_y(width(s), cur\_val, 0)$ ;  $stretch(r) \leftarrow nx\_plus\_y(stretch(s), cur\_val, 0)$ ;
 $shrink(r) \leftarrow nx\_plus\_y(shrink(s), cur\_val, 0)$ ;
end
else begin  $width(r) \leftarrow x\_over\_n(width(s), cur\_val)$ ;  $stretch(r) \leftarrow x\_over\_n(stretch(s), cur\_val)$ ;
 $shrink(r) \leftarrow x\_over\_n(shrink(s), cur\_val)$ ;
end;
 $cur\_val \leftarrow r$ ;
end;
end;

```

This code is used in section 1414.

1419. 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.

\langle Assignments 1395 $\rangle + \equiv$

```

set_box: begin  $scan\_register\_num$ ;
if  $global$  then  $n \leftarrow global\_box\_flag + cur\_val$  else  $n \leftarrow box\_flag + cur\_val$ ;
 $scan\_optional\_equals$ ;
if  $set\_box\_allowed$  then  $scan\_box(n)$ 
else begin  $print\_err("Improper \setbox")$ ;  $print\_esc("setbox")$ ;
 $help2("Sorry, \setbox is not allowed after \halign in a display,")$ 
 $("or between \accent and an accented character.")$ ;  $error$ ;
end;
end;

```

1420. The *space_factor* or *prev_depth* settings are changed when a *set_aux* command is sensed. Similarly, *prev_graf* is changed in the presence of *set_prev_graf*, and *dead_cycles* or *insert_penalties* in the presence of *set_page_int*. These definitions are always global.

When some dimension of a box register is changed, the change isn't exactly global; but TeX does not look at the \global switch.

```
<Assignments 1395> +≡
set_aux: alter_aux;
set_prev_graf: alter_prev_graf;
set_page_dimen: alter_page_so_far;
set_page_int: alter_integer;
set_box_dimen: alter_box_dimen;
```

1421. < Declare subprocedures for *prefixed_command* 1393 > +≡

```
procedure alter_aux;
  var c: halfword; { hmode or vmode }
  begin if cur_chr ≠ abs(mode) then report_illegal_case
  else begin c ← cur_chr; scan_optional_equals;
    if c = vmode then
      begin scan_normal_dimen; prev_depth ← cur_val;
      end
    else begin scan_int;
      if (cur_val ≤ 0) ∨ (cur_val > 32767) then
        begin print_err("Bad_space_factor");
        help1("I_allow_only_values_in_the_range_1..32767_here.");
        int_error(cur_val);
        end
      else space_factor ← cur_val;
      end;
    end;
  end;
```

1422. < Declare subprocedures for *prefixed_command* 1393 > +≡

```
procedure alter_prev_graf;
  var p: 0 .. nest_size; { index into nest }
  begin nest[nest_ptr] ← cur_list; p ← nest_ptr;
  while abs(nest[p].mode_field) ≠ vmode do decr(p);
  scan_optional_equals; scan_int;
  if cur_val < 0 then
    begin print_err("Bad_");
    print_esc("prevgraf");
    help1("I_allow_only_nonnegative_values_here.");
    int_error(cur_val);
    end
  else begin nest[p].pg_field ← cur_val; cur_list ← nest[nest_ptr];
  end;
end;
```

1423. < Declare subprocedures for *prefixed_command* 1393 > +≡

```
procedure alter_page_so_far;
  var c: 0 .. 7; { index into page_so_far }
  begin c ← cur_chr; scan_optional_equals; scan_normal_dimen; page_so_far[c] ← cur_val;
end;
```

1424. \langle Declare subprocedures for *prefixed_command* 1393 $\rangle +\equiv$

```
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 1696 }
  else insert_penalties ← cur_val;
  end;
```

1425. \langle Declare subprocedures for *prefixed_command* 1393 $\rangle +\equiv$

```
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;
```

1426. Paragraph shapes are set up in the obvious way.

\langle Assignments 1395 $\rangle +\equiv$

```
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;
```

1427. Here's something that isn't quite so obvious. It guarantees that *info(par_shape_ptr)* can hold any positive *n* for which *get_node(2 * n + 1)* doesn't overflow the memory capacity.

\langle Check the "constant" values for consistency 14 $\rangle +\equiv$

```
if 2 * max_halfword < mem_top - mem_min then bad ← 41;
```

1428. New hyphenation data is loaded by the *hyph_data* command.

\langle Put each of T_EX's primitives into the hash table 244 $\rangle +\equiv$

```
primitive("hyphenation", hyph_data, 0); primitive("patterns", hyph_data, 1);
```

1429. \langle Cases of *print_cmd_chr* for symbolic printing of primitives 245 $\rangle +\equiv$

```
hyph_data: if chr_code = 1 then print_esc("patterns")
  else print_esc("hyphenation");
```

1430. $\langle \text{Assignments } 1395 \rangle + \equiv$

```

hyph_data: if cur_chr = 1 then
  begin init new_patterns; goto done; tini
  print_err("Patterns\u2022can\u2022be\u2022loaded\u2022only\u2022by\u2022INITEX"); help0; error;
  repeat get_token;
  until cur_cmd = right_brace; { flush the patterns }
  return;
end
else begin new_hyph_exceptions; goto done;
end;
```

1431. All of T_EX's parameters are kept in *eqtb* except the font information, the interaction mode, and the hyphenation tables; these are strictly global.

```

<Assignments 1395> +≡
assign_font_dimen: begin find_font_dimen(true); k ← cur_val; scan_optional_equals; scan_normal_dimen;
  font_info[k].sc ← cur_val;
end;
assign_font_int: begin n ← cur_chr; scan_font_ident; f ← cur_val;
  if n = no_lig_code then set_no_ligatures(f)
  else if n < lp_code_base then
    begin scan_optional_equals; scan_int;
    if n = 0 then hyphen_char[f] ← cur_val else skew_char[f] ← cur_val;
    end
  else begin scan_char_num; p ← cur_val; scan_optional_equals; scan_int;
    case n of
      lp_code_base: set_lp_code(f, p, cur_val);
      rp_code_base: set_rp_code(f, p, cur_val);
      ef_code_base: set_ef_code(f, p, cur_val);
      tag_code: set_tag_code(f, p, cur_val);
      kn_bs_code_base: set_kn_bs_code(f, p, cur_val);
      st_bs_code_base: set_st_bs_code(f, p, cur_val);
      sh_bs_code_base: set_sh_bs_code(f, p, cur_val);
      kn_bc_code_base: set_kn_bc_code(f, p, cur_val);
      kn_ac_code_base: set_kn_ac_code(f, p, cur_val);
    end;
  end;
end;
```

1432. $\langle \text{Put each of T}_E\text{X's primitives into the hash table } 244 \rangle + \equiv$

```

primitive("hyphenchar", assign_font_int, 0); primitive("skewchar", assign_font_int, 1);
primitive("lpcode", assign_font_int, lp_code_base); primitive("rpcode", assign_font_int, rp_code_base);
primitive("efcode", assign_font_int, ef_code_base); primitive("tagcode", assign_font_int, tag_code);
primitive("knbscode", assign_font_int, kn_bs_code_base);
primitive("stbscode", assign_font_int, st_bs_code_base);
primitive("shbscode", assign_font_int, sh_bs_code_base);
primitive("knbccode", assign_font_int, kn_bc_code_base);
primitive("knaccode", assign_font_int, kn_ac_code_base);
primitive("pdfnoligatures", assign_font_int, no_lig_code);
```

1433. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡
assign_font_int: **case** *chr_code* **of**

```
0: print_esc("hyphenchar");
1: print_esc("skewchar");
lp_code_base: print_esc("lpcode");
rp_code_base: print_esc("rancode");
ef_code_base: print_esc("efcode");
tag_code: print_esc("tagcode");
kn_bs_code_base: print_esc("knbscode");
st_bs_code_base: print_esc("stbscode");
sh_bs_code_base: print_esc("shbscode");
kn_bc_code_base: print_esc("knbccode");
kn_ac_code_base: print_esc("knaccode");
no_lig_code: print_esc("pdfnoligatures");
endcases;
```

1434. Here is where the information for a new font gets loaded.

⟨Assignments 1395⟩ +≡
def_font: *new_font*(*a*);
letterspace_font: *new_letterspaced_font*(*a*);
pdf_copy_font: *make_font_copy*(*a*);

1435. ⟨Declare subprocedures for *prefixed_command* 1393⟩ +≡
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 ← cur_cs;
  if u ≥ hash_base then t ← text(u)
  else if u ≥ single_base then
    if u = null_cs then t ← "FONT" else t ← u − single_base
    else begin old_setting ← selector; selector ← new_string; print("FONT"); print(u − active_base);
      selector ← old_setting; str_room(1); t ← make_string;
    end;
  define(u, set_font, null_font); scan_optional_equals; scan_file_name;
  ⟨ Scan the font size specification 1436 ⟩;
  ⟨ If this font has already been loaded, set f to the internal font number and goto common_ending 1438 ⟩;
  f ← read_font_info(u, cur_name, cur_area, s);
common_ending: define(u, set_font, f); eqtb[font_id_base + f] ← eqtb[u]; font_id_text(f) ← t;
end;
```

1436. \langle Scan the font size specification 1436 $\rangle \equiv$

```

name_in_progress  $\leftarrow$  true; { this keeps cur_name from being changed }
if scan_keyword("at") then < Put the (positive) 'at' size into s 1437 >
else if scan_keyword("scaled") then
begin scan_int; s  $\leftarrow$  -cur_val;
if (cur_val  $\leq$  0)  $\vee$  (cur_val  $>$  32768) then
begin print_err("Illegal_magnification_has_been_changed_to_1000");
help1("The_magnification_ratio_must_be_between_1_and_32768."); int_error(cur_val);
s  $\leftarrow$  -1000;
end;
end
else s  $\leftarrow$  -1000;
name_in_progress  $\leftarrow$  false

```

This code is used in section 1435.

1437. \langle Put the (positive) 'at' size into s 1437 $\rangle \equiv$

```

begin scan_normal_dimen; s  $\leftarrow$  cur_val;
if (s  $\leq$  0)  $\vee$  (s  $\geq$  1000000000) then
begin print_err("Improper_at_size"); print_scaled(s); print("pt"), replaced_by_10pt);
help2("I_can_only_handle_fonts_at_positive_sizes_that_are"
("less_than_2048pt, so I've changed what you said to 10pt."); error; s  $\leftarrow$  10 * unity;
end;
end

```

This code is used in section 1436.

1438. When the user gives a new identifier to a font that was previously loaded, the new name becomes the font identifier of record. Font names 'xyz' and 'XYZ' are considered to be different.

\langle If this font has already been loaded, set f to the internal font number and goto common_ending 1438 $\rangle \equiv$

```

flushable_string  $\leftarrow$  str_ptr - 1;
for f  $\leftarrow$  font_base + 1 to font_ptr do
if str_eq_str(font_name[f], cur_name)  $\wedge$  str_eq_str(font_area[f], cur_area) then
begin if cur_name = flushable_string then
begin flush_string; cur_name  $\leftarrow$  font_name[f];
end;
if s  $>$  0 then
begin if s = font_size[f] then goto common_ending;
end
else if font_size[f] = xn_over_d(font_dsize[f], -s, 1000) then goto common_ending;
end

```

This code is used in section 1435.

1439. \langle Cases of print_cmd_chr for symbolic printing of primitives 245 $\rangle + \equiv$

```

set_font: begin print("select_font"); slow_print(font_name[chr_code]);
if font_size[chr_code]  $\neq$  font_dsize[chr_code] then
begin print(" at "); print_scaled(font_size[chr_code]); print("pt");
end;
end;

```

1440. *⟨ Put each of T_EX’s primitives into the hash table 244 ⟩ +≡*

```
primitive("batchmode", set_interaction, batch_mode);
primitive("nonstopmode", set_interaction, nonstop_mode);
primitive("scrollmode", set_interaction, scroll_mode);
primitive("errorstopmode", set_interaction, error_stop_mode);
```

1441. *⟨ Cases of print_cmd_chr for symbolic printing of primitives 245 ⟩ +≡*

```
set_interaction: case chr_code of
  batch_mode: print_esc("batchmode");
  nonstop_mode: print_esc("nonstopmode");
  scroll_mode: print_esc("scrollmode");
  othercases print_esc("errorstopmode")
endcases;
```

1442. *⟨ Assignments 1395 ⟩ +≡*

```
set_interaction: new_interaction;
```

1443. *⟨ Declare subprocedures for prefixed_command 1393 ⟩ +≡*

```
procedure new_interaction;
begin print_ln; interaction ← cur_chr; ⟨ Initialize the print selector based on interaction 75 ⟩;
if log_opened then selector ← selector + 2;
end;
```

1444. The \afterassignment command puts a token into the global variable *after_token*. This global variable is examined just after every assignment has been performed.

⟨ Global variables 13 ⟩ +≡

```
after_token: halfword; { zero, or a saved token }
```

1445. *⟨ Set initial values of key variables 21 ⟩ +≡*

```
after_token ← 0;
```

1446. *⟨ Cases of main_control that don’t depend on mode 1388 ⟩ +≡*

```
any_mode(after_assignment): begin get_token; after_token ← cur_tok;
end;
```

1447. *⟨ Insert a token saved by \afterassignment, if any 1447 ⟩ ≡*

```
if after_token ≠ 0 then
begin cur_tok ← after_token; back_input; after_token ← 0;
end
```

This code is used in section 1389.

1448. Here is a procedure that might be called ‘Get the next non-blank non-relax non-call non-assignment token’.

⟨ Declare action procedures for use by main_control 1221 ⟩ +≡

```
procedure do_assignments;
label exit;
begin loop
begin ⟨ Get the next non-blank non-relax non-call token 430 ⟩;
if cur_cmd ≤ max_non_prefixed_command then return;
set_box_allowed ← false; prefixed_command; set_box_allowed ← true;
end;
exit: end;
```

1449. \langle Cases of *main_control* that don't depend on mode 1388 $\rangle + \equiv$
any_mode(after_group): **begin** *get_token*; *save_for_after*(*cur_tok*);
end;

1450. Files for \read are opened and closed by the *in_stream* command.

\langle Put each of TeX's primitives into the hash table 244 $\rangle + \equiv$
primitive("openin", *in_stream*, 1); *primitive*("closein", *in_stream*, 0);

1451. \langle Cases of *print_cmd_chr* for symbolic printing of primitives 245 $\rangle + \equiv$
in_stream: **if** *chr_code* = 0 **then** *print_esc*("closein")
else *print_esc*("openin");

1452. \langle Cases of *main_control* that don't depend on mode 1388 $\rangle + \equiv$
any_mode(in_stream): *open_or_close_in*;

1453. \langle Declare action procedures for use by *main_control* 1221 $\rangle + \equiv$
procedure *open_or_close_in*;
var *c*: 0 .. 1; {1 for \openin, 0 for \closein}
n: 0 .. 15; {stream number}
begin *c* \leftarrow *cur_chr*; *scan_four_bit_int*; *n* \leftarrow *cur_val*;
if *read_open*[*n*] \neq *closed* **then**
begin *a_close*(*read_file*[*n*]); *read_open*[*n*] \leftarrow *closed*;
end;
if *c* \neq 0 **then**
begin *scan_optional_equals*; *scan_file_name*;
if *cur_ext* = "" **then** *cur_ext* \leftarrow ".tex";
pack_cur_name;
if *a_open_in*(*read_file*[*n*]) **then** *read_open*[*n*] \leftarrow *just_open*;
end;
end;

1454. The user can issue messages to the terminal, regardless of the current mode.

\langle Cases of *main_control* that don't depend on mode 1388 $\rangle + \equiv$
any_mode(message): *issue_message*;

1455. \langle Put each of TeX's primitives into the hash table 244 $\rangle + \equiv$
primitive("message", *message*, 0); *primitive*("errmessage", *message*, 1);

1456. \langle Cases of *print_cmd_chr* for symbolic printing of primitives 245 $\rangle + \equiv$
message: **if** *chr_code* = 0 **then** *print_esc*("message")
else *print_esc*("errmessage");

1457. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡
procedure *issue_message*;

```
var old_setting: 0 .. max_selector; { holds selector setting }
  c: 0 .. 1; { identifies \message and \errmessage }
  s: str_number; { the message }
begin c ← cur_chr; link(garbage) ← scan_toks(false, true); old_setting ← selector;
  selector ← new_string; token_show(def_ref); selector ← old_setting; flush_list(def_ref); str_room(1);
  s ← make_string;
  if c = 0 then ⟨ Print string s on the terminal 1458 ⟩
  else ⟨ Print string s as an error message 1461 ⟩;
  flush_string;
end;
```

1458. ⟨ Print string *s* on the terminal 1458 ⟩ ≡

```
begin if term_offset + length(s) > max_print_line - 2 then print_ln
else if (term_offset > 0) ∨ (file_offset > 0) then print_char(" ");
  slow_print(s); update_terminal;
end
```

This code is used in section 1457.

1459. If \errmessage occurs often in *scroll_mode*, without user-defined \errhelp, we don't want to give a long help message each time. So we give a verbose explanation only once.

⟨ Global variables 13 ⟩ +≡
long_help_seen: boolean; { has the long \errmessage help been used? }

1460. ⟨ Set initial values of key variables 21 ⟩ +≡

```
long_help_seen ← false;
```

1461. ⟨ Print string *s* as an error message 1461 ⟩ ≡

```
begin print_err(""); slow_print(s);
if err_help ≠ null then use_err_help ← true
else if long_help_seen then help1("That was another \errmessage.)")
else begin if interaction < error_stop_mode then long_help_seen ← true;
  help4("This \error \message was generated by an \errmessage")
  ("command, so I can't give any explicit help.")
  ("Pretend that you're Hercule Poirot: Examine all clues,")
  ("and deduce the truth by order and method.");
end;
error; use_err_help ← false;
end
```

This code is used in section 1457.

1462. The *error* routine calls on *give_err_help* if help is requested from the *err_help* parameter.

```
procedure give_err_help;
begin token_show(err_help);
end;
```

1463. The \uppercase and \lowercase commands are implemented by building a token list and then changing the cases of the letters in it.

⟨ Cases of *main_control* that don't depend on mode 1388 ⟩ +≡
any_mode(case_shift): *shift_case*;

1464. \langle Put each of TeX's primitives into the hash table 244 $\rangle + \equiv$
`primitive("lowercase", case_shift, lc_code_base); primitive("uppercase", case_shift, uc_code_base);`

1465. \langle Cases of `print_cmd_chr` for symbolic printing of primitives 245 $\rangle + \equiv$
`case_shift: if chr_code = lc_code_base then print_esc("lowercase")
 else print_esc("uppercase");`

1466. \langle Declare action procedures for use by `main_control` 1221 $\rangle + \equiv$
`procedure shift_case;`

```

var b: pointer; { lc_code_base or uc_code_base }
  p: pointer; { runs through the token list }
  t: halfword; { token }
  c: eight_bits; { character code }
begin b ← cur_chr; p ← scan_toks(false, false); p ← link(def_ref);
while p ≠ null do
  begin { Change the case of the token in p, if a change is appropriate 1467 };
    p ← link(p);
  end;
  back_list(link(def_ref)); free_avail(def_ref); { omit reference count }
end;
```

1467. When the case of a `chr_code` changes, we don't change the `cmd`. We also change active characters, using the fact that `cs_token_flag + active_base` is a multiple of 256.

\langle Change the case of the token in `p`, if a change is appropriate 1467 $\rangle \equiv$
`t ← info(p);`
`if t < cs_token_flag + single_base then`
 `begin c ← t mod 256;`
 `if equiv(b + c) ≠ 0 then info(p) ← t - c + equiv(b + c);`
 `end`

This code is used in section 1466.

1468. We come finally to the last pieces missing from `main_control`, namely the '`\show`' commands that are useful when debugging.

\langle Cases of `main_control` that don't depend on mode 1388 $\rangle + \equiv$
`any_mode(xray): show_whatever;`

1469. `define show_code = 0 { \show }`
`define show_box_code = 1 { \showbox }`
`define show_the_code = 2 { \showthe }`
`define show_lists_code = 3 { \showlists }`

\langle Put each of TeX's primitives into the hash table 244 $\rangle + \equiv$
`primitive("show", xray, show_code); primitive("showbox", xray, show_box_code);`
`primitive("showthe", xray, show_the_code); primitive("showlists", xray, show_lists_code);`

1470. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡
xray: **case** *chr_code* **of**

```
show_box_code: print_esc("showbox");
show_the_code: print_esc("showthe");
show_lists_code: print_esc("showlists");
  ⟨Cases of xray for print_cmd_chr 1676⟩
othercases print_esc("show")
endcases;
```

1471. ⟨Declare action procedures for use by *main_control* 1221⟩ +≡
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_code: begin begin_diagnostic; show_activities;
  end;
show_box_code: ⟨Show the current contents of a box 1474⟩;
show_code: ⟨Show the current meaning of a token, then goto common_ending 1472⟩;
  ⟨Cases for show_whatever 1677⟩
othercases ⟨Show the current value of some parameter or register, then goto common_ending 1475⟩
endcases;
⟨Complete a potentially long \show command 1476⟩;
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;
```

1472. \langle Show the current meaning of a token, then **goto** *common-ending* 1472 $\rangle \equiv$

```

begin get_token;
if interaction = error_stop_mode then wake_up_terminal;
print_nl(">_");
if cur_cs ≠ 0 then
  begin sprint_cs(cur_cs); print_char("=");
  end;
print_meaning; goto common-ending;
end

```

This code is used in section 1471.

1473. \langle Cases of *print_cmd_chr* for symbolic printing of primitives 245 $\rangle +\equiv$

```

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");

```

1474. \langle Show the current contents of a box 1474 $\rangle \equiv$

```

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 1471.

1475. \langle Show the current value of some parameter or register, then **goto** *common-ending* 1475 $\rangle \equiv$

```

begin p ← the_toks;
if interaction = error_stop_mode then wake_up_terminal;
print_nl(">_"); token_show(temp_head); flush_list(link(temp_head)); goto common-ending;
end

```

This code is used in section 1471.

1476. \langle Complete a potentially long \show command 1476 $\rangle \equiv$

```

end_diagnostic(true); print_err("OK");
if selector = term_and_log then
  if tracing_online ≤ 0 then
    begin selector ← term_only; print("_see_the_transcript_file"); selector ← term_and_log;
    end

```

This code is used in section 1471.

1477. Dumping and undumping the tables. After INITEX has seen a collection of fonts and macros, it can write all the necessary information on an auxiliary file so that production versions of T_EX are able to initialize their memory at high speed. The present section of the program takes care of such output and input. We shall consider simultaneously the processes of storing and restoring, so that the inverse relation between them is clear.

The global variable *format_ident* is a string that is printed right after the *banner* line when T_EX is ready to start. For INITEX this string says simply '(INITEX)'; for other versions of T_EX it says, for example, '(preloaded format=plain 1982.11.19)', showing the year, month, and day that the format file was created. We have *format_ident* = 0 before T_EX's tables are loaded.

```
(Global variables 13) +≡
format_ident: str_number;
```

1478. ⟨ Set initial values of key variables 21 ⟩ +≡
 format_ident ← 0;

1479. ⟨ Initialize table entries (done by INITEX only) 182 ⟩ +≡
 format_ident ← "_(INITEX)";

1480. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡
 init procedure *store_fmt_file*;
 label *found1*,*found2*,*done1*,*done2*;
 var *j,k,l*: integer; { all-purpose indices }
 p,q: pointer; { all-purpose pointers }
 x: integer; { something to dump }
 w: four_quarters; { four ASCII codes }
 begin ⟨ If dumping is not allowed, abort 1482 ⟩;
 ⟨ Create the *format_ident*, open the format file, and inform the user that dumping has begun 1508 ⟩;
 ⟨ Dump constants for consistency check 1485 ⟩;
 ⟨ Dump the string pool 1487 ⟩;
 ⟨ Dump the dynamic memory 1489 ⟩;
 ⟨ Dump the table of equivalents 1491 ⟩;
 ⟨ Dump the font information 1498 ⟩;
 ⟨ Dump the hyphenation tables 1502 ⟩;
 ⟨ Dump pdftex data 1504 ⟩;
 ⟨ Dump a couple more things and the closing check word 1506 ⟩;
 ⟨ Close the format file 1509 ⟩;
 end;
 tini

1481. Corresponding to the procedure that dumps a format file, we have a function that reads one in. The function returns *false* if the dumped format is incompatible with the present T_EX table sizes, etc.

```
define bad_fmt = 6666 { go here if the format file is unacceptable }
define too_small(#) ≡
    begin wake_up_terminal; wterm_ln(`---! Must increase the `,#); goto bad_fmt;
    end

⟨ Declare the function called open_fmt_file 550 ⟩
function load_fmt_file: boolean;
label bad_fmt, exit;
var j, k: integer; { all-purpose indices }
p, q: pointer; { all-purpose pointers }
x: integer; { something undumped }
w: four_quarters; { four ASCII codes }
begin ⟨ Undump constants for consistency check 1486 ⟩;
⟨ Undump the string pool 1488 ⟩;
⟨ Undump the dynamic memory 1490 ⟩;
⟨ Undump the table of equivalents 1492 ⟩;
⟨ Undump the font information 1499 ⟩;
⟨ Undump the hyphenation tables 1503 ⟩;
⟨ Undump pdftex data 1505 ⟩;
⟨ Undump a couple more things and the closing check word 1507 ⟩;
prev_depth ← pdf_ignored_dimen; load_fmt_file ← true; return; { it worked! }
bad_fmt: wake_up_terminal; wterm_ln(`(Fatal_format_file_error; I ``m_stymied)` );
load_fmt_file ← false;
exit: end;
```

1482. The user is not allowed to dump a format file unless *save_ptr* = 0. This condition implies that *cur_level* = *level_one*, hence the *xeq_level* array is constant and it need not be dumped.

```
⟨ If dumping is not allowed, abort 1482 ⟩ ≡
if save_ptr ≠ 0 then
    begin print_err("You can't dump inside a group"); help1(``{...}\dump`` is a no-no.");
    succumb;
end
```

This code is used in section 1480.

1483. Format files consist of *memory_word* items, and we use the following macros to dump words of different types:

```
define dump_wd(#) ≡
    begin fmt_file↑ ← #; put(fmt_file); end
define dump_int(#) ≡
    begin fmt_file↑.int ← #; put(fmt_file); end
define dump_hh(#) ≡
    begin fmt_file↑.hh ← #; put(fmt_file); end
define dump_qqqq(#) ≡
    begin fmt_file↑.qqqq ← #; put(fmt_file); end

⟨ Global variables 13 ⟩ +≡
fmt_file: word_file; { for input or output of format information }
```

1484. The inverse macros are slightly more complicated, since we need to check the range of the values we are reading in. We say ‘*undump(a)(b)(x)*’ to read an integer value *x* that is supposed to be in the range $a \leq x \leq b$. System error messages should be suppressed when undumping.

```
define undump_wd(#{@}) ≡
    begin get(fmt_file); #{@} ← fmt_file↑; end
define undump_int(#{@}) ≡
    begin get(fmt_file); #{@} ← fmt_file↑.int; end
define undump_hh(#{@}) ≡
    begin get(fmt_file); #{@} ← fmt_file↑.hh; end
define undump_qqqq(#{@}) ≡
    begin get(fmt_file); #{@} ← fmt_file↑.qqqq; end
define undump_end_end(#{@}) ≡ #{@} ← x; end
define undump_end(#{@}) ≡ (x > #{@}) then goto bad_fmt else undump_end_end
define undump(#{@}) ≡
    begin undump_int(x);
    if (x < #{@}) ∨ undump_end
define undump_size_end_end(#{@}) ≡ too_small(#{@}) else undump_end_end
define undump_size_end(#{@}) ≡
    if x > #{@} then undump_size_end_end
define undump_size(#{@}) ≡
    begin undump_int(x);
    if x < #{@} then goto bad_fmt;
    undump_size_end
```

1485. The next few sections of the program should make it clear how we use the dump/undump macros.

```
⟨ Dump constants for consistency check 1485 ⟩ ≡
    dump_int(@$);
    ⟨ Dump the ε-TEX state 1654 ⟩
    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 1480.

1486. 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 1486 ⟩ ≡

```

 $x \leftarrow \text{fmt\_file}^\uparrow.\text{int};$ 
if  $x \neq @\$$  then goto bad_fmt; { check that strings are the same }
⟨ Undump the ε-TEx state 1655 ⟩
undump_int( $x$ );
if  $x \neq \text{mem\_bot}$  then goto bad_fmt;
undump_int( $x$ );
if  $x \neq \text{mem\_top}$  then goto bad_fmt;
undump_int( $x$ );
if  $x \neq \text{eqtb\_size}$  then goto bad_fmt;
undump_int( $x$ );
if  $x \neq \text{hash\_prime}$  then goto bad_fmt;
undump_int( $x$ );
if  $x \neq \text{hyph\_size}$  then goto bad_fmt

```

This code is used in section 1481.

1487. define dump_four_ASCII ≡ $w.b0 \leftarrow \text{qi}(\text{so}(\text{str_pool}[k]))$; $w.b1 \leftarrow \text{qi}(\text{so}(\text{str_pool}[k + 1]))$;
 $w.b2 \leftarrow \text{qi}(\text{so}(\text{str_pool}[k + 2]))$; $w.b3 \leftarrow \text{qi}(\text{so}(\text{str_pool}[k + 3]))$; dump_qqqq(w)

⟨ Dump the string pool 1487 ⟩ ≡

```

dump_int(pool_ptr); dump_int(str_ptr);
for  $k \leftarrow 0$  to str_ptr do dump_int(str_start[ $k$ ]);
 $k \leftarrow 0$ ;
while  $k + 4 < \text{pool\_ptr}$  do
  begin dump_four_ASCII;  $k \leftarrow k + 4$ ;
  end;
 $k \leftarrow \text{pool\_ptr} - 4$ ; dump_four_ASCII; print_ln; print_int(str_ptr);
print("＼strings＼of＼total＼length＼"); print_int(pool_ptr)

```

This code is used in section 1480.

1488. define undump_four_ASCII ≡ undump_qqqq(w); str_pool[k] ← si(qo($w.b0$));
 $\text{str_pool}[k + 1] \leftarrow \text{si}(\text{qo}(w.b1))$; $\text{str_pool}[k + 2] \leftarrow \text{si}(\text{qo}(w.b2))$; $\text{str_pool}[k + 3] \leftarrow \text{si}(\text{qo}(w.b3))$

⟨ Undump the string pool 1488 ⟩ ≡

```

undump_size(0)(pool_size)('string_pool_size')(pool_ptr);
undump_size(0)(max_strings)('max_strings')(str_ptr);
for  $k \leftarrow 0$  to str_ptr do undump(0)(pool_ptr)(str_start[ $k$ ]);
 $k \leftarrow 0$ ;
while  $k + 4 < \text{pool\_ptr}$  do
  begin undump_four_ASCII;  $k \leftarrow k + 4$ ;
  end;
 $k \leftarrow \text{pool\_ptr} - 4$ ; undump_four_ASCII; init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr

```

This code is used in section 1481.

1489. 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 1489 ⟩ ≡
  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 1480.

1490. ⟨ Undump the dynamic memory 1490 ⟩ ≡

```

  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 1481.

1491. \langle Dump the table of equivalents 1491 $\rangle \equiv$
 \langle Dump regions 1 to 4 of eqtb 1493 $\rangle;$
 \langle Dump regions 5 and 6 of eqtb 1494 $\rangle;$
 $dump_int(par_loc); dump_int(write_loc);$
 \langle Dump the hash table 1496 \rangle

This code is used in section 1480.

1492. \langle Undump the table of equivalents 1492 $\rangle \equiv$
 \langle Undump regions 1 to 6 of eqtb 1495 $\rangle;$
 $undump(hash_base)(frozen_control_sequence)(par_loc); par_token \leftarrow cs_token_flag + par_loc;$
 $undump(hash_base)(frozen_control_sequence)(write_loc);$
 \langle Undump the hash table 1497 \rangle

This code is used in section 1481.

1493. The table of equivalents usually contains repeated information, so we dump it in compressed form:
The sequence of $n + 2$ values (n, x_1, \dots, x_n, m) in the format file represents $n + m$ consecutive entries of eqtb,
with m extra copies of x_n , namely $(x_1, \dots, x_n, x_n, \dots, x_n)$.

```

⟨ Dump regions 1 to 4 of eqtb 1493 ⟩ ≡
  k ← active_base;
  repeat j ← k;
    while j < int_base - 1 do
      begin if (equiv(j) = equiv(j + 1)) ∧ (eq_type(j) = eq_type(j + 1)) ∧ (eq_level(j) = eq_level(j + 1))
             then goto found1;
      incr(j);
      end;
    l ← int_base; goto done1; { j = int_base - 1 }
  found1: incr(j); l ← j;
    while j < int_base - 1 do
      begin if (equiv(j) ≠ equiv(j + 1)) ∨ (eq_type(j) ≠ eq_type(j + 1)) ∨ (eq_level(j) ≠ eq_level(j + 1))
             then goto done1;
      incr(j);
      end;
  done1: dump_int(l - k);
    while k < l do
      begin dump_wd(eqt[k]); incr(k);
      end;
    k ← j + 1; dump_int(k - l);
  until k = int_base

```

This code is used in section 1491.

1494. \langle Dump regions 5 and 6 of *eqtb* 1494 $\rangle \equiv$

```

repeat  $j \leftarrow k$ ;
  while  $j < eqtb\_size$  do
    begin if eqtb[ $j$ ].int = eqtb[ $j + 1$ ].int then goto found2;
      incr( $j$ );
      end;
     $l \leftarrow eqtb\_size + 1$ ; goto done2; {  $j = eqtb\_size$  }
found2: incr( $j$ );  $l \leftarrow j$ ;
  while  $j < eqtb\_size$  do
    begin if eqtb[ $j$ ].int  $\neq eqtb[j + 1]$ .int then goto done2;
      incr( $j$ );
      end;
done2: dump_int( $l - k$ );
  while  $k < l$  do
    begin dump_wd(eqtb[ $k$ ])); incr( $k$ );
    end;
 $k \leftarrow j + 1$ ; dump_int( $k - l$ );
until  $k > eqtb\_size$ 

```

This code is used in section 1491.

1495. \langle Undump regions 1 to 6 of *eqtb* 1495 $\rangle \equiv$

```

 $k \leftarrow active\_base$ ;
repeat undump_int( $x$ );
  if  $(x < 1) \vee (k + x > eqtb\_size + 1)$  then goto bad_fmt;
  for  $j \leftarrow k$  to  $k + x - 1$  do undump_wd(eqtb[ $j$ ])));
   $k \leftarrow k + x$ ; undump_int( $x$ );
  if  $(x < 0) \vee (k + x > eqtb\_size + 1)$  then goto bad_fmt;
  for  $j \leftarrow k$  to  $k + x - 1$  do eqtb[ $j$ ]  $\leftarrow eqtb[k - 1]$ ;
   $k \leftarrow k + x$ ;
until  $k > eqtb\_size$ 

```

This code is used in section 1492.

1496. A different scheme is used to compress the hash table, since its lower region is usually sparse. When $text(p) \neq 0$ for $p \leq hash_used$, we output two words, p and $hash[p]$. The hash table is, of course, densely packed for $p \geq hash_used$, so the remaining entries are output in a block.

\langle Dump the hash table 1496 $\rangle \equiv$

```

for  $p \leftarrow 0$  to prim_size do dump_hh(prim[ $p$ ])));
dump_int(hash_used); cs_count  $\leftarrow frozen\_control\_sequence - 1 - hash\_used$ ;
for  $p \leftarrow hash\_base$  to hash_used do
  if text( $p$ )  $\neq 0$  then
    begin dump_int( $p$ ); dump_hh(hash[ $p$ ])); incr(cs_count);
    end;
for  $p \leftarrow hash\_used + 1$  to undefined_control_sequence - 1 do dump_hh(hash[ $p$ ])));
dump_int(cs_count);
print_ln; print_int(cs_count); print(" multiletter control sequences")

```

This code is used in section 1491.

1497. \langle Undump the hash table 1497 $\rangle \equiv$

```

for  $p \leftarrow 0$  to  $prim\_size$  do  $undump\_hh(prim[p]);$ 
 $undump(hash\_base)(frozen\_control\_sequence)(hash\_used); p \leftarrow hash\_base - 1;$ 
repeat  $undump(p + 1)(hash\_used)(p); undump\_hh(hash[p]);$ 
until  $p = hash\_used;$ 
for  $p \leftarrow hash\_used + 1$  to  $undefined\_control\_sequence - 1$  do  $undump\_hh(hash[p]);$ 
 $undump\_int(cs\_count)$ 

```

This code is used in section 1492.

1498. \langle Dump the font information 1498 $\rangle \equiv$

```

 $dump\_int(fmem\_ptr);$ 
for  $k \leftarrow 0$  to  $fmem\_ptr - 1$  do  $dump\_wd(font\_info[k]);$ 
 $dump\_int(font\_ptr);$ 
for  $k \leftarrow null\_font$  to  $font\_ptr$  do  $\langle$  Dump the array info for internal font number  $k$  1500  $\rangle;$ 
 $print\_ln; print\_int(fmem\_ptr - 7); print("words of font info for");$ 
 $print\_int(font\_ptr - font\_base); print(" preloaded font");$ 
if  $font\_ptr \neq font\_base + 1$  then  $print\_char("s")$ 

```

This code is used in section 1480.

1499. \langle Undump the font information 1499 $\rangle \equiv$

```

 $undump\_size(7)(font\_mem\_size)(`font\_mem\_size')(fmem\_ptr);$ 
for  $k \leftarrow 0$  to  $fmem\_ptr - 1$  do  $undump\_wd(font\_info[k]);$ 
 $undump\_size(font\_base)(font\_max)(`font\_max')(font\_ptr);$ 
for  $k \leftarrow null\_font$  to  $font\_ptr$  do  $\langle$  Undump the array info for internal font number  $k$  1501  $\rangle$ 

```

This code is used in section 1481.

1500. \langle Dump the array info for internal font number k 1500 $\rangle \equiv$

```

begin  $dump\_qqqq(font\_check[k]); dump\_int(font\_size[k]); dump\_int(font\_dsizes[k]);$ 
 $dump\_int(font\_params[k]);$ 
 $dump\_int(hyphen\_char[k]); dump\_int(skew\_char[k]);$ 
 $dump\_int(font\_name[k]); dump\_int(font\_area[k]);$ 
 $dump\_int(font\_bc[k]); dump\_int(font\_ec[k]);$ 
 $dump\_int(char\_base[k]); dump\_int(width\_base[k]); dump\_int(height\_base[k]);$ 
 $dump\_int(depth\_base[k]); dump\_int(italic\_base[k]); dump\_int(lig\_kern\_base[k]);$ 
 $dump\_int(kern\_base[k]); dump\_int(exten\_base[k]); dump\_int(param\_base[k]);$ 
 $dump\_int(font\_glue[k]);$ 
 $dump\_int(bchar\_label[k]); dump\_int(font\_bchar[k]); dump\_int(font\_false\_bchar[k]);$ 
 $print\_nl("\font"); print\_esc(font\_id\_text(k)); print\_char("=");$ 
 $print\_file\_name(font\_name[k], font\_area[k], "");$ 
if  $font\_size[k] \neq font\_dsizes[k]$  then
    begin  $print("at"); print\_scaled(font\_size[k]); print("pt");$ 
    end;
end

```

This code is used in section 1498.

1501. ⟨ Undump the array info for internal font number k 1501 ⟩ ≡

```

begin undump_qqqq(font_check[k]);
undump_int(font_size[k]); undump_int(font_dsize[k]);
undump(min_halfword)(max_halfword)(font_params[k]);
undump_int(hyphen_char[k]); undump_int(skew_char[k]);
undump(0)(str_ptr)(font_name[k]); undump(0)(str_ptr)(font_area[k]);
undump(0)(255)(font_bc[k]); undump(0)(255)(font_ec[k]);
undump_int(char_base[k]); undump_int(width_base[k]); undump_int(height_base[k]);
undump_int(depth_base[k]); undump_int(italic_base[k]); undump_int(lig_kern_base[k]);
undump_int(kern_base[k]); undump_int(exten_base[k]); undump_int(param_base[k]);
undump(min_halfword)(lo_mem_max)(font_glue[k]);
undump(0)(fmem_ptr - 1)(bchar_label[k]); undump(min_quarterword)(non_char)(font_bchar[k]);
undump(min_quarterword)(non_char)(font_false_bchar[k]);
end

```

This code is used in section 1499.

1502. ⟨ Dump the hyphenation tables 1502 ⟩ ≡

```

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 1480.

1503. Only “nonempty” parts of *op_start* need to be restored.

```
< Undump the hyphenation tables 1503 > ≡
  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; tini undump(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 1481.

1504. Store some of the pdftex data structures in the format. The idea here is to ensure that any data structures referenced from pdftex-specific whatsit nodes are retained. For the sake of simplicity and speed, all the filled parts of *pdf_mem* and *obj_tab* are retained, in the present implementation. We also retain three of the linked lists that start from *head_tab*, so that it is possible to, say, load an image in the INITEX run and then reference it in a VIRTEX run that uses the dumped format.

```
< Dump pdftex data 1504 > ≡
  begin dumpimagemeta; { the image information array }
  dump_int(pdf_mem_size); dump_int(pdf_mem_ptr);
  for k ← 1 to pdf_mem_ptr - 1 do
    begin dump_int(pdf_mem[k]);
    end;
  print_ln; print_int(pdf_mem_ptr - 1); print("words of pdfTeX memory"); dump_int(obj_tab_size);
  dump_int(obj_ptr); dump_int(sys_obj_ptr);
  for k ← 1 to sys_obj_ptr do
    begin dump_int(obj_tab[k].int0); dump_int(obj_tab[k].int1); dump_int(obj_tab[k].int3);
    dump_int(obj_tab[k].int4);
    end;
  print_ln; print_int(sys_obj_ptr); print("indirect objects"); dump_int(pdf_obj_count);
  dump_int(pdf_xform_count); dump_int(pdf_ximage_count); dump_int(head_tab[obj_type_obj]);
  dump_int(head_tab[obj_type_xform]); dump_int(head_tab[obj_type_ximage]); dump_int(pdf_last_obj);
  dump_int(pdf_last_xform); dump_int(pdf_last_ximage); dumptounicode;
  end
```

This code is used in section 1480.

1505. And restoring the pdftex data structures from the format. The two function arguments to *undumpimagemeta* have been restored already in an earlier module.

```
< Undump pdftex data 1505 > ≡
begin undumpimagemeta(pdf_major_version, pdf_minor_version, pdf_inclusion_errorlevel);
{ the image information array }
undump_int(pdf_mem_size); pdf_mem ← xrealloc_array(pdf_mem, integer, pdf_mem_size);
undump_int(pdf_mem_ptr);
for k ← 1 to pdf_mem_ptr - 1 do
begin undump_int(pdf_mem[k]);
end;
undump_int(obj_tab_size); undump_int(obj_ptr); undump_int(sys_obj_ptr);
for k ← 1 to sys_obj_ptr do
begin undump_int(obj_tab[k].int0); undump_int(obj_tab[k].int1); obj_tab[k].int2 ← -1;
undump_int(obj_tab[k].int3); undump_int(obj_tab[k].int4);
end;
undump_int(pdf_obj_count); undump_int(pdf_xform_count); undump_int(pdf_ximage_count);
undump_int(head_tab[obj_type_obj]); undump_int(head_tab[obj_type_xform]);
undump_int(head_tab[obj_type_ximage]); undump_int(pdf_last_obj); undump_int(pdf_last_xform);
undump_int(pdf_last_ximage); undumptounicode;
end
```

This code is used in section 1481.

1506. We have already printed a lot of statistics, so we set *tracing_stats* ← 0 to prevent them from appearing again.

```
< Dump a couple more things and the closing check word 1506 > ≡
dump_int(interaction); dump_int(format_ident); dump_int(69069); tracing_stats ← 0
```

This code is used in section 1480.

1507. < Undump a couple more things and the closing check word 1507 > ≡
undump(batch_mode)(error_stop_mode)(interaction); undump(0)(str_ptr)(format_ident); undump_int(x);
if (x ≠ 69069) ∨ eof(fmt_file) then goto bad_fmt

This code is used in section 1481.

1508. < Create the *format_ident*, open the format file, and inform the user that dumping has begun 1508 > ≡

```
selector ← new_string; print("↳(preloaded↳format="); print(job_name); print_char("↳");
print_int(year); print_char("."); print_int(month); print_char("."); print_int(day); print_char(")");
if interaction = batch_mode then selector ← log_only
else selector ← term_and_log;
str_room(1); format_ident ← make_string; pack_job_name(format_extension);
while ¬w_open_out(fmt_file) do prompt_file_name("format↳file↳name", format_extension);
print_nl("Beginning↳to↳dump↳on↳file↳"); slow_print(w_make_name_string(fmt_file)); flush_string;
print_nl(""); slow_print(format_ident)
```

This code is used in section 1480.

1509. < Close the format file 1509 > ≡

```
w_close(fmt_file)
```

This code is used in section 1480.

1510. The main program. This is it: the part of TeX that executes all those procedures we have written.

Well—almost. Let's leave space for a few more routines that we may have forgotten.

⟨ Last-minute procedures 1513 ⟩

1511. We have noted that there are two versions of $\text{TeX}82$. One, called INITEX, has to be run first; it initializes everything from scratch, without reading a format file, and it has the capability of dumping a format file. The other one is called ‘VIRTEX’; it is a “virgin” program that needs to input a format file in order to get started. VIRTEX typically has more memory capacity than INITEX, because it does not need the space consumed by the auxiliary hyphenation tables and the numerous calls on *primitive*, etc.

The VIRTEX program cannot read a format file instantaneously, of course; the best implementations therefore allow for production versions of TeX that not only avoid the loading routine for Pascal object code, they also have a format file pre-loaded. This is impossible to do if we stick to standard Pascal; but there is a simple way to fool many systems into avoiding the initialization, as follows: (1) We declare a global integer variable called *ready_already*. The probability is negligible that this variable holds any particular value like 314159 when VIRTEX is first loaded. (2) After we have read in a format file and initialized everything, we set $\text{ready_already} \leftarrow 314159$. (3) Soon VIRTEX will print ‘*’, waiting for more input; and at this point we interrupt the program and save its core image in some form that the operating system can reload speedily. (4) When that core image is activated, the program starts again at the beginning; but now $\text{ready_already} = 314159$ and all the other global variables have their initial values too. The former chastity has vanished!

In other words, if we allow ourselves to test the condition $\text{ready_already} = 314159$, before *ready_already* has been assigned a value, we can avoid the lengthy initialization. Dirty tricks rarely pay off so handsomely.

On systems that allow such preloading, the standard program called TeX should be the one that has plain format preloaded, since that agrees with *The TeXbook*. Other versions, e.g., AmSTeX, should also be provided for commonly used formats.

⟨ Global variables 13 ⟩ +≡

ready_already: integer; { a sacrifice of purity for economy }

1512. Now this is really it: TeX starts and ends here.

The initial test involving *ready_already* should be deleted if the Pascal runtime system is smart enough to detect such a “mistake.”

```

begin { start_here }
history ← fatal_error_stop; { in case we quit during initialization }
t_open_out; { open the terminal for output }
if ready_already = 314159 then goto start_of_TEX;
{ Check the “constant” values for consistency 14 }
if bad > 0 then
  begin wterm_ln(`Ouch---my internal constants have been clobbered!', `---case', bad : 1);
  goto final_end;
  end;
initialize; { set global variables to their starting values }
init if ¬get_strings_started then goto final_end;
init_prim; { call primitive for each primitive }
init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr; fix_date_and_time;
tini
ready_already ← 314159;
start_of_TEX: { Initialize the output routines 55 };
{ Get the first line of input and prepare to start 1517 };
history ← spotless; { ready to go! }
main_control; { come to life }
final_cleanup; { prepare for death }
end_of_TEX: close_files_and_terminate;
final_end: ready_already ← 0;
end.

```

1513. Here we do whatever is needed to complete T_EX's job gracefully on the local operating system. The code here might come into play after a fatal error; it must therefore consist entirely of "safe" operations that cannot produce error messages. For example, it would be a mistake to call *str_room* or *make_string* at this time, because a call on *overflow* might lead to an infinite loop. (Actually there's one way to get error messages, via *prepare_mag*; but that can't cause infinite recursion.)

If *final_cleanup* is bypassed, this program doesn't bother to close the input files that may still be open.

⟨ Last-minute procedures 1513 ⟩ ≡

```

procedure close_files_and_terminate;
label done, done1;
var a, b, c, i, j, k, l: integer; { all-purpose index }
is_root: boolean; { pdf_last_pages is root of Pages tree? }
is_names: boolean; { flag for name tree output: is it Names or Kids? }
root, outlines, threads, names_tree, dests: integer; xref_offset_width, names_head, names_tail: integer;
begin {Finish the extensions 1626};
new_line_char ← -1;
stat if tracing_stats > 0 then ⟨ Output statistics about this job 1514 ⟩; tats
wake_up_terminal;
if ¬fixed_pdfoutput_set then fix_pdfoutput;
if fixed_pdfoutput > 0 then
begin if history = fatal_error_stop then
begin remove_pdffile;
print_err("==> Fatal error occurred, no output PDF file produced!");
end
else begin {Finish the PDF file 794};
if log_opened then
begin wlog_cr; wlog_ln(`PDF_statistics:'); wlog_ln(`', obj_ptr : 1, `PDF_objects_out_of',
obj_tab_size : 1, `',(max., sup_obj_tab_size : 1, `)');
if pdf_os_cntr > 0 then
begin wlog(`', ((pdf_os_cntr - 1) * pdf_os_max_objs + pdf_os_objidx + 1) : 1,
`compressed_objects_within', pdf_os_cntr : 1, `object_stream');
if pdf_os_cntr > 1 then wlog(`s');
wlog_cr;
end;
wlog_ln(`', pdf_dest_names_ptr : 1, `named_destinations_out_of', dest_names_size : 1,
`',(max., sup_dest_names_size : 1, `)');
wlog_ln(`', pdf_mem_ptr : 1, `words_of_extra_memory_for_PDF_output_out_of',
pdf_mem_size : 1, `',(max., sup_pdf_mem_size : 1, `)');
end;
end;
end
else begin {Finish the DVI file 670};
end;
if log_opened then
begin wlog_cr; a_close(log_file); selector ← selector - 2;
if selector = term_only then
begin print_nl("Transcript_written_on"); slow_print(log_name); print_char(".");
end;
end;
end;
```

See also sections 1515, 1516, and 1518.

This code is used in section 1510.

1514. The present section goes directly to the log file instead of using *print* commands, because there's no need for these strings to take up *str_pool* memory when a non-stat version of T_EX is being used.

⟨Output statistics about this job 1514⟩ ≡

```

if log_opened then
begin wlog_ln(` `); wlog_ln(`Here is how much of TeX's memory you used: `);
wlog(` ` , str_ptr - init_str_ptr : 1, `string`);
if str_ptr ≠ init_str_ptr + 1 then wlog(`s`);
wlog_ln(` ` , max_strings - init_str_ptr : 1);
wlog_ln(` ` , pool_ptr - init_pool_ptr : 1, `string`characters`out`of` , pool_size - init_pool_ptr : 1);
wlog_ln(` ` , lo_mem_max - mem_min + mem_end - hi_mem_min + 2 : 1,
`words`of`memory`out`of` , mem_end + 1 - mem_min : 1);
wlog_ln(` ` , cs_count : 1, `multiletter`control`sequences`out`of` , hash_size : 1);
wlog(` ` , fmem_ptr : 1, `words`of`font`info`for` , font_ptr - font_base : 1, `font`);
if font_ptr ≠ font_base + 1 then wlog(`s`);
wlog_ln(` ` , `out`of` , font_mem_size : 1, `for` , font_max - font_base : 1);
wlog(` ` , hyph_count : 1, `hyphenation`exception`);
if hyph_count ≠ 1 then wlog(`s`);
wlog_ln(` ` , `out`of` , hyph_size : 1);
wlog_ln(` ` , max_in_stack : 1, `i` , ` , max_nest_stack : 1, `n` , ` , max_param_stack : 1, `p` ,
max_buf_stack + 1 : 1, `b` , ` , max_save_stack + 6 : 1, `s`stack`positions`out`of` ,
stack_size : 1, `i` , ` , nest_size : 1, `n` , ` , param_size : 1, `p` , ` , buf_size : 1, `b` , ` , save_size : 1, `s` );
end

```

This code is used in section 1513.

1515. We get to the *final_cleanup* routine when `\end` or `\dump` has been scanned and *its_all_over*.

```

⟨ Last-minute procedures 1513 ⟩ +≡
procedure final_cleanup;
label exit;
var c: small_number; { 0 for \end, 1 for \dump }
begin c ← cur_chr;
if c ≠ 1 then new_line_char ← -1;
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;

```

1516. ⟨ Last-minute procedures 1513 ⟩ +≡

```

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 244 ⟩;
no_new_control_sequence ← true;
end;
tini

```

1517. 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 1517 ⟩ ≡
begin ⟨ Initialize the input routines 353 ⟩;
⟨ Enable ε-TEx, if requested 1648 ⟩
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 (pdf_output_option ≠ 0) then pdf_output ← pdf_output_value;
if (pdf_draftmode_option ≠ 0) then pdf_draftmode ← pdf_draftmode_value;
pdf_init_map_file(`pdftex.map`);
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;
random_seed ← (microseconds * 1000) + (epochseconds mod 1000000);
init_randoms(random_seed);
⟨ Compute the magic offset 941 ⟩;
⟨ 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 1512.

1518. Debugging. Once TeX is working, you should be able to diagnose most errors with the `\show` commands and other diagnostic features. But for the initial stages of debugging, and for the revelation of really deep mysteries, you can compile TeX with a few more aids, including the Pascal runtime checks and its debugger. An additional routine called `debug_help` will also come into play when you type ‘D’ after an error message; `debug_help` also occurs just before a fatal error causes TeX to succumb.

The interface to `debug_help` is primitive, but it is good enough when used with a Pascal debugger that allows you to set breakpoints and to read variables and change their values. After getting the prompt ‘`debug #`’, you type either a negative number (this exits `debug_help`), or zero (this goes to a location where you can set a breakpoint, thereby entering into dialog with the Pascal debugger), or a positive number m followed by an argument n . The meaning of m and n will be clear from the program below. (If $m = 13$, there is an additional argument, l .)

```
define breakpoint = 888 { place where a breakpoint is desirable }

{ Last-minute procedures 1513 } +≡
debug procedure debug_help; { routine to display various things }
label breakpoint, exit;
var k, l, m, n: integer;
begin clear_terminal;
loop
  begin wake_up_terminal; print_nl("debug#(-1 to exit):"); update_terminal; read(term_in, m);
  if m < 0 then return
  else if m = 0 then
    begin goto breakpoint;
    { go to every declared label at least once }
    breakpoint: m ← 0; @{'BREAKPOINT'@}
    end
  else begin read(term_in, n);
  case m of
    { Numbered cases for debug_help 1519 }
    othercases print("?")
    endcases;
    end;
  end;
exit: end;
gubed
```

1519. \langle Numbered cases for *debug_help* 1519 $\rangle \equiv$

```

1: print_word(mem[n]); { display mem[n] in all forms }
2: print_int(info(n));
3: print_int(link(n));
4: print_word(eqtb[n]);
5: print_word(font_info[n]);
6: print_word(save_stack[n]);
7: show_box(n); { show a box, abbreviated by show_box_depth and show_box_breadth }
8: begin breadth_max  $\leftarrow$  10000; depth_threshold  $\leftarrow$  pool_size - pool_ptr - 10; show_node_list(n);
   { show a box in its entirety }
end;
9: show_token_list(n, null, 1000);
10: slow_print(n);
11: check_mem(n > 0); { check wellformedness; print new busy locations if n > 0 }
12: search_mem(n); { look for pointers to n }
13: begin read(term_in, l); print_cmd_chr(n, l);
   end;
14: for k  $\leftarrow$  0 to n do print(buffer[k]);
15: begin font_in_short_display  $\leftarrow$  null_font; short_display(n);
   end;
16: panicking  $\leftarrow$   $\neg$ panicking;
```

This code is used in section 1518.

1520. Extensions. The program above includes a bunch of “hooks” that allow further capabilities to be added without upsetting TeX’s basic structure. Most of these hooks are concerned with “whatsit” nodes, which are intended to be used for special purposes; whenever a new extension to TeX involves a new kind of whatsit node, a corresponding change needs to be made to the routines below that deal with such nodes, but it will usually be unnecessary to make many changes to the other parts of this program.

In order to demonstrate how extensions can be made, we shall treat ‘\write’, ‘\openout’, ‘\closeout’, ‘\immediate’, ‘\special’, and ‘\setlanguage’ as if they were extensions. These commands are actually primitives of TeX, and they should appear in all implementations of the system; but let’s try to imagine that they aren’t. Then the program below illustrates how a person could add them.

Sometimes, of course, an extension will require changes to TeX itself; no system of hooks could be complete enough for all conceivable extensions. The features associated with ‘\write’ are almost all confined to the following paragraphs, but there are small parts of the *print_ln* and *print_char* procedures that were introduced specifically to \write characters. Furthermore one of the token lists recognized by the scanner is a *write_text*; and there are a few other miscellaneous places where we have already provided for some aspect of \write. The goal of a TeX extender should be to minimize alterations to the standard parts of the program, and to avoid them completely if possible. He or she should also be quite sure that there’s no easy way to accomplish the desired goals with the standard features that TeX already has. “Think thrice before extending,” because that may save a lot of work, and it will also keep incompatible extensions of TeX from proliferating.

1521. First let’s consider the format of whatsit nodes that are used to represent the data associated with \write and its relatives. Recall that a whatsit has *type* = *whatsit_node*, and the *subtype* is supposed to distinguish different kinds of whatsits. Each node occupies two or more words; the exact number is immaterial, as long as it is readily determined from the *subtype* or other data.

We shall introduce five *subtype* values here, corresponding to the control sequences \openout, \write, \closeout, \special, and \setlanguage. The second word of I/O whatsits has a *write_stream* field that identifies the write-stream number (0 to 15, or 16 for out-of-range and positive, or 17 for out-of-range and negative). In the case of \write and \special, there is also a field that points to the reference count of a token list that should be sent. In the case of \openout, we need three words and three auxiliary subfields to hold the string numbers for name, area, and extension.

```

define write_node_size = 2 { number of words in a write/whatsit node }
define open_node_size = 3 { number of words in an open/whatsit node }
define open_node = 0 { subtype in whatsits that represent files to \openout }
define write_node = 1 { subtype in whatsits that represent things to \write }
define close_node = 2 { subtype in whatsits that represent streams to \closeout }
define special_node = 3 { subtype in whatsits that represent \special things }
define latespecial_node ≡ 4
    { subtype in whatsits that represent \special things expanded during output }
define language_node = 5 { subtype in whatsits that change the current language }
define what_lang(#) ≡ link(# + 1) { language number, in the range 0 .. 255 }
define what_lhm(#) ≡ type(# + 1) { minimum left fragment, in the range 1 .. 63 }
define what_rhm(#) ≡ subtype(# + 1) { minimum right fragment, in the range 1 .. 63 }
define write_tokens(#) ≡ link(# + 1) { reference count of token list to write }
define write_stream(#) ≡ info(# + 1) { stream number (0 to 17) }
define open_name(#) ≡ link(# + 1) { string number of file name to open }
define open_area(#) ≡ info(# + 2) { string number of file area for open_name }
define open_ext(#) ≡ link(# + 2) { string number of file extension for open_name }

```

1522. The sixteen possible `\write` streams are represented by the `write_file` array. The j th file is open if and only if `write_open[j] = true`. The last two streams are special; `write_open[16]` represents a stream number greater than 15, while `write_open[17]` represents a negative stream number, and both of these variables are always `false`.

```
< Global variables 13 > +≡  
write_file: array [0 .. 15] of alpha_file;  
write_open: array [0 .. 17] of boolean;
```

1523. < Set initial values of key variables 21 > +≡
for $k \leftarrow 0$ to 17 do `write_open[k] ← false`;

1524. Extensions might introduce new command codes; but it's best to use *extension* with a modifier, whenever possible, so that *main_control* stays the same.

```

define immediate_code = 5 { command modifier for \immediate }
define set_language_code = 6 { command modifier for \setlanguage }
define pdftex_first_extension_code = 7
define pdf_literal_node ≡ pdftex_first_extension_code + 0
define pdf_lateliteral_node ≡ pdftex_first_extension_code + 1
define pdf_obj_code ≡ pdftex_first_extension_code + 2
define pdf_refobj_node ≡ pdftex_first_extension_code + 3
define pdf_xform_code ≡ pdftex_first_extension_code + 4
define pdf_refxform_node ≡ pdftex_first_extension_code + 5
define pdf_ximage_code ≡ pdftex_first_extension_code + 6
define pdf_refximage_node ≡ pdftex_first_extension_code + 7
define pdf_annot_node ≡ pdftex_first_extension_code + 8
define pdf_start_link_node ≡ pdftex_first_extension_code + 9
define pdf_end_link_node ≡ pdftex_first_extension_code + 10
define pdf_outline_code ≡ pdftex_first_extension_code + 11
define pdf_dest_node ≡ pdftex_first_extension_code + 12
define pdf_thread_node ≡ pdftex_first_extension_code + 13
define pdf_start_thread_node ≡ pdftex_first_extension_code + 14
define pdf_end_thread_node ≡ pdftex_first_extension_code + 15
define pdf_save_pos_node ≡ pdftex_first_extension_code + 16
define pdf_info_code ≡ pdftex_first_extension_code + 17
define pdf_catalog_code ≡ pdftex_first_extension_code + 18
define pdf_names_code ≡ pdftex_first_extension_code + 19
define pdf_font_attr_code ≡ pdftex_first_extension_code + 20
define pdf_include_chars_code ≡ pdftex_first_extension_code + 21
define pdf_map_file_code ≡ pdftex_first_extension_code + 22
define pdf_map_line_code ≡ pdftex_first_extension_code + 23
define pdf_trailer_code ≡ pdftex_first_extension_code + 24
define pdf_trailer_id_code ≡ pdftex_first_extension_code + 25
define reset_timer_code ≡ pdftex_first_extension_code + 26
define pdf_font_expand_code ≡ pdftex_first_extension_code + 27
define set_random_seed_code ≡ pdftex_first_extension_code + 28
define pdf_snap_ref_point_node ≡ pdftex_first_extension_code + 29
define pdf_snapy_node ≡ pdftex_first_extension_code + 30
define pdf_snapy_comp_node ≡ pdftex_first_extension_code + 31
define pdf_glyph_to_unicode_code ≡ pdftex_first_extension_code + 32
define pdf_colorstack_node ≡ pdftex_first_extension_code + 33
define pdf_setmatrix_node ≡ pdftex_first_extension_code + 34
define pdf_save_node ≡ pdftex_first_extension_code + 35
define pdf_restore_node ≡ pdftex_first_extension_code + 36
define pdf_nobuiltin_tounicode_code ≡ pdftex_first_extension_code + 37
define pdf_interword_space_on_node ≡ pdftex_first_extension_code + 38
define pdf_interword_space_off_node ≡ pdftex_first_extension_code + 39
define pdf_fake_space_node ≡ pdftex_first_extension_code + 40
define pdf_running_link_off_node ≡ pdftex_first_extension_code + 41
define pdf_running_link_on_node ≡ pdftex_first_extension_code + 42
define pdf_space_font_code ≡ pdftex_first_extension_code + 43
define pdftex_last_extension_code ≡ pdftex_first_extension_code + 43

```

{ Put each of TeX's primitives into the hash table 244 } +≡
 primitive("openout", extension, open_node);

```
primitive("write", extension, write_node); write_loc ← cur_val;
primitive("closeout", extension, close_node);
primitive("special", extension, special_node);
primitive("immediate", extension, immediate_code);
primitive("setlanguage", extension, set_language_code);
primitive("pdfliteral", extension, pdf_literal_node);
primitive("pdfcolorstack", extension, pdf_colorstack_node);
primitive("pdfsetmatrix", extension, pdf_setmatrix_node);
primitive("pdfsave", extension, pdf_save_node);
primitive("pdfrestore", extension, pdf_restore_node);
primitive("pdfobj", extension, pdf_obj_code);
primitive("pdfrefobj", extension, pdf_refobj_node);
primitive("pdfxform", extension, pdf_xform_code);
primitive("pdfrefxform", extension, pdf_refxform_node);
primitive("pdfximage", extension, pdf_ximage_code);
primitive("pdfrefiximage", extension, pdf_refiximage_node);
primitive("pdfannot", extension, pdf_annot_node);
primitive("pdfstartlink", extension, pdf_start_link_node);
primitive("pdfendlink", extension, pdf_end_link_node);
primitive("pdfoutline", extension, pdf_outline_code);
primitive("pdfdest", extension, pdf_dest_node);
primitive("pdfthread", extension, pdf_thread_node);
primitive("pdfstartthread", extension, pdf_start_thread_node);
primitive("pdfendthread", extension, pdf_end_thread_node);
primitive("pdfsavepos", extension, pdf_save_pos_node);
primitive("pdfsnaprefpoint", extension, pdf_snap_ref_point_node);
primitive("pdfsnappy", extension, pdf_snappy_node);
primitive("pdfsnappycomp", extension, pdf_snappy_comp_node);
primitive("pdfinfo", extension, pdf_info_code);
primitive("pdfcatalog", extension, pdf_catalog_code);
primitive("pdfnames", extension, pdf_names_code);
primitive("pdfincludechars", extension, pdf_include_chars_code);
primitive("pdffontattr", extension, pdf_font_attr_code);
primitive("pdfmapfile", extension, pdf_map_file_code);
primitive("pdfmapline", extension, pdf_map_line_code);
primitive("pdftrailer", extension, pdf_trailer_code);
primitive("pdftrailerid", extension, pdf_trailer_id_code);
primitive("pdfresettimer", extension, reset_timer_code);
primitive("pdfsetrandomseed", extension, set_random_seed_code);
primitive("pdffontexpand", extension, pdf_font_expand_code);
primitive("pdffglyptounicode", extension, pdf_glyph_to_unicode_code);
primitive("pdfnobuiltintounicode", extension, pdf_no_builtin_tounicode_code);
primitive("pdfinterwordspaceon", extension, pdf_interword_space_on_node);
primitive("pdfinterwordspaceoff", extension, pdf_interword_space_off_node);
primitive("pdffakespace", extension, pdf_fake_space_node);
primitive("pdfrunninglinkoff", extension, pdf_running_link_off_node);
primitive("pdfrunninglinkon", extension, pdf_running_link_on_node);
primitive("pdfspacefont", extension, pdf_space_font_code);
```

1525. The variable *write_loc* just introduced is used to provide an appropriate error message in case of “runaway” write texts.

⟨ Global variables 13 ⟩ +≡
write_loc: pointer; { *eqtb* address of \write }

1526. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 245⟩ +≡
extension: **case** *chr_code* **of**

```

open_node: print_esc("openout");
write_node: print_esc("write");
close_node: print_esc("closeout");
special_node: print_esc("special");
immediate_code: print_esc("immediate");
set_language_code: print_esc("setlanguage");
pdf_annot_node: print_esc("pdfannot");
pdf_catalog_code: print_esc("pdfcatalog");
pdf_dest_node: print_esc("pdfdest");
pdf_end_link_node: print_esc("pdfendlink");
pdf_end_thread_node: print_esc("pdfendthread");
pdf_font_attr_code: print_esc("pdffontattr");
pdf_font_expand_code: print_esc("pdffontexpand");
pdf_include_chars_code: print_esc("pdfincludechars");
pdf_info_code: print_esc("pdfinfo");
pdf_literal_node: print_esc("pdfliteral");
pdf_colorstack_node: print_esc("pdfcolorstack");
pdf_setmatrix_node: print_esc("pdfsetmatrix");
pdf_save_node: print_esc("pdfsave");
pdf_restore_node: print_esc("pdfrestore");
pdf_map_file_code: print_esc("pdfmapfile");
pdf_map_line_code: print_esc("pdfmapline");
pdf_names_code: print_esc("pdfnames");
pdf_obj_code: print_esc("pdfobj");
pdf_outline_code: print_esc("pdfoutline");
pdf_refobj_node: print_esc("pdfrefobj");
pdf_refxform_node: print_esc("pdfrefxform");
pdf_refximage_node: print_esc("pdfrefximage");
pdf_save_pos_node: print_esc("pdfsavepos");
pdf_snap_ref_point_node: print_esc("pdfsnaprefpoint");
pdf_snapy_comp_node: print_esc("pdfsnappycomp");
pdf_snapy_node: print_esc("pdfsnappy");
pdf_start_link_node: print_esc("pdfstartlink");
pdf_start_thread_node: print_esc("pdfstartthread");
pdf_thread_node: print_esc("pdfthread");
pdf_trailer_code: print_esc("pdftrailer");
pdf_trailer_id_code: print_esc("pdftrailerid");
pdf_xform_code: print_esc("pdfxform");
pdf_ximage_code: print_esc("pdfximage");
reset_timer_code: print_esc("pdfresettimer");
set_random_seed_code: print_esc("pdfsetrandomseed");
pdf_nobuiltin_tounicode_code: print_esc("pdfnobuiltintounicode");
pdf_glyph_to_unicode_code: print_esc("pdfglyphtounicode");
pdf_interword_space_on_node: print_esc("pdfinterwordspaceon");
pdf_interword_space_off_node: print_esc("pdfinterwordspaceoff");
pdf_fake_space_node: print_esc("pdffakespace");
pdf_running_link_off_node: print_esc("pdfrunninglinkoff");
pdf_running_link_on_node: print_esc("pdfrunninglinkon");
pdf_space_font_code: print_esc("pdfspacefont");
othercases print(" [unknown_\extension!] ")

```

endcases;

1527. When an *extension* command occurs in *main_control*, in any mode, the *do_extension* routine is called.

(Cases of main_control that are for extensions to T_EX 1527) ≡

any_mode(extension): do_extension;

This code is used in section 1223.

1528. ⟨ Declare action procedures for use by *main_control* 1221 ⟩ +≡
 ⟨ Declare procedures needed in *do_extension* 1529 ⟩

```

procedure do_extension;
  var i,j,k: integer; { all-purpose integers }
  p,q,r: pointer; { all-purpose pointers }
begin case cur_chr of
  open_node: < Implement \openout 1531 >;
  write_node: < Implement \write 1532 >;
  close_node: < Implement \closeout 1533 >;
  special_node: < Implement \special 1534 >;
  immediate_code: < Implement \immediate 1623 >;
  set_language_code: < Implement \setlanguage 1625 >;
  pdf_annot_node: < Implement \pdfannot 1558 >;
  pdf_catalog_code: < Implement \pdfcatalog 1579 >;
  pdf_dest_node: < Implement \pdfdest 1565 >;
  pdf_end_link_node: < Implement \pdfendlink 1561 >;
  pdf_end_thread_node: < Implement \pdfendthread 1569 >;
  pdf_font_attr_code: < Implement \pdffontattr 1589 >;
  pdf_font_expand_code: < Implement \pdffontexpand 1535 >;
  pdf_include_chars_code: < Implement \pdfincludechars 1588 >;
  pdf_info_code: < Implement \pdfinfo 1578 >;
  pdf_literal_node: < Implement \pdfliteral 1538 >;
  pdf_colorstack_node: < Implement \pdfcolorstack 1539 >;
  pdf_setmatrix_node: < Implement \pdfsetmatrix 1540 >;
  pdf_save_node: < Implement \pdfsave 1541 >;
  pdf_restore_node: < Implement \pdfrestore 1542 >;
  pdf_map_file_code: < Implement \pdfmapfile 1590 >;
  pdf_map_line_code: < Implement \pdfmapline 1591 >;
  pdf_names_code: < Implement \pdfnames 1580 >;
  pdf_obj_code: < Implement \pdfobj 1544 >;
  pdf_outline_code: < Implement \pdfoutline 1563 >;
  pdf_refobj_node: < Implement \pdfrefobj 1546 >;
  pdf_refxform_node: < Implement \pdfrefxform 1549 >;
  pdf_refximage_node: < Implement \pdfrefximage 1554 >;
  pdf_save_pos_node: < Implement \pdfsavepos 1576 >;
  pdf_snap_ref_point_node: < Implement \pdfsnaprefpoint 1572 >;
  pdf_snapy_comp_node: < Implement \pdfsnappycomp 1575 >;
  pdf_snapy_node: < Implement \pdfsnappy 1574 >;
  pdf_start_link_node: < Implement \pdfstartlink 1560 >;
  pdf_start_thread_node: < Implement \pdfstartthread 1568 >;
  pdf_thread_node: < Implement \pdfthread 1567 >;
  pdf_trailer_code: < Implement \pdftrailer 1581 >;
  pdf_trailer_id_code: < Implement \pdftrailerid 1582 >;
  pdf_xform_code: < Implement \pdfxform 1548 >;
  pdf_ximage_code: < Implement \pdfximage 1553 >;
  reset_timer_code: < Implement \pdfresettimer 1586 >;
  set_random_seed_code: < Implement \pdfsetrandomseed 1585 >;
  pdf_glyph_to_unicode_code: < Implement \pdfglyptounicode 1592 >;
  pdf_no builtin_tounicode_code: < Implement \pdfnobuiltintounicode 1593 >;
  pdf_interword_space_on_node: < Implement \pdfinterwordspaceon 1594 >;
  pdf_interword_space_off_node: < Implement \pdfinterwordspaceoff 1595 >;
  pdf_fake_space_node: < Implement \pdffake space 1596 >;

```

```

pdf_running_link_off_node: ⟨ Implement \pdfrunninglinkoff 1597 ⟩;
pdf_running_link_on_node: ⟨ Implement \pdfrunninglinkon 1598 ⟩;
pdf_space_font_code: ⟨ Implement \pdfspacefont 1599 ⟩;
othercases confusion("ext1")
endcases;
end;

```

1529. Here is a subroutine that creates a whatsit node having a given *subtype* and a given number of words. It initializes only the first word of the whatsit, and appends it to the current list.

```

⟨ Declare procedures needed in do_extension 1529 ⟩ ≡
procedure new_whatsit(s : small_number; w : small_number);
  var p: pointer; { the new node }
  begin p ← get_node(w); type(p) ← whatsit_node; subtype(p) ← s; link(tail) ← p; tail ← p;
  end;

```

See also sections 1530, 1537, 1552, 1556, 1562, 1566, 1573, 1577, 1587, and 1600.

This code is used in section 1528.

1530. The next subroutine uses *cur_chr* to decide what sort of whatsit is involved, and also inserts a *write_stream* number.

```

⟨ Declare procedures needed in do_extension 1529 ⟩ +≡
procedure new_write_whatsit(w : small_number);
  begin new_whatsit(cur_chr,w);
  if w ≠ write_node_size then scan_four_bit_int
  else begin scan_int;
    if cur_val < 0 then cur_val ← 17
    else if cur_val > 15 then cur_val ← 16;
    end;
  write_stream(tail) ← cur_val;
  end;

```

1531. ⟨ Implement \openout 1531 ⟩ ≡

```

begin new_write_whatsit(open_node_size); scan_optional_equals; scan_file_name;
open_name(tail) ← cur_name; open_area(tail) ← cur_area; open_ext(tail) ← cur_ext;
end

```

This code is used in section 1528.

1532. When ‘\write 12{...}’ appears, we scan the token list ‘{...}’ without expanding its macros; the macros will be expanded later when this token list is rescanned.

```

⟨ Implement \write 1532 ⟩ ≡
begin k ← cur_cs; new_write_whatsit(write_node_size);
cur_cs ← k; p ← scan_toks(false, false); write_tokens(tail) ← def_ref;
end

```

This code is used in section 1528.

1533. ⟨ Implement \closeout 1533 ⟩ ≡

```

begin new_write_whatsit(write_node_size); write_tokens(tail) ← null;
end

```

This code is used in section 1528.

1534. When ‘\special{...}’ appears, we expand the macros in the token list as in \xdef and \mark. When marked with shipout, we keep tokens unexpanded for now.

Unfortunately, the *write_stream(tail) ← null* done here is not a valid assignment in Web2C, because *null* (a.k.a. *min_halfword*) is a large negative number ($-268435455 = -\text{FFFFFFFFFF}$, set in *tex.ch*); too large to fit in the *short* structure element that’s being assigned. The warning from gcc 8.5.0 was:

pdftex0.c: In function ‘doextension’:

```
pdftex0.c:37849:40: warning: overflow in conversion from 'long int' to 'short int'
changes value from '-268435455' to '1' [-Woverflow]
```

```
mem [curlist .tailfield + 1 ].hh.b0 = -268435455L ;
```

The correct thing to do is not immediately evident. However, for Web2C, it does not matter, because these lines are changed for encTeX, in *encTeX2.ch*, and now zero is assigned, instead of *null*.

⟨ Implement \special 1534 ⟩ ≡

```
begin if scan_keyword("shipout") then
  begin new_whatsit(latespecial_node, write_node_size); write_stream(tail) ← null;
  p ← scan_toks(false, false); write_tokens(tail) ← def_ref;
  end
else begin new_whatsit(special_node, write_node_size); write_stream(tail) ← null;
  p ← scan_toks(false, true); write_tokens(tail) ← def_ref;
  end;
end
```

This code is used in section 1528.

1535. ⟨ Implement \pdffontexpand 1535 ⟩ ≡

```
read_expand_font
```

This code is used in section 1528.

1536. The following macros are needed for further manipulation with whatsit nodes for pdfTeX extensions (copying, destroying, etc.).

```
define add_action_ref(#) ≡ incr(pdf_action_refcount(#)) { increase count of references to this action }
define delete_action_ref(#) ≡
  { decrease count of references to this action; free it if there is no reference to this action }
begin if pdf_action_refcount(#) = null then
  begin if pdf_action_type(#) = pdf_action_user then delete_token_ref(pdf_action_user_tokens(#))
  else begin if pdf_action_file(#) ≠ null then delete_token_ref(pdf_action_file(#));
    if pdf_action_type(#) = pdf_action_page then delete_token_ref(pdf_action_page_tokens(#))
    else if (pdf_action_named_id(#) ∧ 1) = 1 then delete_token_ref(pdf_action_id(#));
    if (pdf_action_named_id(#) ∧ 2) = 2 then delete_token_ref(pdf_action_struct_id(#));
    end;
  free_node(#, pdf_action_size);
  end
else decr(pdf_action_refcount(#));
end
```

1537. We have to check whether `\pdfoutput` is set for using pdfTeX extensions.

```
< Declare procedures needed in do_extension 1529 > +≡
procedure check_pdfoutput(s : str_number; is_error : boolean);
begin if pdf_output ≤ 0 then
  begin if is_error then pdf_error(s, "not allowed in DVI mode (\pdfoutput≤0)")
  else pdf_warning(s, "not allowed in DVI mode (\pdfoutput≤0); ignoring it", true, true);
  end
end;
procedure scan_pdf_ext_toks;
begin call_func(scan_toks(false, true)); { like \special }
end;
procedure scan_pdf_ext_late_toks;
begin call_func(scan_toks(false, false)); { like \special, but doesn't expand }
end;
procedure compare_strings; { to implement \pdfstrcmp }
label done;
var s1, s2: str_number; i1, i2, j1, j2: pool_pointer; save_cur_cs: pointer;
begin save_cur_cs ← cur_cs; call_func(scan_toks(false, true)); s1 ← tokens_to_string(def_ref);
delete_token_ref(def_ref); cur_cs ← save_cur_cs; call_func(scan_toks(false, true));
s2 ← tokens_to_string(def_ref); delete_token_ref(def_ref); i1 ← str_start[s1]; j1 ← str_start[s1 + 1];
i2 ← str_start[s2]; j2 ← str_start[s2 + 1];
while (i1 < j1) ∧ (i2 < j2) do
  begin if str_pool[i1] < str_pool[i2] then
    begin cur_val ← -1; goto done;
    end;
  if str_pool[i1] > str_pool[i2] then
    begin cur_val ← 1; goto done;
    end;
  incr(i1); incr(i2);
  end;
if (i1 = j1) ∧ (i2 = j2) then cur_val ← 0
else if i1 < j1 then cur_val ← 1
  else cur_val ← -1;
done: flush_str(s2); flush_str(s1); cur_val_level ← int_val;
end;
```

1538. { Implement `\pdfliteral` 1538 } ≡

```
begin check_pdfoutput("\pdfliteral", true);
if scan_keyword("shipout") then k ← pdf_lateliteral_node
else k ← pdf_literal_node;
new_whatsit(k, write_node_size);
if scan_keyword("direct") then pdf_literal_mode(tail) ← direct_always
else if scan_keyword("page") then pdf_literal_mode(tail) ← direct_page
  else pdf_literal_mode(tail) ← set_origin;
if k = pdf_literal_node then scan_pdf_ext_toks
  else scan_pdf_ext_late_toks;
pdf_literal_data(tail) ← def_ref;
end
```

This code is used in section 1528.

1539. $\langle \text{Implement } \backslash\text{pdfcolorstack } 1539 \rangle \equiv$

```

begin check_pdfoutput("\pdfcolorstack", true);
  { Scan and check the stack number and store in cur_val }
  scan_int;
  if cur_val ≥ colorstackused then
    begin print_err("Unknown_color_stack_number");
      help3("Allocate_and_initialize_a_color_stack_with_\\pdfcolorstackinit.")
      ("I'll_use_default_color_stack_0_here.")
      ("Proceed, with_fingers_crossed."); error; cur_val ← 0;
    end;
  if cur_val < 0 then
    begin print_err("Invalid_negative_color_stack_number");
      help2("I'll_use_default_color_stack_0_here.")
      ("Proceed, with_fingers_crossed."); error; cur_val ← 0;
    end; { Scan the command and store in i, j holds the node size }
  if scan_keyword("set") then
    begin i ← colorstack_set; j ← pdf_colorstack_setter_node_size;
  end
  else if scan_keyword("push") then
    begin i ← colorstack_push; j ← pdf_colorstack_setter_node_size;
  end
  else if scan_keyword("pop") then
    begin i ← colorstack_pop; j ← pdf_colorstack_getter_node_size;
  end
  else if scan_keyword("current") then
    begin i ← colorstack_current; j ← pdf_colorstack_getter_node_size;
  end
  else begin i ← -1; { error }
  end;
  if i ≥ 0 then
    begin new_whatsit(pdf_colorstack_node, j); pdf_colorstack_stack(tail) ← cur_val;
    pdf_colorstack_cmd(tail) ← i;
    if i ≤ colorstack_data then
      begin scan_pdf_ext_toks; pdf_colorstack_data(tail) ← def_ref;
    end;
  end
  else begin print_err("Color_stack_action_is_missing");
    help3("The_expected_actions_for_\\pdfcolorstack:")
    ("set, push, pop, current")
    ("I'll_ignore_the_color_stack_command."); error;
  end
end

```

This code is used in section 1528.

1540. $\langle \text{Implement } \backslash\text{pdfsetmatrix } 1540 \rangle \equiv$

```

begin check_pdfoutput("\pdfsetmatrix", true);
new_whatsit(pdf_setmatrix_node, pdf_setmatrix_node_size); scan_pdf_ext_toks;
pdf_setmatrix_data(tail) ← def_ref;
end

```

This code is used in section 1528.

1541. $\langle \text{Implement } \text{\textbackslash pdfsave 1541} \rangle \equiv$
begin *check_pdfoutput*("**\pdfsave**", *true*); *new_whatst*(*pdf_save_node*, *pdf_save_node_size*);
end

This code is used in section 1528.

1542. $\langle \text{Implement } \text{\textbackslash pdfrestore 1542} \rangle \equiv$
begin *check_pdfoutput*("**\pdfrestore**", *true*); *new_whatst*(*pdf_restore_node*, *pdf_restore_node_size*);
end

This code is used in section 1528.

1543. The **\pdfobj** primitive is used to create a “raw” object in the PDF output file. The object contents will be hold in memory and will be written out only when the object is referenced by **\pdfrefobj**. When **\pdfobj** is used with **\immediate**, the object contents will be written out immediately. Objects referenced in the current page are appended into *pdf_obj_list*.

$\langle \text{Global variables 13} \rangle +\equiv$
pdf_last_obj: *integer*;

1544. $\langle \text{Implement } \text{\textbackslash pdfobj 1544} \rangle \equiv$
begin *check_pdfoutput*("**\pdfobj**", *true*);
if *scan_keyword*("**reserveobjnum**") **then**
begin (Scan an optional space 469);
incr(*pdf_obj_count*); *pdf_create_obj*(*obj_type_obj*, *pdf_obj_count*); *pdf_last_obj* \leftarrow *obj_ptr*;
end
else begin *k* \leftarrow -1;
if *scan_keyword*("**useobjnum**") **then**
begin *scan_int*; *k* \leftarrow *cur_val*;
if (*k* \leq 0) \vee (*k* $>$ *obj_ptr*) \vee (*obj_data_ptr*(*k*) \neq 0) **then**
begin *pdf_warning*("**\pdfobj**", "invalid_object_number_being_ignored", *true*, *true*);
pdf_retnal \leftarrow -1; { signal the problem }
k \leftarrow -1; { will be generated again }
end;
end;
if *k* < 0 **then**
begin *incr*(*pdf_obj_count*); *pdf_create_obj*(*obj_type_obj*, *pdf_obj_count*); *k* \leftarrow *obj_ptr*;
end;
obj_data_ptr(*k*) \leftarrow *pdf_get_mem*(*pdffmem_obj_size*);
if *scan_keyword*("**stream**") **then**
begin *obj_obj_is_stream*(*k*) \leftarrow 1;
if *scan_keyword*("**attr**") **then**
begin *scan_pdf_ext_toks*; *obj_obj_stream_attr*(*k*) \leftarrow *def_ref*;
end
else *obj_obj_stream_attr*(*k*) \leftarrow null;
end
else *obj_obj_is_stream*(*k*) \leftarrow 0;
if *scan_keyword*("**file**") **then** *obj_obj_is_file*(*k*) \leftarrow 1
else *obj_obj_is_file*(*k*) \leftarrow 0;
scan_pdf_ext_toks; *obj_obj_data*(*k*) \leftarrow *def_ref*; *pdf_last_obj* \leftarrow *k*;
end;
end

This code is used in section 1528.

1545. We need to check whether the referenced object exists.

```
( Declare procedures that need to be declared forward for pdfTeX 686 ) +≡
function prev_rightmost(s, e : pointer): pointer;
    { finds the node preceding the rightmost node e; s is some node before e }
var p: pointer;
begin prev_rightmost ← null; p ← s;
if p = null then return;
while link(p) ≠ e do
begin p ← link(p);
if p = null then return;
end;
prev_rightmost ← p;
end;

procedure pdf_check_obj(t, n : integer);
var k: integer;
begin k ← head_tab[t];
while (k ≠ 0) ∧ (k ≠ n) do k ← obj_link(k);
if k = 0 then pdf_error("ext1", "cannot_find_referenced_object");
end;
```

1546. ⟨ Implement \pdfrefobj 1546 ⟩ ≡

```
begin check_pdfoutput("\pdfrefobj", true); scan_int; pdf_check_obj(obj_type_obj, cur_val);
new_whatsit(pdf_refobj_node, pdf_refobj_node_size); pdf_obj_objnum(tail) ← cur_val;
end
```

This code is used in section 1528.

1547. \pdfxform and \pdfrefxform are similar to \pdfobj and \pdfrefobj.

```
( Global variables 13 ) +≡
pdf_last_xform: integer;
```

1548. ⟨ Implement \pdfxform 1548 ⟩ ≡

```
begin check_pdfoutput("\pdfxform", true); incr(pdf_xform_count);
pdf_create_obj(obj_type_xform, pdf_xform_count); k ← obj_ptr;
obj_data_ptr(k) ← pdf_get_mem(pdfmem_xform_size);
if scan_keyword("attr") then
begin scan_pdf_ext_toks; obj_xform_attr(k) ← def_ref;
end
else obj_xform_attr(k) ← null;
if scan_keyword("resources") then
begin scan_pdf_ext_toks; obj_xform_resources(k) ← def_ref;
end
else obj_xform_resources(k) ← null;
scan_register_num; fetch_box(p);
if p = null then pdf_error("ext1", "\pdfxform cannot be used with a void box");
obj_xform_width(k) ← width(p); obj_xform_height(k) ← height(p); obj_xform_depth(k) ← depth(p);
obj_xform_box(k) ← p; { save pointer to the box }
change_box(null); pdf_last_xform ← k;
end
```

This code is used in section 1528.

1549. $\langle \text{Implement } \text{\textbackslash pdfrefxform 1549} \rangle \equiv$
`begin check_pdfoutput("pdfrefxform", true); scan_int; pdf_check_obj(obj_type_xform, cur_val);
new_whatsit(pdf_refxform_node, pdf_refxform_node_size); pdf_xform_objnum(tail) ← cur_val;
pdf_width(tail) ← obj_xform_width(cur_val); pdf_height(tail) ← obj_xform_height(cur_val);
pdf_depth(tail) ← obj_xform_depth(cur_val);
end`

This code is used in section 1528.

1550. $\text{\textbackslash pdfximage}$ and $\text{\textbackslash pdfrefximage}$ are similar to $\text{\textbackslash pdfxform}$ and $\text{\textbackslash pdfrefxform}$. As we have to scan $\langle \text{rule spec} \rangle$ quite often, it is better have a rule_node that holds the most recently scanned $\langle \text{rule spec} \rangle$.

$\langle \text{Global variables 13} \rangle +\equiv$
`pdf_last_ximage: integer;
pdf_last_ximage_pages: integer;
pdf_last_ximage_colorddepth: integer;
alt_rule: pointer;
warn_pdfpagebox: boolean;`

1551. $\langle \text{Set initial values of key variables 21} \rangle +\equiv$
`alt_rule ← null; warn_pdfpagebox ← true;`

1552. ⟨ Declare procedures needed in *do_extension* 1529 ⟩ +≡

```

procedure scale_image(n : integer);
  var x, y, xr, yr: integer; { size and resolution of image }
    w, h: scaled; { indeed size corresponds to image resolution }
    default_res: integer; image: integer;
  begin image ← obj_ximage_data(n);
  if (image_rotate(image) = 90) ∨ (image_rotate(image) = 270) then
    begin y ← image_width(image); x ← image_height(image); yr ← image_x_res(image);
    xr ← image_y_res(image);
    end
  else begin x ← image_width(image); y ← image_height(image); xr ← image_x_res(image);
    yr ← image_y_res(image);
    end;
  if (xr > 65535) ∨ (yr > 65535) then
    begin xr ← 0; yr ← 0; pdf_warning("ext1", "too_large_image_resolution_ignored", true, true);
    end;
  if (x ≤ 0) ∨ (y ≤ 0) ∨ (xr < 0) ∨ (yr < 0) then pdf_error("ext1", "invalid_image_dimensions");
  if is_pdf_image(image) then
    begin w ← x; h ← y;
    end
  else begin default_res ← fix_int(pdf_image_resolution, 0, 65535);
    if (default_res > 0) ∧ ((xr = 0) ∨ (yr = 0)) then
      begin xr ← default_res; yr ← default_res;
      end;
    if is_running(obj_ximage_width(n)) ∧ is_running(obj_ximage_height(n)) then
      begin if (xr > 0) ∧ (yr > 0) then
        begin w ← ext_xn_over_d(one_hundred_inch, x, 100 * xr);
        h ← ext_xn_over_d(one_hundred_inch, y, 100 * yr);
        end
      else begin w ← ext_xn_over_d(one_hundred_inch, x, 7200);
        h ← ext_xn_over_d(one_hundred_inch, y, 7200);
        end;
      end;
    end;
  end;
  if is_running(obj_ximage_width(n)) ∧ is_running(obj_ximage_height(n)) ∧ is_running(obj_ximage_depth(n))
    then
  begin obj_ximage_width(n) ← w; obj_ximage_height(n) ← h; obj_ximage_depth(n) ← 0;
  end
  else if is_running(obj_ximage_width(n)) then
    begin { image depth or height is explicitly specified }
    if is_running(obj_ximage_height(n)) then
      begin { image depth is explicitly specified }
      obj_ximage_width(n) ← ext_xn_over_d(h, x, y); obj_ximage_height(n) ← h - obj_ximage_depth(n);
      end
    else if is_running(obj_ximage_depth(n)) then
      begin { image height is explicitly specified }
      obj_ximage_width(n) ← ext_xn_over_d(obj_ximage_height(n), x, y); obj_ximage_depth(n) ← 0;
      end
    else begin { both image depth and height are explicitly specified }
      obj_ximage_width(n) ← ext_xn_over_d(obj_ximage_height(n) + obj_ximage_depth(n), x, y);
      end;
    end;
  end
end

```

```

else begin { image width is explicitly specified }
  if is_running(obj_ximage_height(n)) ∧ is_running(obj_ximage_depth(n)) then
    begin { both image depth and height are not specified }
      obj_ximage_height(n) ← ext_xn_over_d(obj_ximage_width(n), y, x); obj_ximage_depth(n) ← 0;
    end { image depth is explicitly specified }
  else if is_running(obj_ximage_height(n)) then
    begin obj_ximage_height(n) ← ext_xn_over_d(obj_ximage_width(n), y, x) – obj_ximage_depth(n);
    end { image height is explicitly specified }
  else if is_running(obj_ximage_depth(n)) then
    begin obj_ximage_depth(n) ← 0;
    end { both image depth and height are explicitly specified }
  else do_nothing;
end;
end;
function scan_pdf_box_spec: integer; { scans PDF pagebox specification }
begin scan_pdf_box_spec ← 0;
if scan_keyword("mediabox") then scan_pdf_box_spec ← pdf_box_spec_media
else if scan_keyword("cropbox") then scan_pdf_box_spec ← pdf_box_spec_crop
else if scan_keyword("bleedbox") then scan_pdf_box_spec ← pdf_box_spec_bleed
else if scan_keyword("trimbox") then scan_pdf_box_spec ← pdf_box_spec_trim
else if scan_keyword("artbox") then scan_pdf_box_spec ← pdf_box_spec_art
end;
procedure scan_alt_rule; { scans rule spec to alt_rule }
label reswitch;
begin if alt_rule = null then alt_rule ← new_rule;
width(alt_rule) ← null_flag; height(alt_rule) ← null_flag; depth(alt_rule) ← null_flag;
reswitch: if scan_keyword("width") then
  begin scan_normal_dimen; width(alt_rule) ← cur_val; goto reswitch;
end;
if scan_keyword("height") then
  begin scan_normal_dimen; height(alt_rule) ← cur_val; goto reswitch;
end;
if scan_keyword("depth") then
  begin scan_normal_dimen; depth(alt_rule) ← cur_val; goto reswitch;
end;
end;
procedure scan_image;
label reswitch;
var k: integer; named: str_number; s: str_number; page, pagebox, colorspace: integer;
begin incr(pdf_ximage_count); pdf_create_obj(obj_type_ximage, pdf_ximage_count); k ← obj_ptr;
obj_data_ptr(k) ← pdf_get_mem(pdffmem_ximage_size); scan_alt_rule; { scans <rule spec> to alt_rule }
obj_ximage_width(k) ← width(alt_rule); obj_ximage_height(k) ← height(alt_rule);
obj_ximage_depth(k) ← depth(alt_rule);
if scan_keyword("attr") then
  begin scan_pdf_ext_toks; obj_ximage_attr(k) ← def_ref;
end
else obj_ximage_attr(k) ← null;
named ← 0;
if scan_keyword("named") then
  begin scan_pdf_ext_toks; named ← tokens_to_string(def_ref); delete_token_ref(def_ref);
end
else if scan_keyword("page") then

```

```

begin scan_int; page ← cur_val;
end
else page ← 1;
if scan_keyword("colorspace") then
begin scan_int; colorspace ← cur_val;
end
else colorspace ← 0;
pagebox ← scan_pdf_box_spec;
if pagebox = 0 then pagebox ← pdf_pagebox;
scan_pdf_ext_toks; s ← tokens_to_string(def_ref); delete_token_ref(def_ref);
if pdf_option_always_use_pdfpagebox ≠ 0 then
begin pdf_warning("PDF_inclusion",
    "Primitive\pdfoptionalwaysusepdfpagebox_isObsolete;\use\pdfpagebox instead.",
    true, true); pdf_force_pagebox ← pdf_option_always_use_pdfpagebox;
pdf_option_always_use_pdfpagebox ← 0; { warn once }
warn_pdfpagebox ← false;
end;
if pdf_option_pdf_inclusion_errorlevel ≠ 0 then
begin pdf_warning("PDF_inclusion",
    "Primitive\pdfoptionpdfinclusionerrorlevel_isObsolete;\use\pdfinclusionerrorlevel instead",
    true, true); pdf_inclusion_errorlevel ← pdf_option_pdf_inclusion_errorlevel;
pdf_option_pdf_inclusion_errorlevel ← 0; { warn once }
end;
if pdf_force_pagebox > 0 then
begin if warn_pdfpagebox then
begin pdf_warning("PDF_inclusion",
    "Primitive\pdfforcepagebox_isObsolete;\use\pdfpagebox instead.", true, true);
warn_pdfpagebox ← false;
end;
pagebox ← pdf_force_pagebox;
end;
if pagebox = 0 then { no pagebox specification given }
pagebox ← pdf_box_spec_crop;
obj_ximage_data(k) ← read_image(s, page, named, colorspace, pagebox, pdf_major_version,
pdf_minor_version, pdf_inclusion_errorlevel);
if named ≠ 0 then flush_str(named);
flush_str(s); scale_image(k); pdf_last_ximage ← k;
pdf_last_ximage_pages ← image_pages(obj_ximage_data(k));
pdf_last_ximage_colordepth ← image_colordepth(obj_ximage_data(k));
end;

```

1553. ⟨Implement \pdffximage 1553⟩ ≡

```

begin check_pdfoutput("\pdffximage", true); check_pdfversion; scan_image;
end

```

This code is used in section 1528.

1554. *⟨ Implement \pdfrefximage 1554 ⟩* ≡

```

begin check_pdfoutput("pdfrefximage", true); scan_int; pdf_check_obj(obj_type_ximage, cur_val);
new_whatis(pdf_refximage_node, pdf_refximage_node_size); pdf_ximage_objnum(tail) ← cur_val;
pdf_width(tail) ← obj_ximage_width(cur_val); pdf_height(tail) ← obj_ximage_height(cur_val);
pdf_depth(tail) ← obj_ximage_depth(cur_val);
end

```

This code is used in section 1528.

1555. The following function finds object with identifier i and type t . $i < 0$ indicates that $-i$ should be treated as a string number. If no such object exists then it will be created. This function is used mainly to find destination for link annotations and outlines; however it is also used in *pdf_ship_out* (to check whether a Page object already exists) so we need to declare it together with subroutines needed in *pdf_hlist_out* and *pdf_vlist_out*.

```

⟨ Declare procedures that need to be declared forward for pdfTeX 686 ⟩ +≡
function find_obj(t, i : integer; byname : boolean): integer;
begin find_obj ← avl_find_obj(t, i, byname);
end;
procedure flush_str(s : str_number); { flush a string if possible }
begin if flushable(s) then flush_string;
end;
function get_obj(t, i : integer; byname : boolean): integer;
var r: integer; s: str_number;
begin if byname > 0 then
begin s ← tokens_to_string(i); r ← find_obj(t, s, true);
end
else begin s ← 0; r ← find_obj(t, i, false);
end;
if r = 0 then
begin if byname > 0 then
begin pdf_create_obj(t, -s); s ← 0;
end
else pdf_create_obj(t, i);
r ← obj_ptr;
if (t = obj_type_dest) ∨ (t = obj_type_struct_dest) then obj_dest_ptr(r) ← null;
end;
if s ≠ 0 then flush_str(s);
get_obj ← r;
end;
function get_microinterval: integer;
var s, m: integer; { seconds and microseconds }
begin seconds_and_micros(s, m);
if (s - epochseconds) > 32767 then get_microinterval ← max_integer
else if (microseconds > m) then
get_microinterval ← ((s-1-epochseconds)*65536)+(((m+1000000-microseconds)/100)*65536)/10000
else get_microinterval ← ((s - epochseconds) * 65536) + (((m - microseconds)/100) * 65536)/10000;
end;

```

1556. ⟨ Declare procedures needed in *do_extension* 1529 ⟩ +≡

```

function scan_action: pointer; { read an action specification }

var p: integer;
begin p ← get_node(pdf_action_size); scan_action ← p; pdf_action_file(p) ← null;
pdf_action_refcount(p) ← null;
if scan_keyword("user") then pdf_action_type(p) ← pdf_action_user
else if scan_keyword("goto") then pdf_action_type(p) ← pdf_action_goto
else if scan_keyword("thread") then pdf_action_type(p) ← pdf_action_thread
else pdf_error("ext1", "action_type_missing");
if pdf_action_type(p) = pdf_action_user then
begin scan_pdf_ext_toks; pdf_action_user_tokens(p) ← def_ref; return;
end;
pdf_action_named_id(p) ← 0;
if scan_keyword("file") then
begin scan_pdf_ext_toks; pdf_action_file(p) ← def_ref;
end;
if scan_keyword("struct") then
begin if pdf_action_type(p) ≠ pdf_action_goto then
pdf_error("ext1", "only_GoTo_action_can_be_used_with_struct");
if pdf_action_file(p) ≠ null then
begin scan_pdf_ext_toks; pdf_action_named_id(p) ← pdf_action_named_id(p) + 2;
pdf_action_struct_id(p) ← def_ref;
end
else if scan_keyword("name") then
begin scan_pdf_ext_toks; pdf_action_named_id(p) ← pdf_action_named_id(p) + 2;
pdf_action_struct_id(p) ← def_ref;
end
else if scan_keyword("num") then
begin scan_int;
if cur_val ≤ 0 then pdf_error("ext1", "num_identifier_must_be_positive");
pdf_action_struct_id(p) ← cur_val;
end
else pdf_error("ext1", "identifier_type_missing");
end
else pdf_action_struct_id(p) ← null;
if scan_keyword("page") then
begin if pdf_action_type(p) ≠ pdf_action_goto then
pdf_error("ext1", "only_GoTo_action_can_be_used_with_page");
pdf_action_type(p) ← pdf_action_page; scan_int;
if cur_val ≤ 0 then pdf_error("ext1", "page_number_must_be_positive");
pdf_action_id(p) ← cur_val; scan_pdf_ext_toks; pdf_action_page_tokens(p) ← def_ref;
end
else if scan_keyword("name") then
begin scan_pdf_ext_toks; pdf_action_named_id(p) ← pdf_action_named_id(p) + 1;
pdf_action_id(p) ← def_ref;
end
else if scan_keyword("num") then
begin if (pdf_action_type(p) = pdf_action_goto) ∧ (pdf_action_file(p) ≠ null) then
pdf_error("ext1", "goto_option_cannot_be_used_with_both_file_and_num");
scan_int;
if cur_val ≤ 0 then pdf_error("ext1", "num_identifier_must_be_positive");
pdf_action_id(p) ← cur_val;

```

```

    end
    else pdf_error("ext1", "identifier_type_missing");
if scan_keyword("newwindow") then
  begin pdf_action_new_window(p) ← 1; {Scan an optional space 469};
  end
else if scan_keyword("nonewwindow") then
  begin pdf_action_new_window(p) ← 2; {Scan an optional space 469};
  end
else pdf_action_new_window(p) ← 0;
if (pdf_action_new_window(p) > 0) ∧ (((pdf_action_type(p) ≠ pdf_action_goto) ∧ (pdf_action_type(p) ≠
  pdf_action_page)) ∨ (pdf_action_file(p) = null)) then
  pdf_error("ext1", "`newwindow`/`nonewwindow` must be used with `goto` and `file` option");
end;
procedure new_annot_whatsit(w, s : small_number); {create a new whatsit node for annotation}
begin new_whatsit(w, s); scan_alt_rule; {scans <rule spec> to alt_rule}
pdf_width(tail) ← width(alt_rule); pdf_height(tail) ← height(alt_rule); pdf_depth(tail) ← depth(alt_rule);
if (w = pdf_start_link_node) then
  begin if scan_keyword("attr") then
    begin scan_pdf_ext_toks; pdf_link_attr(tail) ← def_ref;
    end
  else pdf_link_attr(tail) ← null;
  end;
if (w = pdf_thread_node) ∨ (w = pdf_start_thread_node) then
  begin if scan_keyword("attr") then
    begin scan_pdf_ext_toks; pdf_thread_attr(tail) ← def_ref;
    end
  else pdf_thread_attr(tail) ← null;
  end;
end;

```

1557. {Global variables 13} +≡

pdf_last_annot: integer;

1558. {Implement \pdfannot 1558} ≡

```

begin check_pdfoutput("\pdfannot", true);
if scan_keyword("reserveobjnum") then
  begin pdf_last_annot ← pdf_new_objnum; {Scan an optional space 469};
  end
else begin if scan_keyword("useobjnum") then
  begin scan_int; k ← cur_val;
  if (k ≤ 0) ∨ (k > obj_ptr) ∨ (obj_annotation_ptr(k) ≠ 0) then
    pdf_error("ext1", "invalid object number");
  end
  else k ← pdf_new_objnum;
  new_annot_whatsit(pdf_annotation_node, pdf_annotation_node_size); pdf_annotation_objnum(tail) ← k;
  scan_pdf_ext_toks; pdf_annotation_data(tail) ← def_ref; pdf_last_annot ← k;
  end
end

```

This code is used in section 1528.

1559. `\pdflastlink` needs an extra global variable.

```
< Global variables 13 > +≡
pdf_last_link: integer;
```

1560. `< Implement \pdfstartlink 1560 >` ≡

```
begin check_pdfoutput("\pdfstartlink", true);
if abs(mode) = vmode then pdf_error("ext1", "\pdfstartlink cannot be used in vertical mode");
k ← pdf_new_objnum; new_annot_whatsit(pdf_start_link_node, pdf_annot_node_size);
pdf_link_action(tail) ← scan_action; pdf_link_objnum(tail) ← k; pdf_last_link ← k; { N.B.: although it is
possible to set obj_annot_ptr(k) ← tail here, it is not safe if nodes are later copied/destroyed/moved;
a better place to do this is inside do_link, when the whatsit node is written out }
end
```

This code is used in section 1528.

1561. `< Implement \pdfendlink 1561 >` ≡

```
begin check_pdfoutput("\pdfendlink", true);
if abs(mode) = vmode then pdf_error("ext1", "\pdfendlink cannot be used in vertical mode");
new_whatsit(pdf_end_link_node, small_node_size);
end
```

This code is used in section 1528.

1562. `< Declare procedures needed in do_extension 1529 >` +≡

```
function outline_list_count(p: pointer): integer;
{ return number of outline entries in the same level with p }
var k: integer;
begin k ← 1;
while obj_outline_prev(p) ≠ 0 do
begin incr(k); p ← obj_outline_prev(p);
end;
outline_list_count ← k;
end;
```

```

1563. < Implement \pdfoutline 1563 > ≡
begin check_pdfoutput("\pdfoutline", true);
if scan_keyword("attr") then
  begin scan-pdf-ext-toks; r ← def-ref;
  end
else r ← 0;
p ← scan-action;
if scan_keyword("count") then
  begin scan-int; i ← cur-val;
  end
else i ← 0;
scan-pdf-ext-toks; q ← def-ref; pdf_new_obj(obj_type_others, 0, 1); j ← obj_ptr; write_action(p);
pdf_end_obj; delete_action_ref(p); pdf_create_obj(obj_type_outline, 0); k ← obj_ptr;
obj_outline_ptr(k) ← pdf_get_mem(pdffmem_outline_size); obj_outline_action_objnum(k) ← j;
obj_outline_count(k) ← i; pdf_new_obj(obj_type_others, 0, 1); pdf_print_str_ln(tokens_to_string(q));
flush_str(last_tokens_string); delete_token_ref(q); pdf_end_obj; obj_outline_title(k) ← obj_ptr;
obj_outline_prev(k) ← 0; obj_outline_next(k) ← 0; obj_outline_first(k) ← 0; obj_outline_last(k) ← 0;
obj_outline_parent(k) ← pdf_parent_outline; obj_outline_attr(k) ← r;
if pdf_first_outline = 0 then pdf_first_outline ← k;
if pdf_last_outline = 0 then
  begin if pdf_parent_outline ≠ 0 then obj_outline_first(pdf_parent_outline) ← k;
  end
else begin obj_outline_next(pdf_last_outline) ← k; obj_outline_prev(k) ← pdf_last_outline;
  end;
pdf_last_outline ← k;
if obj_outline_count(k) ≠ 0 then
  begin pdf_parent_outline ← k; pdf_last_outline ← 0;
  end
else if (pdf_parent_outline ≠ 0) ∧ (outline_list_count(k) = abs(obj_outline_count(pdf_parent_outline)))
  then
    begin j ← pdf_last_outline;
    repeat obj_outline_last(pdf_parent_outline) ← j; j ← pdf_parent_outline;
      pdf_parent_outline ← obj_outline_parent(pdf_parent_outline);
    until (pdf_parent_outline = 0) ∨ (outline_list_count(j) < abs(obj_outline_count(pdf_parent_outline)));
    if pdf_parent_outline = 0 then pdf_last_outline ← pdf_first_outline
    else pdf_last_outline ← obj_outline_first(pdf_parent_outline);
    while obj_outline_next(pdf_last_outline) ≠ 0 do pdf_last_outline ← obj_outline_next(pdf_last_outline);
    end;
  end
end

```

This code is used in section 1528.

1564. When a destination is created we need to check whether another destination with the same identifier already exists and give a warning if needed.

```
< Declare procedures needed in pdf_hlist.out, pdf_vlist.out 727 > +≡  
procedure warn_dest_dup(id : integer; byname : small_number; s1, s2 : str_number);  
begin if pdf_suppress_warning_dup_dest > 0 then return;  
pdf_warning(s1, "destination with the same identifier", true, false);  
if byname > 0 then  
begin print("name"); print_mark(id);  
end  
else begin print("num"); print_int(id);  
end;  
print(")"); print(s2); print_ln; show_context;  
end;
```

1565. Notice that *scan_keyword* doesn't care if two words have same prefix; so we should be careful when scan keywords with same prefix. The main rule: if there are two or more keywords with the same prefix, then always test in order from the longest one to the shortest one.

```

⟨ Implement \pdfdest 1565 ⟩ ≡
begin check_pdfoutput("pdfdest", true); q ← tail; new_whatsit(pdf_dest_node, pdf_dest_node_size);
if scan_keyword("struct") then
begin scan_int;
if cur_val ≤ 0 then pdf_error("ext1", "struct_identifier_must_be_positive");
pdf_dest_objnum(tail) ← cur_val; j ← obj_type_struct_dest;
end
else begin pdf_dest_objnum(tail) ← null; j ← obj_type_dest;
end;
if scan_keyword("num") then
begin scan_int;
if cur_val ≤ 0 then pdf_error("ext1", "num_identifier_must_be_positive");
if cur_val > max_halfword then pdf_error("ext1", "number_too_big");
pdf_dest_id(tail) ← cur_val; pdf_dest_named_id(tail) ← 0;
end
else if scan_keyword("name") then
begin scan_pdf_ext_toks; pdf_dest_id(tail) ← def_ref; pdf_dest_named_id(tail) ← 1;
end
else pdf_error("ext1", "identifier_type_missing");
if scan_keyword("xyz") then
begin pdf_dest_type(tail) ← pdf_dest_xyz;
if scan_keyword("zoom") then
begin scan_int;
if cur_val > max_halfword then pdf_error("ext1", "number_too_big");
pdf_dest_xyz_zoom(tail) ← cur_val;
end
else pdf_dest_xyz_zoom(tail) ← null;
end
else if scan_keyword("fitbh") then pdf_dest_type(tail) ← pdf_dest_fitbh
else if scan_keyword("fitbv") then pdf_dest_type(tail) ← pdf_dest_fitbv
else if scan_keyword("fitb") then pdf_dest_type(tail) ← pdf_dest_fitb
else if scan_keyword("fith") then pdf_dest_type(tail) ← pdf_dest_fith
else if scan_keyword("fitv") then pdf_dest_type(tail) ← pdf_dest_fitv
else if scan_keyword("fitr") then pdf_dest_type(tail) ← pdf_dest_fitr
else if scan_keyword("fit") then pdf_dest_type(tail) ← pdf_dest_fit
else pdf_error("ext1", "destination_type_missing");
⟨ Scan an optional space 469 ⟩;
if pdf_dest_type(tail) = pdf_dest_fitr then
begin scan_alt_rule; { scans <rule spec> to alt_rule }
pdf_width(tail) ← width(alt_rule); pdf_height(tail) ← height(alt_rule);
pdf_depth(tail) ← depth(alt_rule);
end;
if pdf_dest_named_id(tail) ≠ 0 then
begin i ← tokens_to_string(pdf_dest_id(tail)); k ← find_obj(j, i, true); flush_str(i);
end
else k ← find_obj(j, pdf_dest_id(tail), false);
if (k ≠ 0) ∧ (obj_dest_ptr(k) ≠ null) then
begin warn_dest_dup(pdf_dest_id(tail), pdf_dest_named_id(tail), "ext4",
"has已被already used, duplicate ignored"); flush_node_list(tail); tail ← q;

```

```

link(q) ← null;
end;
end

```

This code is used in section 1528.

1566. ⟨ Declare procedures needed in *do_extension* 1529 ⟩ +≡

```

procedure scan_thread_id;
begin if scan_keyword("num") then
  begin scan_int;
  if cur_val ≤ 0 then pdf_error("ext1", "num_identifier_must_be_positive");
  if cur_val > max_halfword then pdf_error("ext1", "number_too_big");
  pdf_thread_id(tail) ← cur_val; pdf_thread_named_id(tail) ← 0;
  end
else if scan_keyword("name") then
  begin scan_pdf_ext_toks; pdf_thread_id(tail) ← def_ref; pdf_thread_named_id(tail) ← 1;
  end
else pdf_error("ext1", "identifier_type_missing");
end;

```

1567. ⟨ Implement \pdfthread 1567 ⟩ ≡

```

begin check_pdfoutput("\pdfthread", true); new_annot_whatsit(pdf_thread_node, pdf_thread_node_size);
scan_thread_id;
end

```

This code is used in section 1528.

1568. ⟨ Implement \pdfstartthread 1568 ⟩ ≡

```

begin check_pdfoutput("\pdfstartthread", true);
new_annot_whatsit(pdf_start_thread_node, pdf_thread_node_size); scan_thread_id;
end

```

This code is used in section 1528.

1569. ⟨ Implement \pdfendthread 1569 ⟩ ≡

```

begin check_pdfoutput("\pdfendthread", true); new_whatst(pdf_end_thread_node, small_node_size);
end

```

This code is used in section 1528.

1570. ⟨ Global variables 13 ⟩ +≡

```

pdf_last_x_pos: integer;
pdf_last_y_pos: integer;
pdf_snapx_refpos: integer;
pdf_snapy_refpos: integer;
count_do_snapy: integer;

```

1571. ⟨ Set initial values of key variables 21 ⟩ +≡

```

count_do_snapy ← 0;

```

1572. ⟨ Implement \pdfsnaprefpoint 1572 ⟩ ≡

```

begin check_pdfoutput("\pdfsnaprefpoint", true);
new_whatst(pdf_snap_ref_point_node, small_node_size);
end

```

This code is used in section 1528.

1573. *< Declare procedures needed in do_extension 1529 > +≡*

```
function new_snap_node(s : small_number): pointer;
  var p: pointer;
  begin scan_glue(glue_val);
  if width(cur_val) < 0 then pdf_error("ext1", "negative_snap_glue");
  p ← get_node(snap_node_size); type(p) ← whatsit_node; subtype(p) ← s; link(p) ← null;
  snap_glue_ptr(p) ← cur_val; final_skip(p) ← 0; new_snap_node ← p;
  end;
```

1574. *< Implement \pdfsnapy 1574 > ≡*

```
begin check_pdfoutput("\pdfsnapy", true); tail_append(new_snap_node(pdf_snapy_node));
end
```

This code is used in section 1528.

1575. *< Implement \pdfsnapycomp 1575 > ≡*

```
begin check_pdfoutput("\pdfsnapycomp", true); new_whatsit(pdf_snapy_comp_node, small_node_size);
  scan_int; snapy_comp_ratio(tail) ← fix_int(cur_val, 0, 1000);
end
```

This code is used in section 1528.

1576. *< Implement \pdfsavepos 1576 > ≡*

```
begin new_whatsit(pdf_save_pos_node, small_node_size);
end
```

This code is used in section 1528.

1577. To implement primitives as \pdfinfo, \pdfcatalog or \pdfnames we need to concatenate tokens lists.

< Declare procedures needed in do_extension 1529 > +≡

```
function concat_tokens(q, r : pointer): pointer; { concat q and r and returns the result tokens list }
  var p: pointer;
  begin if q = null then
    begin concat_tokens ← r; return;
    end;
  p ← q;
  while link(p) ≠ null do p ← link(p);
  link(p) ← link(r); free_avail(r); concat_tokens ← q;
  end;
```

1578. *< Implement \pdfinfo 1578 > ≡*

```
begin check_pdfoutput("\pdfinfo", false); scan_pdf_ext_toks;
  if pdf_output > 0 then pdf_info_toks ← concat_tokens(pdf_info_toks, def_ref);
end
```

This code is used in section 1528.

1579. $\langle \text{Implement } \backslash\text{pdfcatalog } 1579 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash\text{pdfcatalog}$ ", *false*); *scan_pdf_ext_toks*;
if *pdf_output* > 0 **then** *pdf_catalog_toks* \leftarrow *concat_tokens*(*pdf_catalog_toks*, *def_ref*);
if *scan_keyword*("openaction") **then**
 begin if *pdf_catalog_openaction* \neq 0 **then** *pdf_error*("ext1", "duplicate_of_openaction")
 else begin *p* \leftarrow *scan_action*; *pdf_new_obj*(*obj_type_others*, 0, 1);
 if *pdf_output* > 0 **then** *pdf_catalog_openaction* \leftarrow *obj_ptr*;
 write_action(*p*); *pdf_end_obj*; *delete_action_ref*(*p*);
 end;
 end
end

This code is used in section 1528.

1580. $\langle \text{Implement } \backslash\text{pdfnames } 1580 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash\text{pdfnames}$ ", *true*); *scan_pdf_ext_toks*;
pdf_names_toks \leftarrow *concat_tokens*(*pdf_names_toks*, *def_ref*);
end

This code is used in section 1528.

1581. $\langle \text{Implement } \backslash\text{pdftrailer } 1581 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash\text{pdftrailer}$ ", *false*); *scan_pdf_ext_toks*;
if *pdf_output* > 0 **then** *pdf_trailer_toks* \leftarrow *concat_tokens*(*pdf_trailer_toks*, *def_ref*);
end

This code is used in section 1528.

1582. $\langle \text{Implement } \backslash\text{pdftrailerid } 1582 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash\text{pdftrailerid}$ ", *false*); *scan_pdf_ext_toks*;
if *pdf_output* > 0 **then** *pdf_trailer_id_toks* \leftarrow *concat_tokens*(*pdf_trailer_id_toks*, *def_ref*);
end

This code is used in section 1528.

1583. $\langle \text{Global variables } 13 \rangle + \equiv$
pdf_retval: *integer*; { global multi-purpose return value }

1584. $\langle \text{Set initial values of key variables } 21 \rangle + \equiv$
seconds_and_micros(*epochseconds*, *microseconds*); *init_start_time*;

1585. Negative random seed values are silently converted to positive ones.

$\langle \text{Implement } \backslash\text{pdfsetrandomseed } 1585 \rangle \equiv$
begin *scan_int*;
if *cur_val* < 0 **then** *negate*(*cur_val*);
random_seed \leftarrow *cur_val*; *init_randoms*(*random_seed*);
end

This code is used in section 1528.

1586. $\langle \text{Implement } \backslash\text{pdfresettimer } 1586 \rangle \equiv$
begin *seconds_and_micros*(*epochseconds*, *microseconds*);
end

This code is used in section 1528.

1587. The following subroutines are about PDF-specific font issues.

```

⟨ Declare procedures needed in do_extension 1529 ⟩ +≡
procedure pdf_include_chars;
  var s: str_number; k: pool_pointer; { running indices }
  f: internal_font_number;
begin scan_font_ident; f ← cur_val;
if f = null_font then pdf_error("font", "invalid_font_identifier");
pdf_check_vf_cur_val;
if ¬font_used[f] then pdf_init_font(f);
scan_pdf_ext_toks; s ← tokens_to_string(def_ref); delete_token_ref(def_ref); k ← str_start[s];
while k < str_start[s + 1] do
  begin pdf_mark_char(f, str_pool[k]); incr(k);
  end;
flush_str(s);
end;
procedure glyph_to_unicode;
  var s1, s2: str_number;
begin scan_pdf_ext_toks; s1 ← tokens_to_string(def_ref); delete_token_ref(def_ref); scan_pdf_ext_toks;
s2 ← tokens_to_string(def_ref); delete_token_ref(def_ref); def_tounicode(s1, s2); flush_str(s2);
flush_str(s1);
end;

```

1588. ⟨ Implement \pdfincludechars 1588 ⟩ ≡

```

begin check_pdffoutput("\pdfincludechars", true); pdf_include_chars;
end

```

This code is used in section 1528.

1589. ⟨ Implement \pdffontattr 1589 ⟩ ≡

```

begin check_pdffoutput("\pdffontattr", true); scan_font_ident; k ← cur_val;
if k = null_font then pdf_error("font", "invalid_font_identifier");
scan_pdf_ext_toks; pdf_font_attr[k] ← tokens_to_string(def_ref);
end

```

This code is used in section 1528.

1590. ⟨ Implement \pdfmapfile 1590 ⟩ ≡

```

begin check_pdffoutput("\pdfmapfile", true); scan_pdf_ext_toks; pdfmapfile(def_ref);
delete_token_ref(def_ref);
end

```

This code is used in section 1528.

1591. ⟨ Implement \pdfmapline 1591 ⟩ ≡

```

begin check_pdffoutput("\pdfmapline", true); scan_pdf_ext_toks; pdfmapline(def_ref);
delete_token_ref(def_ref);
end

```

This code is used in section 1528.

1592. ⟨ Implement \pdffglyptounicode 1592 ⟩ ≡

```

begin glyph_to_unicode;
end

```

This code is used in section 1528.

1593. $\langle \text{Implement } \backslash \text{pdfnobuiltintounicode } 1593 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash \text{pdfnobuiltintounicode}$ ", *true*); *scan_font_ident*; *k* \leftarrow *cur_val*;
if *k* = *null_font* **then** *pdf_error*("font", "invalid font identifier");
pdf_font_nobuiltin_tounicode[*k*] \leftarrow *true*;
end

This code is used in section 1528.

1594. $\langle \text{Implement } \backslash \text{pdfinterwordspaceon } 1594 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash \text{pdfinterwordspaceon}$ ", *true*);
new_whatsit(*pdf_interword_space_on_node*, *small_node_size*);
end

This code is used in section 1528.

1595. $\langle \text{Implement } \backslash \text{pdfinterwordspaceoff } 1595 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash \text{pdfinterwordspaceoff}$ ", *true*);
new_whatsit(*pdf_interword_space_off_node*, *small_node_size*);
end

This code is used in section 1528.

1596. $\langle \text{Implement } \backslash \text{pdffakespace } 1596 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash \text{pdffakespace}$ ", *true*); *new_whatsit*(*pdf_fake_space_node*, *small_node_size*);
end

This code is used in section 1528.

1597. $\langle \text{Implement } \backslash \text{pdfrunninglinkoff } 1597 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash \text{pdfrunninglinkoff}$ ", *true*);
new_whatsit(*pdf_running_link_off_node*, *small_node_size*);
end

This code is used in section 1528.

1598. $\langle \text{Implement } \backslash \text{pdfrunninglinkon } 1598 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash \text{pdfrunninglinkon}$ ", *true*);
new_whatsit(*pdf_running_link_on_node*, *small_node_size*);
end

This code is used in section 1528.

1599. $\langle \text{Implement } \backslash \text{pdfspacefont } 1599 \rangle \equiv$
begin *check_pdfoutput*(" $\backslash \text{pdfspacefont}$ ", *true*); *scan_pdf_ext_toks*;
pdf_space_font_name \leftarrow *tokens_to_string*(*def_ref*); *delete_token_ref*(*def_ref*);
end

This code is used in section 1528.

1600. The following function are needed for outputting article thread.

```
< Declare procedures needed in do_extension 1529 > +≡
procedure thread_title(thread : integer);
begin pdf_print("/Title");
if obj_info(thread) < 0 then pdf_print(-obj_info(thread))
else pdf_print_int(obj_info(thread));
pdf_print_ln(")");
end;
procedure pdf_fix_thread(thread : integer);
var a: pointer;
begin pdf_warning("thread", "destination", true, false);
if obj_info(thread) < 0 then
begin print("name{"); print(-obj_info(thread)); print("}");
end
else begin print("num"); print_int(obj_info(thread));
end;
print("_has_referenced_but_does_not_exist,_replaced_by_a_fixed_one"); print_ln;
print_ln; pdf_new_dict(obj_type_others, 0, 0); a ← obj_ptr; pdf_indirect_ln("T", thread);
pdf_indirect_ln("V", a); pdf_indirect_ln("N", a); pdf_indirect_ln("P", head_tab[obj_type_page]);
pdf_print("/R[0_0]"); pdf_print_bp(pdf_page_width); pdf_out(" "); pdf_print_bp(pdf_page_height);
pdf_print_ln("]"); pdf_end_dict; pdf_begin_dict(thread, 1); pdf_print_ln("/I<<"); thread_title(thread);
pdf_print_ln(">>"); pdf_indirect_ln("F", a); pdf_end_dict;
end;
procedure out_thread(thread : integer);
var a, b: pointer; last_attr: integer;
begin if obj_thread_first(thread) = 0 then
begin pdf_fix_thread(thread); return;
end;
pdf_begin_dict(thread, 1); a ← obj_thread_first(thread); b ← a; last_attr ← 0;
repeat if obj_bead_attr(a) ≠ 0 then last_attr ← obj_bead_attr(a);
a ← obj_bead_next(a);
until a = b;
if last_attr ≠ 0 then pdf_print_ln(last_attr)
else begin pdf_print_ln("/I<<"); thread_title(thread); pdf_print_ln(">>");
end;
pdf_indirect_ln("F", a); pdf_end_dict;
repeat pdf_begin_dict(a, 1);
if a = b then pdf_indirect_ln("T", thread);
pdf_indirect_ln("V", obj_bead_prev(a)); pdf_indirect_ln("N", obj_bead_next(a));
pdf_indirect_ln("P", obj_bead_page(a)); pdf_indirect_ln("R", obj_bead_rect(a)); pdf_end_dict;
a ← obj_bead_next(a);
until a = b;
end;
```

1601. < Display jrule spec_j for whatsit node created by pdfTEX 1601 > ≡

```
print("("); print_rule_dimen(pdf_height(p)); print_char("+"); print_rule_dimen(pdf_depth(p));
print(")x"); print_rule_dimen(pdf_width(p))
```

This code is used in sections 1603, 1603, and 1603.

1602. Each new type of node that appears in our data structure must be capable of being displayed, copied, destroyed, and so on. The routines that we need for write-oriented whatsits are somewhat like those for mark nodes; other extensions might, of course, involve more subtlety here.

```
< Basic printing procedures 57 > +≡  
procedure print_write_whatsit(s : str_number; p : pointer);  
begin print_esc(s);  
if write_stream(p) < 16 then print_int(write_stream(p))  
else if write_stream(p) = 16 then print_char("*")  
else print_char("-");  
end;
```

1603. \langle Display the whatsit node p [1603](#) $\rangle \equiv$

```

case subtype( $p$ ) of
  open_node: begin print_write_whatsit("openout",  $p$ ); print_char("=");
    print_file_name(open_name( $p$ ), open_area( $p$ ), open_ext( $p$ ));
  end;
  write_node: begin print_write_whatsit("write",  $p$ ); print_mark(write_tokens( $p$ ));
  end;
  close_node: print_write_whatsit("closeout",  $p$ );
  special_node: begin print_esc("special"); print_mark(write_tokens( $p$ ));
  end;
  late_special_node: begin print_esc("special"); print("\u2193shipout"); print_mark(write_tokens( $p$ ));
  end;
  language_node: begin print_esc("setlanguage"); print_int(what_lang( $p$ )); print("\u2193(hyphenmin\u2193");
    print_int(what_lhm( $p$ )); print_char(","); print_int(what_rhm( $p$ )); print_char(")\"");
  end;
  pdf_literal_node, pdf_lateliteral_node: begin print_esc("pdfliteral");
    if subtype( $p$ ) = pdf_lateliteral_node then print("\u2193shipout");
    case pdf_literal_mode( $p$ ) of
      set_origin: do_nothing;
      direct_page: print("\u2193page");
      direct_always: print("\u2193direct");
      othercases confusion("literal2")
    endcases; print_mark(pdf_literal_data( $p$ ));
  end;
  pdf_colorstack_node: begin print_esc("pdfcolorstack\u2193"); print_int(pdf_colorstack_stack( $p$ ));
    case pdf_colorstack_cmd( $p$ ) of
      colorstack_set: print("\u2193set\u2193");
      colorstack_push: print("\u2193push\u2193");
      colorstack_pop: print("\u2193pop\u2193");
      colorstack_current: print("\u2193current\u2193");
      othercases confusion("pdfcolorstack")
    endcases;
    if pdf_colorstack_cmd( $p$ )  $\leq$  colorstack_data then print_mark(pdf_colorstack_data( $p$ ));
  end;
  pdf_setmatrix_node: begin print_esc("pdfsetmatrix"); print_mark(pdf_setmatrix_data( $p$ ));
  end;
  pdf_save_node: begin print_esc("pdfsave");
  end;
  pdf_restore_node: begin print_esc("pdfrestore");
  end;
  pdf_refobj_node: begin print_esc("pdfrefobj");
    if obj_obj_is_stream(pdf_obj_objnum( $p$ )) > 0 then
      begin if obj_obj_stream_attr(pdf_obj_objnum( $p$ ))  $\neq$  null then
        begin print("\u2193attr"); print_mark(obj_obj_stream_attr(pdf_obj_objnum( $p$ )));
        end;
        print("\u2193stream");
      end;
      if obj_obj_is_file(pdf_obj_objnum( $p$ )) > 0 then print("\u2193file");
      print_mark(obj_obj_data(pdf_obj_objnum( $p$ )));
    end;
  pdf_refxform_node: begin print_esc("pdfrefxform"); print("(");
    print_scaled(obj_xform_height(pdf_xform_objnum( $p$ ))); print_char("+");

```

```

print_scaled(obj_xform_depth(pdf_xform_objnum(p))); print(")x");
print_scaled(obj_xform_width(pdf_xform_objnum(p)));
end;

pdf_refximage_node: begin print_esc("pdfrefximage"); print("(");
print_scaled(obj_ximage_height(pdf_ximage_objnum(p))); print_char("+");
print_scaled(obj_ximage_depth(pdf_ximage_objnum(p))); print(")x");
print_scaled(obj_ximage_width(pdf_ximage_objnum(p)));
end;

pdf_annot_node: begin print_esc("pdfannot");
⟨ Display rule spec for whatsit node created by pdfTeX 1601 ⟩;
print_mark(pdf_annot_data(p));
end;

pdf_start_link_node: begin print_esc("pdfstartlink");
⟨ Display rule spec for whatsit node created by pdfTeX 1601 ⟩;
if pdf_link_attr(p) ≠ null then
begin print("↳attr"); print_mark(pdf_link_attr(p));
end;
print("↳action");
if pdf_action_type(pdf_link_action(p)) = pdf_action_user then
begin print("↳user"); print_mark(pdf_action_user_tokens(pdf_link_action(p)));
end
else begin if pdf_action_file(pdf_link_action(p)) ≠ null then
begin print("↳file"); print_mark(pdf_action_file(pdf_link_action(p)));
end;
case pdf_action_type(pdf_link_action(p)) of
pdf_action_goto: begin if (pdf_action_named_id(pdf_link_action(p)) mod 2) = 1 then
begin print("↳goto↳name"); print_mark(pdf_action_id(pdf_link_action(p)));
end
else begin print("↳goto↳num"); print_int(pdf_action_id(pdf_link_action(p)));
end;
end;
pdf_action_page: begin print("↳page"); print_int(pdf_action_id(pdf_link_action(p)));
print_mark(pdf_action_page_tokens(pdf_link_action(p)));
end;
pdf_action_thread: begin if (pdf_action_named_id(pdf_link_action(p)) mod 2) = 1 then
begin print("↳thread↳name"); print_mark(pdf_action_id(pdf_link_action(p)));
end
else begin print("↳thread↳num"); print_int(pdf_action_id(pdf_link_action(p)));
end;
end;
othercases pdf_error("displaying", "unknown↳action↳type");
endcases;
end
end;

pdf_end_link_node: print_esc("pdfendlink");
pdf_dest_node: begin print_esc("pdfdest");
if pdf_dest_objnum(p) ≠ null then
begin print("↳struct"); print_int(pdf_dest_objnum(p));
end;
if pdf_dest_named_id(p) > 0 then
begin print("↳name"); print_mark(pdf_dest_id(p));
end

```

```

else begin print(" \u00b9num"); print_int(pdf_dest_id(p));
  end;
print(" \u00b9");
case pdf_dest_type(p) of
pdf_dest_xyz: begin print("xyz");
  if pdf_dest_xyz_zoom(p) ≠ null then
    begin print(" \u00b9zoom"); print_int(pdf_dest_xyz_zoom(p));
    end;
  end;
pdf_dest_fitbh: print("fitbh");
pdf_dest_fitbv: print("fitbv");
pdf_dest_fitb: print("fitb");
pdf_dest_fith: print("fith");
pdf_dest_fitv: print("fitv");
pdf_dest_fitr: begin print("fitr"); ⟨Display rule spec for whatsit node created by pdfTeX 1601⟩;
  end;
pdf_dest_fit: print("fit");
othercases print("unknown!");
endcases;
end;

pdf_thread_node, pdf_start_thread_node: begin if subtype(p) = pdf_thread_node then
  print_esc("pdfthread")
else print_esc("pdfstartthread");
print("("); print_rule_dimen(pdf_height(p)); print_char("+"); print_rule_dimen(pdf_depth(p));
print("x"); print_rule_dimen(pdf_width(p));
if pdf_thread_attr(p) ≠ null then
  begin print(" \u00b9attr"); print_mark(pdf_thread_attr(p));
  end;
if pdf_thread_named_id(p) > 0 then
  begin print(" \u00b9name"); print_mark(pdf_thread_id(p));
  end
else begin print(" \u00b9num"); print_int(pdf_thread_id(p));
  end;
end;
pdf_end_thread_node: print_esc("pdfendthread");
pdf_save_pos_node: print_esc("pdfsavepos");
pdf_snap_ref_point_node: print_esc("pdfsnaprefpoint");
pdf_snapy_node: begin print_esc("pdfsnappy"); print_char(" \u00b9"); print_spec(snap_glue_ptr(p), 0);
  print_char(" \u00b9"); print_spec(final_skip(p), 0);
  end;
pdf_snapy_comp_node: begin print_esc("pdfsnappycomp"); print_char(" \u00b9");
  print_int(snapy_comp_ratio(p));
  end;
pdf_interword_space_on_node: print_esc("pdfinterwordspaceon");
pdf_interword_space_off_node: print_esc("pdfinterwordspaceoff");
pdf_fake_space_node: print_esc("pdffakespace");
pdf_running_link_off_node: print_esc("pdfrunninglinkoff");
pdf_running_link_on_node: print_esc("pdfrunninglinkon");
othercases print("whatsit?")
endcases

```

This code is used in section 201.

1604. ⟨ 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 1604 ⟩ ≡

```

case subtype( $p$ ) of
  open_node: begin  $r \leftarrow get\_node(open\_node\_size)$ ;  $words \leftarrow open\_node\_size$ ;
  end;
  write_node, special_node, latespecial_node: begin  $r \leftarrow get\_node(write\_node\_size)$ ;
    add_token_ref(write_tokens( $p$ ));  $words \leftarrow write\_node\_size$ ;
  end;
  close_node, language_node: begin  $r \leftarrow get\_node(small\_node\_size)$ ;  $words \leftarrow small\_node\_size$ ;
  end;
  pdf_literal_node, pdf_lateliteral_node: begin  $r \leftarrow get\_node(write\_node\_size)$ ;
    add_token_ref(pdf_literal_data( $p$ ));  $words \leftarrow write\_node\_size$ ;
  end;
  pdf_colorstack_node: begin if pdf_colorstack_cmd( $p$ ) ≤ colorstack_data then
    begin  $r \leftarrow get\_node(pdf\_colorstack\_setter\_node\_size)$ ; add_token_ref(pdf_colorstack_data( $p$ ));
     $words \leftarrow pdf\_colorstack\_setter\_node\_size$ ;
    end
  else begin  $r \leftarrow get\_node(pdf\_colorstack\_getter\_node\_size)$ ;  $words \leftarrow pdf\_colorstack\_getter\_node\_size$ ;
    end;
  end;
  pdf_setmatrix_node: begin  $r \leftarrow get\_node(pdf\_setmatrix\_node\_size)$ ; add_token_ref(pdf_setmatrix_data( $p$ ));
     $words \leftarrow pdf\_setmatrix\_node\_size$ ;
  end;
  pdf_save_node: begin  $r \leftarrow get\_node(pdf\_save\_node\_size)$ ;  $words \leftarrow pdf\_save\_node\_size$ ;
  end;
  pdf_restore_node: begin  $r \leftarrow get\_node(pdf\_restore\_node\_size)$ ;  $words \leftarrow pdf\_restore\_node\_size$ ;
  end;
  pdf_refobj_node: begin  $r \leftarrow get\_node(pdf\_refobj\_node\_size)$ ;  $words \leftarrow pdf\_refobj\_node\_size$ ;
  end;
  pdf_refxform_node: begin  $r \leftarrow get\_node(pdf\_refxform\_node\_size)$ ;  $words \leftarrow pdf\_refxform\_node\_size$ ;
  end;
  pdf_refximage_node: begin  $r \leftarrow get\_node(pdf\_refximage\_node\_size)$ ;  $words \leftarrow pdf\_refximage\_node\_size$ ;
  end;
  pdf_annot_node: begin  $r \leftarrow get\_node(pdf\_annot\_node\_size)$ ; add_token_ref(pdf_annot_data( $p$ ));
     $words \leftarrow pdf\_annot\_node\_size$ ;
  end;
  pdf_start_link_node: begin  $r \leftarrow get\_node(pdf\_annot\_node\_size)$ ; pdf_height( $r$ ) ← pdf_height( $p$ );
    pdf_depth( $r$ ) ← pdf_depth( $p$ ); pdf_width( $r$ ) ← pdf_width( $p$ ); pdf_link_attr( $r$ ) ← pdf_link_attr( $p$ );
    if pdf_link_attr( $r$ ) ≠ null then add_token_ref(pdf_link_attr( $r$ ));
    pdf_link_action( $r$ ) ← pdf_link_action( $p$ ); add_action_ref(pdf_link_action( $r$ ));
    pdf_link_objnum( $r$ ) ← pdf_link_objnum( $p$ );
  end;
  pdf_end_link_node:  $r \leftarrow get\_node(small\_node\_size)$ ;
  pdf_dest_node: begin  $r \leftarrow get\_node(pdf\_dest\_node\_size)$ ;
    if pdf_dest_named_id( $p$ ) > 0 then add_token_ref(pdf_dest_id( $p$ ));
     $words \leftarrow pdf\_dest\_node\_size$ ;
  end;
  pdf_thread_node, pdf_start_thread_node: begin  $r \leftarrow get\_node(pdf\_thread\_node\_size)$ ;
    if pdf_thread_named_id( $p$ ) > 0 then add_token_ref(pdf_thread_id( $p$ ));
    if pdf_thread_attr( $p$ ) ≠ null then add_token_ref(pdf_thread_attr( $p$ ));
     $words \leftarrow pdf\_thread\_node\_size$ ;
  end;

```

```
pdf_end_thread_node: r ← get_node(small_node_size);
pdf_save_pos_node: r ← get_node(small_node_size);
pdf_snap_ref_point_node: r ← get_node(small_node_size);
pdf_snapy_node: begin add_glue_ref(snap_glue_ptr(p)); r ← get_node(snap_node_size);
    words ← snap_node_size;
end;
pdf_snapy_comp_node: r ← get_node(small_node_size);
pdf_interword_space_on_node: r ← get_node(small_node_size);
pdf_interword_space_off_node: r ← get_node(small_node_size);
pdf_fake_space_node: r ← get_node(small_node_size);
pdf_running_link_off_node: r ← get_node(small_node_size);
pdf_running_link_on_node: r ← get_node(small_node_size);
othercases confusion("ext2")
endcases
```

This code is used in sections [224](#) and [1733](#).

```

1605. ⟨Wipe out the whatsit node p and goto done 1605⟩ ≡
begin case subtype(p) of
open_node: free_node(p, open_node_size);
write_node, special_node, latespecial_node: begin delete_token_ref(write_tokens(p));
  free_node(p, write_node_size); goto done;
end;
close_node, language_node: free_node(p, small_node_size);
pdf_literal_node, pdf_lateliteral_node: begin delete_token_ref(pdf_literal_data(p));
  free_node(p, write_node_size);
end;
pdf_colorstack_node: begin if pdf_colorstack_cmd(p) ≤ colorstack_data then
  begin delete_token_ref(pdf_colorstack_data(p)); free_node(p, pdf_colorstack_setter_node_size);
  end
  else free_node(p, pdf_colorstack_getter_node_size);
  end;
pdf_setmatrix_node: begin delete_token_ref(pdf_setmatrix_data(p)); free_node(p, pdf_setmatrix_node_size);
  end;
pdf_save_node: begin free_node(p, pdf_save_node_size);
  end;
pdf_restore_node: begin free_node(p, pdf_restore_node_size);
  end;
pdf_refobj_node: free_node(p, pdf_refobj_node_size);
pdf_refform_node: free_node(p, pdf_refform_node_size);
pdf_reffimage_node: free_node(p, pdf_reffimage_node_size);
pdf_annot_node: begin delete_token_ref(pdf_annot_data(p)); free_node(p, pdf_annot_node_size);
  end;
pdf_start_link_node: begin if pdf_link_attr(p) ≠ null then delete_token_ref(pdf_link_attr(p));
  delete_action_ref(pdf_link_action(p)); free_node(p, pdf_annot_node_size);
  end;
pdf_end_link_node: free_node(p, small_node_size);
pdf_dest_node: begin if pdf_dest_named_id(p) > 0 then delete_token_ref(pdf_dest_id(p));
  free_node(p, pdf_dest_node_size);
  end;
pdf_thread_node, pdf_start_thread_node: begin if pdf_thread_named_id(p) > 0 then
  delete_token_ref(pdf_thread_id(p));
  if pdf_thread_attr(p) ≠ null then delete_token_ref(pdf_thread_attr(p));
  free_node(p, pdf_thread_node_size);
  end;
pdf_end_thread_node: free_node(p, small_node_size);
pdf_save_pos_node: free_node(p, small_node_size);
pdf_snap_ref_point_node: free_node(p, small_node_size);
pdf_snapy_node: begin delete_glue_ref(snap_glue_ptr(p)); free_node(p, snap_node_size);
  end;
pdf_snapy_comp_node: free_node(p, small_node_size);
pdf_interword_space_on_node: free_node(p, small_node_size);
pdf_interword_space_off_node: free_node(p, small_node_size);
pdf_fake_space_node: free_node(p, small_node_size);
pdf_running_link_off_node: free_node(p, small_node_size);
pdf_running_link_on_node: free_node(p, small_node_size);
othercases confusion("ext3")
endcases;
goto done;

```

```
end
```

This code is used in section 220.

1606. *< Incorporate a whatsit node into a vbox 1606 >* ≡
if (*subtype(p) = pdf_refform_node*) ∨ (*subtype(p) = pdf_refximage_node*) **then**
 begin *x* ← *x* + *d* + *pdf_height(p)*; *d* ← *pdf_depth(p)*; *s* ← 0;
 if *pdf_width(p)* + *s* > *w* **then** *w* ← *pdf_width(p)* + *s*;
 end

This code is used in section 845.

1607. *< Incorporate a whatsit node into an hbox 1607 >* ≡
if (*subtype(p) = pdf_refform_node*) ∨ (*subtype(p) = pdf_refximage_node*) **then**
 begin *x* ← *x* + *pdf_width(p)*; *s* ← 0;
 if *pdf_height(p)* − *s* > *h* **then** *h* ← *pdf_height(p)* − *s*;
 if *pdf_depth(p)* + *s* > *d* **then** *d* ← *pdf_depth(p)* + *s*;
 end

This code is used in section 825.

1608. *< Let d be the width of the whatsit p 1608 >* ≡
if (*subtype(p) = pdf_refform_node*) ∨ (*subtype(p) = pdf_refximage_node*) **then** *d* ← *pdf_width(p)*
else *d* ← 0

This code is used in section 1325.

1609. **define** *adv_past(#)* ≡ **if** *subtype(#)* = *language_node* **then**
 begin *cur_lang* ← *what_lang(#)*; *l_hyf* ← *what_lhm(#)*; *r_hyf* ← *what_rhm(#)*; **end**
< Advance past a whatsit node in the line_break loop 1609 > ≡ **begin** *adv_past(cur_p)*;
 if (*subtype(cur_p) = pdf_refform_node*) ∨ (*subtype(cur_p) = pdf_refximage_node*) **then**
 act_width ← *act_width* + *pdf_width(cur_p)*;
 end

This code is used in section 1042.

1610. *< Advance past a whatsit node in the pre-hyphenation loop 1610 >* ≡ **if** *subtype(s)* = *language_node* **then**
 begin *cur_lang* ← *what_lang(s)*; *l_hyf* ← *what_lhm(s)*; *r_hyf* ← *what_rhm(s)*; *set_hyph_index*;
 end

This code is used in section 1073.

1611. *< Prepare to move whatsit p to the current page, then goto contribute 1611 >* ≡
begin **if** (*subtype(p) = pdf_refform_node*) ∨ (*subtype(p) = pdf_refximage_node*) **then**
 begin *page_total* ← *page_total* + *page_depth* + *pdf_height(p)*; *page_depth* ← *pdf_depth(p)*;
 end;
 goto *contribute*;
end

This code is used in section 1177.

1612. *< Process whatsit p in vert_break loop, goto not_found 1612 >* ≡
begin **if** (*subtype(p) = pdf_refform_node*) ∨ (*subtype(p) = pdf_refximage_node*) **then**
 begin *cur_height* ← *cur_height* + *prev_dp* + *pdf_height(p)*; *prev_dp* ← *pdf_depth(p)*;
 end;
 goto *not_found*;
end

This code is used in section 1150.

1613. \langle Output the whatsit node p in a vlist [1613](#) $\rangle \equiv$
 $out_what(p)$

This code is used in section [659](#).

1614. \langle Output the whatsit node p in an hlist [1614](#) $\rangle \equiv$
 $out_what(p)$

This code is used in section [650](#).

1615. After all this preliminary shuffling, we come finally to the routines that actually send out the requested data. Let's do `\special` first (it's easier).

\langle Declare procedures needed in $hlist_out$, $vlist_out$ [1615](#) $\rangle \equiv$

procedure $special_out(p : pointer);$

```

var  $old\_setting$ : 0 ..  $max\_selector$ ; { holds print selector }
 $k$ :  $pool\_pointer$ ; { index into str_pool }
 $h$ :  $halfword$ ;  $q, r$ :  $pointer$ ; { temporary variables for list manipulation }
 $old\_mode$ :  $integer$ ; { saved mode }
begin  $synch\_h$ ;  $synch\_v$ ;
 $old\_setting \leftarrow selector$ ;  $selector \leftarrow new\_string$ ;  $selector \leftarrow old\_setting$ ;
if  $subtype(p) = late special\_node$  then
  begin { Expand macros in the token list and make  $link(def\_ref)$  point to the result 1618 };
     $h \leftarrow def\_ref$ ;
  end
else  $h \leftarrow write\_tokens(p)$ ;
 $selector \leftarrow new\_string$ ;  $show\_token\_list(link(h), null, pool\_size - pool\_ptr)$ ;  $selector \leftarrow old\_setting$ ;
 $str\_room(1)$ ;
if  $cur\_length < 256$  then
  begin  $dvi\_out(xxx1)$ ;  $dvi\_out(cur\_length)$ ;
  end
else begin  $dvi\_out(xxx4)$ ;  $dvi\_four(cur\_length)$ ;
  end;
for  $k \leftarrow str\_start[str\_ptr]$  to  $pool\_ptr - 1$  do  $dvi\_out(so(str\_pool[k]))$ ;
 $pool\_ptr \leftarrow str\_start[str\_ptr]$ ; { erase the string }
if  $subtype(p) = late special\_node$  then  $flush\_list(def\_ref)$ ;
end;
```

See also sections [1617](#), [1620](#), [1719](#), and [1723](#).

This code is used in section [647](#).

1616. To write a token list, we must run it through TeX's scanner, expanding macros and `\the` and `\number`, etc. This might cause runaways, if a delimited macro parameter isn't matched, and runaways would be extremely confusing since we are calling on TeX's scanner in the middle of a `\shipout` command. Therefore we will put a dummy control sequence as a "stopper," right after the token list. This control sequence is artificially defined to be `\outer`.

\langle Initialize table entries (done by INITEX only) [182](#) $\rangle +\equiv$

```

 $text(end\_write) \leftarrow "endwrite"$ ;  $eq\_level(end\_write) \leftarrow level\_one$ ;  $eq\_type(end\_write) \leftarrow outer\_call$ ;
 $equiv(end\_write) \leftarrow null$ ;
```

1617. ⟨Declare procedures needed in *hlist_out*, *vlist_out* 1615⟩ +≡

```

procedure write_out(p : pointer);
  var old_setting: 0 .. max_selector; { holds print selector }
  old_mode: integer; { saved mode }
  j: small_number; { write stream number }
  q, r: pointer; { temporary variables for list manipulation }
begin ⟨Expand macros in the token list and make link(def_ref) point to the result 1618⟩;
  old_setting ← selector; j ← write_stream(p);
  if write_open[j] then selector ← j
  else begin { write to the terminal if file isn't open }
    if (j = 17) ∧ (selector = term_and_log) then selector ← log_only;
    print_nl("");
    end;
  token_show(def_ref); print_ln; flush_list(def_ref); selector ← old_setting;
end;
```

1618. The final line of this routine is slightly subtle; at least, the author didn't think about it until getting burnt! There is a used-up token list on the stack, namely the one that contained *end_write_token*. (We insert this artificial ‘\endwrite’ to prevent runaways, as explained above.) If it were not removed, and if there were numerous writes on a single page, the stack would overflow.

```

define end_write_token ≡ cs_token_flag + end_write
⟨Expand macros in the token list and make link(def_ref) point to the result 1618⟩ ≡
  q ← get_avail; info(q) ← right_brace_token + "}";
  r ← get_avail; link(q) ← r; info(r) ← end_write_token; ins_list(q);
  begin_token_list(write_tokens(p), write_text);
  q ← get_avail; info(q) ← left_brace_token + "{"; ins_list(q);
  { now we're ready to scan '{( token list )} \endwrite'}
  old_mode ← mode; mode ← 0; { disable \prevdepth, \spacefactor, \lastskip, \prevgraf }
  cur_cs ← write_loc; q ← scan_toks(false, true); { expand macros, etc. }
  get_token; if cur_tok ≠ end_write_token then ⟨Recover from an unbalanced write command 1619⟩;
  mode ← old_mode; end_token_list { conserve stack space }
```

This code is used in sections 727, 727, 1615, and 1617.

1619. ⟨Recover from an unbalanced write command 1619⟩ ≡

```

begin print_err("Unbalanced\\write\\command");
  help2("On\\this\\page\\there\\'s\\a\\write\\with\\fewer\\real\\{'s\\than\\}\\'s\\.");
  ("I\\can\\'t\\handle\\that\\very\\well;\\good\\luck."); error;
repeat get_token;
until cur_tok = end_write_token;
end
```

This code is used in section 1618.

1620. The *out.what* procedure takes care of outputting whatsit nodes for *vlist_out* and *hlist_out*.

```
< Declare procedures needed in hlist_out, vlist_out 1615 > +≡
procedure out.what(p : pointer);
  var j: small_number; { write stream number }
  begin case subtype(p) of
    open_node, write_node, close_node: < Do some work that has been queued up for \write 1622 >;
    special_node, latespecial_node: special_out(p);
    language_node: do_nothing;
    pdf_save_pos_node: < Save current position in DVI mode 1621 >;
    others: begin if (pdftex_first_extension_code ≤ subtype(p)) ∧ (subtype(p) ≤ pdftex_last_extension_code)
              then pdf_error("ext4", "pdf_node-ended-up-in-DVI-mode")
              else confusion("ext4")
            end;
    endcases;
  end;
```

1621. < Save current position in DVI mode 1621 > ≡

```
begin { 4736286 = 1in, the funny DVI origin offset }
  pdf_last_x_pos ← cur_h + 4736286; pdf_last_y_pos ← cur_page_height - cur_v - 4736286;
end
```

This code is used in section 1620.

1622. We don't implement \write inside of leaders. (The reason is that the number of times a leader box appears might be different in different implementations, due to machine-dependent rounding in the glue calculations.)

```
< Do some work that has been queued up for \write 1622 > ≡
if ¬doing_leaders then
  begin j ← write_stream(p);
  if subtype(p) = write_node then write_out(p)
  else begin if write_open[j] then a_close(write_file[j]);
           if subtype(p) = close_node then write_open[j] ← false
           else if j < 16 then
             begin cur_name ← open_name(p); cur_area ← open_area(p); cur_ext ← open_ext(p);
             if cur_ext = "" then cur_ext ← ".tex";
             pack_cur_name;
             while ¬a_open_out(write_file[j]) do prompt_file_name("output_file_name", ".tex");
             write_open[j] ← true;
           end;
         end;
       end;
end
```

This code is used in section 1620.

1623. The presence of ‘\immediate’ causes the *do_extension* procedure to descend to one level of recursion. Nothing happens unless \immediate is followed by ‘\openout’, ‘\write’, or ‘\closeout’.

```
< Implement \immediate 1623 > ≡
begin get_x_token;
if cur_cmd = extension then
begin if cur_chr ≤ close_node then
begin p ← tail; do_extension; { append a whatsit node }
out_what(tail); { do the action immediately }
flush_node_list(tail); tail ← p; link(p) ← null;
end
else case cur_chr of
pdf_obj_code: begin do_extension; { scan object and set pdf_last_obj }
if obj_data_ptr(pdf_last_obj) = 0 then { this object has not been initialized yet }
pdf_error("ext1", " ``\pdfobj\reserveobjnum' cannot be used with \immediate");
pdf_write_obj(pdf_last_obj);
end;
pdf_xform_code: begin do_extension; { scan form and set pdf_last_xform }
pdf_cur_form ← pdf_last_xform; pdf_ship_out(obj_xform_box(pdf_last_xform), false);
end;
pdf_ximage_code: begin do_extension; { scan image and set pdf_last_ximage }
pdf_write_image(pdf_last_ximage);
end;
othercases back_input
endcases;
end
else back_input;
end
```

This code is used in section 1528.

1624. The \language extension is somewhat different. We need a subroutine that comes into play when a character of a non-*clang* language is being appended to the current paragraph.

```
< Declare action procedures for use by main_control 1221 > +≡
procedure fix_language;
var l: ASCII_code; { the new current language }
begin if language ≤ 0 then l ← 0
else if language > 255 then l ← 0
else l ← language;
if l ≠ clang then
begin new_whatsit(language_node, small_node_size); what_lang(tail) ← l; clang ← l;
what_lhm(tail) ← norm_min(left_hyphen_min); what_rhm(tail) ← norm_min(right_hyphen_min);
end;
end;
```

```
1625. ⟨Implement \setlanguage 1625⟩ ≡
  if abs(mode) ≠ hmode then report_illegal_case
  else begin new_whatsit(language_node, small_node_size); scan_int;
    if cur_val ≤ 0 then clang ← 0
    else if cur_val > 255 then clang ← 0
      else clang ← cur_val;
    what_lang(tail) ← clang; what_lhm(tail) ← norm_min(left_hyphen_min);
    what_rhm(tail) ← norm_min(right_hyphen_min);
  end
```

This code is used in section 1528.

```
1626. ⟨Finish the extensions 1626⟩ ≡
  for k ← 0 to 15 do
    if write_open[k] then a_close(write_file[k])
```

This code is used in section 1513.

1627. Shipping out PDF marks.

```
⟨Types in the outer block 18⟩ +≡
dest_name_entry = record objname: str_number; { destination name }
objnum: integer; { destination object number }
end;
```

1628. $\langle \text{Global variables } 13 \rangle +\equiv$

`cur_page_width: scaled;` { width of page being shipped }
`cur_page_height: scaled;` { height of page being shipped }
`cur_h_offset: scaled;` { horizontal offset of page being shipped }
`cur_v_offset: scaled;` { vertical offset of page being shipped }
`pdf_obj_list: pointer;` { list of objects in the current page }
`pdf_xform_list: pointer;` { list of forms in the current page }
`pdf_ximage_list: pointer;` { list of images in the current page }
`last_thread: pointer;` { pointer to the last thread }
`pdf_thread_ht, pdf_thread_dp, pdf_thread_wd: scaled;` { dimensions of the last thread }
`pdf_last_thread_id: halfword;` { identifier of the last thread }
`pdf_last_thread_named_id: boolean;` { is identifier of the last thread named }
`pdf_thread_level: integer;` { depth of nesting of box containing the last thread }
`pdf_annot_list: pointer;` { list of annotations in the current page }
`pdf_link_list: pointer;` { list of link annotations in the current page }
`pdf_dest_list: pointer;` { list of destinations in the current page }
`pdf_bead_list: pointer;` { list of thread beads in the current page }
`pdf_obj_count: integer;` { counter of objects }
`pdf_xform_count: integer;` { counter of forms }
`pdf_ximage_count: integer;` { counter of images }
`pdf_cur_form: integer;` { the form being output }
`pdf_first_outline, pdf_last_outline, pdf_parent_outline: integer;`
`pdf_xform_width, pdf_xform_height, pdf_xform_depth: scaled;` { dimension of the current form }
`pdf_info_toks: pointer;` { additional keys of Info dictionary }
`pdf_catalog_toks: pointer;` { additional keys of Catalog dictionary }
`pdf_catalog_openaction: integer;`
`pdf_names_toks: pointer;` { additional keys of Names dictionary }
`pdf_dest_names_ptr: integer;` { first unused position in dest_names }
`dest_names_size: integer;` { maximum number of names in name tree of PDF output file }
`dest_names: \uparrow dest_name_entry;`
`pk_dpi: integer;` { PK pixel density value from `texmf.cnf` }
`image_orig_x, image_orig_y: integer;` { origin of cropped PDF images }
`pdf_trailer_toks: pointer;` { additional keys of Trailer dictionary }
`pdf_trailer_id_toks: pointer;` { custom Trailer ID }
`gen_faked_interword_space: boolean;` { flag to turn on/off faked interword spaces }
`gen_running_link: boolean;` { flag to turn on/off running link }
`pdf_space_font_name: str_number;` { name of font used for inter-word space in PDF output }

1629. $\langle \text{Set initial values of key variables } 21 \rangle +\equiv$

`pdf_first_outline \leftarrow 0; pdf_last_outline \leftarrow 0; pdf_parent_outline \leftarrow 0; pdf_obj_count \leftarrow 0;`
`pdf_xform_count \leftarrow 0; pdf_ximage_count \leftarrow 0; pdf_dest_names_ptr \leftarrow 0; pdf_info_toks \leftarrow null;`
`pdf_catalog_toks \leftarrow null; pdf_names_toks \leftarrow null; pdf_catalog_openaction \leftarrow 0; pdf_trailer_toks \leftarrow null;`
`pdf_trailer_id_toks \leftarrow null; gen_faked_interword_space \leftarrow false; gen_running_link \leftarrow true;`
`pdf_space_font_name \leftarrow "pdftexspace";`

1630. The following procedures are needed for outputting whatsit nodes for pdfTeX.

```

⟨ Declare procedures needed in pdf_hlist_out, pdf_vlist_out 727 ⟩ +≡
procedure write_action(p : pointer); { write an action specification }
  var s: str_number; d: integer;
  begin if pdf_action_type(p) = pdf_action_user then
    begin pdf_print_toks_ln(pdf_action_user_tokens(p)); return;
    end;
    pdf_print("<<„");
    if pdf_action_file(p) ≠ null then
      begin pdf_print("F„"); s ← tokens_to_string(pdf_action_file(p));
      if (str_pool[str_start[s] = 40) ∧ (str_pool[str_start[s] + length(s) – 1] = 41) then pdf_print(s)
      else begin pdf_print_str(s);
      end;
      flush_str(s); pdf_print("„");
      if pdf_action_new_window(p) > 0 then
        begin pdf_print("NewWindow„");
        if pdf_action_new_window(p) = 1 then pdf_print("true„")
        else pdf_print("false„");
        end;
      end;
    case pdf_action_type(p) of
      pdf_action_page: begin if pdf_action_file(p) = null then
        begin pdf_print("S„/GoTo„/D„["); pdf_print_int(get_obj(obj_type_page, pdf_action_id(p), false));
        pdf_print("„0„R");
        end
      else begin pdf_print("S„/GoToR„/D„["); pdf_print_int(pdf_action_id(p) – 1);
        end;
      pdf_out("„"); pdf_print(tokens_to_string(pdf_action_page_tokens(p))); flush_str(last_tokens_string);
      pdf_out("]");
      end;
      pdf_action_goto: begin if pdf_action_file(p) = null then
        begin pdf_print("S„/GoTo„");
        d ← get_obj(obj_type_dest, pdf_action_id(p), pdf_action_named_id(p) mod 2);
        end
      else pdf_print("S„/GoToR„");
      if (pdf_action_named_id(p) mod 2) = 1 then
        begin pdf_str_entry("D", tokens_to_string(pdf_action_id(p))); flush_str(last_tokens_string);
        end
      else if pdf_action_file(p) = null then pdf_indirect("D", d)
      else pdf_error("ext4", "goto` „option „cannot „be „used „with „both „file „and „num „");
      end;
      pdf_action_thread: begin pdf_print("S„/Thread„");
      if pdf_action_file(p) = null then
        d ← get_obj(obj_type_thread, pdf_action_id(p), pdf_action_named_id(p) mod 2);
      if (pdf_action_named_id(p) mod 2) = 1 then
        begin pdf_str_entry("D", tokens_to_string(pdf_action_id(p))); flush_str(last_tokens_string);
        end
      else if pdf_action_file(p) = null then pdf_indirect("D", d)
      else pdf_int_entry("D", pdf_action_id(p));
      end;
    endcases;
    if pdf_action_struct_id(p) ≠ null then

```

```

begin pdf_out("„");
if pdf_action_file(p) = null then pdf_indirect("SD", get_obj(obj_type_struct_dest,
    pdf_action_struct_id(p), (pdf_action_named_id(p) div 2) mod 2))
else begin pdf_print("/SD„"); pdf_print(tokens_to_string(pdf_action_struct_id(p)));
    flush_str(last_tokens_string);
end;
end;
pdf_print_ln("„>>");
end;

procedure set_rect_dimens(p, parent_box : pointer; x, y, w, h, d, margin : scaled);
begin pdf_left(p)  $\leftarrow$  cur_h;
if is_running(w) then pdf_right(p)  $\leftarrow$  x + width(parent_box)
else pdf_right(p)  $\leftarrow$  cur_h + w;
if is_running(h) then pdf_top(p)  $\leftarrow$  y - height(parent_box)
else pdf_top(p)  $\leftarrow$  cur_v - h;
if is_running(d) then pdf_bottom(p)  $\leftarrow$  y + depth(parent_box)
else pdf_bottom(p)  $\leftarrow$  cur_v + d;
if is_shipping_page  $\wedge$  matrixused then
    begin matrixtransformrect(pdf_left(p), cur_page_height - pdf_bottom(p), pdf_right(p),
        cur_page_height - pdf_top(p)); pdf_left(p)  $\leftarrow$  getllx; pdf_bottom(p)  $\leftarrow$  cur_page_height - getlly;
        pdf_right(p)  $\leftarrow$  geturx; pdf_top(p)  $\leftarrow$  cur_page_height - getury;
    end;
    pdf_left(p)  $\leftarrow$  pdf_left(p) - margin; pdf_top(p)  $\leftarrow$  pdf_top(p) - margin;
    pdf_right(p)  $\leftarrow$  pdf_right(p) + margin; pdf_bottom(p)  $\leftarrow$  pdf_bottom(p) + margin;
end;
procedure do_annot(p, parent_box : pointer; x, y : scaled);
begin if  $\neg$ is_shipping_page then pdf_error("ext4", "annotations cannot be inside an XForm");
if doing_leaders then return;
if is_obj_scheduled(pdf_annot_objnum(p)) then pdf_annot_objnum(p)  $\leftarrow$  pdf_new_objnum;
set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), 0);
obj_annot_ptr(pdf_annot_objnum(p))  $\leftarrow$  p; pdf_append_list(pdf_annot_objnum(p))(pdf_annot_list);
set_obj_scheduled(pdf_annot_objnum(p));
end;

```

1631. To implement nested link annotations, we need a stack to hold copy of *pdf_start_link_node*'s that are being written out, together with their box nesting level.

```

define pdf_link_stack_top  $\equiv$  pdf_link_stack[pdf_link_stack_ptr]
 $\langle$  Constants in the outer block 11  $\rangle$   $\equiv$ 
pdf_max_link_level = 10; { maximum depth of link nesting }

```

1632. \langle Types in the outer block 18 \rangle \equiv

```

pdf_link_stack_record = record nesting_level: integer;
    link_node: pointer; { holds a copy of the corresponding pdf_start_link_node }
    ref_link_node: pointer; { points to original pdf_start_link_node, or a copy of link_node created by
        append_link in case of multi-line link }
end;

```

1633. \langle Global variables 13 \rangle \equiv

```

pdf_link_stack: array [1 .. pdf_max_link_level] of pdf_link_stack_record;
pdf_link_stack_ptr: small_number;

```

1634. ⟨ Set initial values of key variables 21 ⟩ +≡
 $\text{pdf_link_stack_ptr} \leftarrow 0;$

1635. ⟨ Declare procedures needed in pdf_hlist_out , pdf_vlist_out 727 ⟩ +≡
procedure $\text{push_link_level}(p : \text{pointer})$;
 begin if $\text{pdf_link_stack_ptr} \geq \text{pdf_max_link_level}$ **then**
 overflow("pdf_link_stack_size", $\text{pdf_max_link_level}$);
 pdfassert(($\text{type}(p) = \text{whatsit_node}$) \wedge ($\text{subtype}(p) = \text{pdf_start_link_node}$)); $\text{incr}(\text{pdf_link_stack_ptr})$;
 $\text{pdf_link_stack_top.nesting_level} \leftarrow \text{cur_s}$; $\text{pdf_link_stack_top.link_node} \leftarrow \text{copy_node_list}(p)$;
 $\text{pdf_link_stack_top.ref_link_node} \leftarrow p$;
 end;
procedure pop_link_level ;
 begin $\text{pdfassert}(\text{pdf_link_stack_ptr} > 0)$; $\text{flush_node_list}(\text{pdf_link_stack_top.link_node})$;
 decr($\text{pdf_link_stack_ptr}$);
 end;
procedure $\text{do_link}(p, \text{parent_box} : \text{pointer}; x, y : \text{scaled})$;
 begin if $\neg \text{is_shipping_page}$ **then**
 pdf_error("ext4", "link_annotations_cannot_be_inside_an_XForm");
 pdfassert($\text{type}(\text{parent_box}) = \text{hlist_node}$);
 if $\text{is_obj_scheduled}(\text{pdf_link_objnum}(p))$ **then** $\text{pdf_link_objnum}(p) \leftarrow \text{pdf_new_objnum}$;
 push_link_level(p);
 set_rect_dimens($p, \text{parent_box}, x, y, \text{pdf_width}(p), \text{pdf_height}(p), \text{pdf_depth}(p), \text{pdf_link_margin}$);
 $\text{obj_annot_ptr}(\text{pdf_link_objnum}(p)) \leftarrow p$; { the reference for the pdf annot object must be set here }
 pdf_append_list($\text{pdf_link_objnum}(p)$)(pdf_link_list); **set_obj_scheduled**($\text{pdf_link_objnum}(p)$);
 end;
procedure end_link ;
 var $p : \text{pointer}$;
 begin if $\text{pdf_link_stack_ptr} < 1$ **then**
 pdf_error("ext4", "pdf_link_stack_empty, \pdfendlink used without \pdfstartlink");
 if $\text{pdf_link_stack_top.nesting_level} \neq \text{cur_s}$ **then** $\text{pdf_warning}(0,$
 "\pdfendlink ended up in different nesting level than \pdfstartlink", true, true);
 { N.B.: test for running link must be done on link_node and not ref_link_node , as ref_link_node can
 be set by do_link or append_link already }
 if $\text{is_running}(\text{pdf_width}(\text{pdf_link_stack_top.link_node}))$ **then**
 begin $p \leftarrow \text{pdf_link_stack_top.ref_link_node}$;
 if $\text{is_shipping_page} \wedge \text{matrixused}$ **then**
 begin $\text{matrixrecalculate}(\text{cur_h} + \text{pdf_link_margin})$; $\text{pdf_left}(p) \leftarrow \text{getllx} - \text{pdf_link_margin}$;
 $\text{pdf_top}(p) \leftarrow \text{cur_page_height} - \text{getury} - \text{pdf_link_margin}$; $\text{pdf_right}(p) \leftarrow \text{geturx} + \text{pdf_link_margin}$;
 $\text{pdf_bottom}(p) \leftarrow \text{cur_page_height} - \text{getlly} + \text{pdf_link_margin}$;
 end
 else $\text{pdf_right}(p) \leftarrow \text{cur_h} + \text{pdf_link_margin}$;
 end;
 pop_link_level;
 end;

1636. For “running” annotations we must append a new node when the end of annotation is in other box than its start. The new created node is identical to corresponding whatsit node representing the start of annotation, but its *info* field is *max_halfword*. We set *info* field just before destroying the node, in order to use *flush_node_list* to do the job.

```
< Declare procedures needed in pdf_hlist_out, pdf_vlist_out 727 > +≡  
procedure append_link(parent_box : pointer; x,y : scaled; i : small_number);  
    { append a new pdf annot to pdf_link_list }  
var p: pointer;  
begin pdfassert(type(parent_box) = hlist_node); p ← copy_node_list(pdf_link_stack[i].link_node);  
pdf_link_stack[i].ref_link_node ← p; info(p) ← max_halfword;  
    { mark that this node is not a whatsit node }  
link(p) ← null; { this node is not linked in any list }  
set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_link_margin);  
pdf_create_obj(obj_type_others, 0); obj_annot_ptr(obj_ptr) ← p; pdf_append_list(obj_ptr)(pdf_link_list);  
end;
```

1637. Threads are handled in similar way as link annotations.

```

< Declare procedures needed in pdf_hlist_out, pdf_vlist_out 727 > +≡
procedure append_bead(p : pointer);
  var a, b, c, t: integer;
  begin if ¬is_shipping_page then pdf_error("ext4", "threads cannot be inside an XForm");
  t ← get_obj(obj_type_thread, pdf_thread_id(p), pdf_thread_named_id(p)); b ← pdf_new_objnum;
  obj_bead_ptr(b) ← pdf_get_mem(pdffmem_bead_size); obj_bead_page(b) ← pdf_last_page;
  obj_bead_data(b) ← p;
  if pdf_thread_attr(p) ≠ null then obj_bead_attr(b) ← tokens_to_string(pdf_thread_attr(p))
  else obj_bead_attr(b) ← 0;
  if obj_thread_first(t) = 0 then
    begin obj_thread_first(t) ← b; obj_bead_next(b) ← b; obj_bead_prev(b) ← b;
    end
  else begin a ← obj_thread_first(t); c ← obj_bead_prev(a); obj_bead_prev(b) ← c; obj_bead_next(b) ← a;
  obj_bead_prev(a) ← b; obj_bead_next(c) ← b;
  end;
  pdf_append_list(b)(pdf_bead_list);
  end;
procedure do_thread(p, parent_box : pointer; x, y : scaled);
  begin if doing_leaders then return;
  if subtype(p) = pdf_start_thread_node then
    begin pdf_thread_wd ← pdf_width(p); pdf_thread_ht ← pdf_height(p); pdf_thread_dp ← pdf_depth(p);
    pdf_last_thread_id ← pdf_thread_id(p); pdf_last_thread_named_id ← (pdf_thread_named_id(p) > 0);
    if pdf_last_thread_named_id then add_token_ref(pdf_thread_id(p));
    pdf_thread_level ← cur_s;
    end;
  set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_thread_margin);
  append_bead(p); last_thread ← p;
  end;
procedure append_thread(parent_box : pointer; x, y : scaled);
  var p: pointer;
  begin p ← get_node(pdf_thread_node_size); info(p) ← max_halfword; { this is not a whatsit node }
  link(p) ← null; { this node will be destroyed separately }
  pdf_width(p) ← pdf_thread_wd; pdf_height(p) ← pdf_thread_ht; pdf_depth(p) ← pdf_thread_dp;
  pdf_thread_attr(p) ← null; pdf_thread_id(p) ← pdf_last_thread_id;
  if pdf_last_thread_named_id then
    begin add_token_ref(pdf_thread_id(p)); pdf_thread_named_id(p) ← 1;
    end
  else pdf_thread_named_id(p) ← 0;
  set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_thread_margin);
  append_bead(p); last_thread ← p;
  end;
procedure end_thread;
  begin if pdf_thread_level ≠ cur_s then pdf_error("ext4",
  "\pdfendthread ended up in different nesting level than \pdfstartthread");
  if is_running(pdf_thread_dp) ∧ (last_thread ≠ null) then
    pdf_bottom(last_thread) ← cur_v + pdf_thread_margin;
  if pdf_last_thread_named_id then delete_token_ref(pdf_last_thread_id);
  last_thread ← null;
  end;
function open_subentries(p : pointer): integer;
  var k, c: integer; l, r: integer;

```

```

begin k ← 0;
if obj_outline_first(p) ≠ 0 then
  begin l ← obj_outline_first(p);
  repeat incr(k); c ← open_subentries(l);
    if obj_outline_count(l) > 0 then k ← k + c;
    obj_outline_parent(l) ← p; r ← obj_outline_next(l);
    if r = 0 then obj_outline_last(p) ← l;
    l ← r;
  until l = 0;
end;

if obj_outline_count(p) > 0 then obj_outline_count(p) ← k
else obj_outline_count(p) ← -k;
open_subentries ← k;
end;

procedure do_dest(p, parent_box : pointer; x, y : scaled);
var k: integer;
begin if ¬is_shipping_page then pdf_error("ext4", "destinations cannot be inside an XForm");
if doing_leaders then return;
if pdf_dest_objnum(p) = null then k ← get_obj(obj_type_dest, pdf_dest_id(p), pdf_dest_named_id(p))
else k ← get_obj(obj_type_struct_dest, pdf_dest_id(p), pdf_dest_named_id(p));
if obj_dest_ptr(k) ≠ null then
  begin warn_dest_dup(pdf_dest_id(p), pdf_dest_named_id(p), "ext4",
    "has been already used, duplicate ignored"); return;
  end;
obj_dest_ptr(k) ← p; pdf_append_list(k)(pdf_dest_list);
case pdf_dest_type(p) of
  pdf_dest_xyz: if matrixused then
    set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_dest_margin)
  else begin pdf_left(p) ← cur_h; pdf_top(p) ← cur_v;
  end;
  pdf_dest_fith, pdf_dest_fitbh: if matrixused then
    set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_dest_margin)
  else pdf_top(p) ← cur_v;
  pdf_dest_fitv, pdf_dest_fitbv: if matrixused then
    set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_dest_margin)
  else pdf_left(p) ← cur_h;
  pdf_dest_fit, pdf_dest_fitb: do nothing;
  pdf_dest_fitr: set_rect_dimens(p, parent_box, x, y, pdf_width(p), pdf_height(p), pdf_depth(p), pdf_dest_margin);
endcases;
end;

procedure out_form(p : pointer);
begin pdf_end_text; pdf_print_ln("q");
if pdf_lookup_list(pdf_xform_list, pdf_xform_objnum(p)) = null then
  pdf_append_list(pdf_xform_objnum(p))(pdf_xform_list);
cur_v ← cur_v + obj_xform_depth(pdf_xform_objnum(p)); pdf_print("1 0 0 1 ");
pdf_print_bp(pdf_x(cur_h)); pdf_out(" "); pdf_print_bp(pdf_y(cur_v)); pdf_print_ln(" cm");
pdf_print("/Fm"); pdf_print_int(obj_info(pdf_xform_objnum(p))); pdf_print_resname_prefix;
pdf_print_ln("Do"); pdf_print_ln("Q");
end;

procedure out_image(p : pointer);
var image, groupref: integer; img_w, img_h: integer;
begin image ← obj_ximage_data(pdf_ximage_objnum(p));

```

```

if (image_rotate(image) = 90) ∨ (image_rotate(image) = 270) then
begin img_h ← image_width(image); img_w ← image_height(image);
end
else begin img_w ← image_width(image); img_h ← image_height(image);
end;
pdf_end_text; pdf_print_ln("q");
if pdf_lookup_list(pdf_ximage_list, pdf_ximage_objnum(p)) = null then
pdf_append_list(pdf_ximage_objnum(p))(pdf_ximage_list);
if ¬is_pdf_image(image) then
begin if is_png_image(image) then
begin groupref ← get_image_group_ref(image);
if (groupref > 0) ∧ (pdf_page_group_val = 0) then pdf_page_group_val ← groupref;
end;
pdf_print_real(ext_xn_over_d(pdf_width(p), ten_pow[6], one_hundred_bp), 4); pdf_print("000");
pdf_print_real(ext_xn_over_d(pdf_height(p) + pdf_depth(p), ten_pow[6], one_hundred_bp), 4);
pdf_out(" "); pdf_print_bp(pdf_x(cur_h)); pdf_out(" "); pdf_print_bp(pdf_y(cur_v));
end
else begin { for pdf images we generate the page group object number here }
groupref ← get_image_group_ref(image); { 0: no group, -1: to be generated; >0: already written }
if (groupref ≠ 0) ∧ (pdf_page_group_val = 0) then
begin if groupref = -1 then
begin pdf_page_group_val ← pdf_new_objnum; set_image_group_ref(image, pdf_page_group_val);
end
else { groupref > 0 }
pdf_page_group_val ← groupref;
end;
pdf_print_real(ext_xn_over_d(pdf_width(p), ten_pow[6], img_w), 6); pdf_print("000");
pdf_print_real(ext_xn_over_d(pdf_height(p) + pdf_depth(p), ten_pow[6], img_h), 6); pdf_out(" ");
pdf_print_bp(pdf_x(cur_h) - ext_xn_over_d(pdf_width(p), epdf_orig_x(image), img_w)); pdf_out(" ");
pdf_print_bp(pdf_y(cur_v) - ext_xn_over_d(pdf_height(p) + pdf_depth(p), epdf_orig_y(image), img_h));
end;
pdf_print_ln("cm"); pdf_print("/Im"); pdf_print_int(obj_info(pdf_ximage_objnum(p)));
pdf_print_resname_prefix; pdf_print_ln("Do"); pdf_print_ln("Q");
end;
function gap_amount(p : pointer; cur_pos : scaled): scaled;
{ find the gap between the position of the current snap node p and the nearest point on the grid }
var snap_unit, stretch_amount, shrink_amount: scaled; last_pos, next_pos, g, g2: scaled;
begin snap_unit ← width(snap_glue_ptr(p));
if stretch_order(snap_glue_ptr(p)) > normal then stretch_amount ← max_dimen
else stretch_amount ← stretch(snap_glue_ptr(p));
if shrink_order(snap_glue_ptr(p)) > normal then shrink_amount ← max_dimen
else shrink_amount ← shrink(snap_glue_ptr(p));
if subtype(p) = pdf_snappy_node then
last_pos ← pdf_snappy_refpos + snap_unit * ((cur_pos - pdf_snappy_refpos) div snap_unit)
else pdf_error("snapping", "invalid_parameter_value_for_gap_amount");
next_pos ← last_pos + snap_unit; @print_nl("snap_ref_pos="); print_scaled(pdf_snappy_refpos);
print_nl("snap_glue="); print_spec(snap_glue_ptr(p), 0); print_nl("gap_amount=");
print_scaled(snap_unit); print_nl("stretch_amount="); print_scaled(stretch_amount);
print_nl("shrink_amount="); print_scaled(shrink_amount); print_nl("last_point=");
print_scaled(last_pos); print_nl("cur_point="); print_scaled(cur_pos); print_nl("next_point=");
print_scaled(next_pos); @g ← max_dimen; g2 ← max_dimen; gap_amount ← 0;
if cur_pos - last_pos < shrink_amount then g ← cur_pos - last_pos;

```

```

if ( $next\_pos - cur\_pos < stretch\_amount$ ) then  $g2 \leftarrow next\_pos - cur\_pos$ ;
if ( $g = max\_dimen \wedge g2 = max\_dimen$ ) then return; { unable to snap }
if  $g2 \leq g$  then  $gap\_amount \leftarrow g2$  { skip forward }
else  $gap\_amount \leftarrow -g$ ; { skip backward }
end;

function get_vpos( $p, q, b : pointer$ ): pointer;
{ find the vertical position of node  $q$  in the output PDF page; this functions is called when the
current node is  $p$  and current position is  $cur\_v$  (global variable);  $b$  is the parent box; }

var  $tmp\_v: scaled; g\_order: glue\_ord$ ; { applicable order of infinity for glue }
 $g\_sign: normal \dots shrinking$ ; { selects type of glue }
 $glue\_temp: real$ ; { glue value before rounding }
 $cur\_glue: real$ ; { glue seen so far }
 $cur\_g: scaled$ ; { rounded equivalent of  $cur\_glue$  times the glue ratio }
 $this\_box: pointer$ ; { pointer to containing box }

begin  $tmp\_v \leftarrow cur\_v; this\_box \leftarrow b; cur\_g \leftarrow 0; cur\_glue \leftarrow float\_constant(0)$ ;
 $g\_order \leftarrow glue\_order(this\_box); g\_sign \leftarrow glue\_sign(this\_box)$ ;
while ( $p \neq q \wedge p \neq null$ ) do
begin if is_char_node( $p$ ) then confusion("get_vpos")
else begin case type( $p$ ) of
 $hlist\_node, vlist\_node, rule\_node: tmp\_v \leftarrow tmp\_v + height(p) + depth(p)$ ;
 $whatsit\_node: if (subtype(p) = pdf_reftxform_node) \vee (subtype(p) = pdf_refximage_node) then$ 
 $\quad tmp\_v \leftarrow tmp\_v + pdf_height(p) + pdf_depth(p)$ ;
 $glue\_node: begin \langle$  Move down without outputting leaders 1638  $\rangle$ ;
 $\quad tmp\_v \leftarrow tmp\_v + rule\_ht$ ;
end;
 $kern\_node: tmp\_v \leftarrow tmp\_v + width(p)$ ;
othercases do_nothing;
endcases;
end;
 $p \leftarrow link(p)$ ;
end;
get_vpos  $\leftarrow tmp\_v$ ;
end;

procedure do_snapy_comp( $p, b : pointer$ ); { do snapping compensation in vertical direction; search for the
next snap node and do the compensation if found }

var  $q: pointer; tmp\_v, g, g2: scaled$ ;
begin if  $\neg(is\_char\_node(p) \wedge (type(p) = whatsit\_node) \wedge (subtype(p) = pdf_snapy\_comp\_node))$  then
pdf_error("snapping", "invalid_parameter_value_for_do_snapy_comp");
 $q \leftarrow p$ ;
while ( $q \neq null$ ) do
begin if  $\neg(is\_char\_node(q) \wedge (type(q) = whatsit\_node) \wedge (subtype(q) = pdf_snapy\_node))$  then
begin  $tmp\_v \leftarrow get\_vpos(p, q, b)$ ; { get the position of  $q$  }
 $g \leftarrow gap\_amount(q, tmp\_v)$ ; { get the gap to the grid }
 $g2 \leftarrow round\_xn\_over\_d(g, snapy\_comp\_ratio(p), 1000)$ ; { adjustment for  $p$  }
@if print_nl("do_snapy_comp:@tmp_v= "); print_scaled(tmp_v);
print_nl("do_snapy_comp:@cur_v= "); print_scaled(cur_v); print_nl("do_snapy_comp:@g= ");
print_scaled(g); print_nl("do_snapy_comp:@g2= "); print_scaled(g2); @} cur_v  $\leftarrow cur\_v + g2$ ;
final_skip( $q$ )  $\leftarrow g - g2$ ; { adjustment for  $q$  }
if final_skip( $q$ ) = 0 then final_skip( $q$ )  $\leftarrow 1$ ;
{ use 1sp as the magic value to record that final_skip has been set here }
return;
end;

```

```

 $q \leftarrow link(q);$ 
end;
end;
procedure do_snapy( $p : pointer$ );
begin incr(count_do_snapy); @print_nl("do_snapy: count = "); print_int(count_do_snapy);
  print_nl("do_snapy: cur_v = "); print_scaled(cur_v); print_nl("do_snapy: final_skip = ");
  print_scaled(final_skip( $p$ )); @}
  if final_skip( $p$ )  $\neq 0$  then cur_v  $\leftarrow$  cur_v + final_skip( $p$ )
  else cur_v  $\leftarrow$  cur_v + gap_amount( $p$ , cur_v);
@print_nl("do_snapy: cur_v after snap = "); print_scaled(cur_v); @}
end;

```

1638. ⟨Move down without outputting leaders 1638⟩ ≡

```

begin g  $\leftarrow$  glue_ptr( $p$ ); rule_ht  $\leftarrow$  width(g) - cur_g;
if g_sign  $\neq$  normal then
  begin if g_sign = stretching then
    begin if stretch_order(g) = g_order then
      begin cur_glue  $\leftarrow$  cur_glue + stretch(g); vet_glue(float(glue_set(this_box)) * cur_glue);
        cur_g  $\leftarrow$  round(glue_temp);
      end;
    end
  else if shrink_order(g) = g_order then
    begin cur_glue  $\leftarrow$  cur_glue - shrink(g); vet_glue(float(glue_set(this_box)) * cur_glue);
      cur_g  $\leftarrow$  round(glue_temp);
    end;
  end;
  rule_ht  $\leftarrow$  rule_ht + cur_g;
end

```

This code is used in section 1637.

```
1639. ⟨Output the whatsit node  $p$  in  $\text{pdf\_vlist\_out}$  1639⟩ ≡
case  $\text{subtype}(p)$  of
   $\text{pdf\_literal\_node}, \text{pdf\_lateliteral\_node}$ :  $\text{pdf\_out\_literal}(p)$ ;
   $\text{pdf\_colorstack\_node}$ :  $\text{pdf\_out\_colorstack}(p)$ ;
   $\text{pdf\_setmatrix\_node}$ :  $\text{pdf\_out\_setmatrix}(p)$ ;
   $\text{pdf\_save\_node}$ :  $\text{pdf\_out\_save}(p)$ ;
   $\text{pdf\_restore\_node}$ :  $\text{pdf\_out\_restore}(p)$ ;
   $\text{pdf\_refobj\_node}$ :  $\text{pdf\_append\_list}(\text{pdf\_obj\_objnum}(p))(\text{pdf\_obj\_list})$ ;
   $\text{pdf\_refxform\_node}$ : ⟨Output a Form node in a vlist 1644⟩;
   $\text{pdf\_refximage\_node}$ : ⟨Output a Image node in a vlist 1643⟩;
   $\text{pdf\_annot\_node}$ :  $\text{do\_annot}(p, \text{this\_box}, \text{left\_edge}, \text{top\_edge} + \text{height}(\text{this\_box}))$ ;
   $\text{pdf\_start\_link\_node}$ :  $\text{pdf\_error}("ext4", "\text{pdfstartlink\_ended\_up\_in\_vlist}")$ ;
   $\text{pdf\_end\_link\_node}$ :  $\text{pdf\_error}("ext4", "\text{pdfendlink\_ended\_up\_in\_vlist}")$ ;
   $\text{pdf\_dest\_node}$ :  $\text{do\_dest}(p, \text{this\_box}, \text{left\_edge}, \text{top\_edge} + \text{height}(\text{this\_box}))$ ;
   $\text{pdf\_thread\_node}, \text{pdf\_start\_thread\_node}$ :  $\text{do\_thread}(p, \text{this\_box}, \text{left\_edge}, \text{top\_edge} + \text{height}(\text{this\_box}))$ ;
   $\text{pdf\_end\_thread\_node}$ :  $\text{end\_thread}$ ;
   $\text{pdf\_save\_pos\_node}$ : ⟨Save current position to  $\text{pdf\_last\_x\_pos}, \text{pdf\_last\_y\_pos}$  1641⟩;
   $\text{special\_node}, \text{latespecial\_node}$ :  $\text{pdf\_special}(p)$ ;
   $\text{pdf\_snap\_ref\_point\_node}$ : ⟨Save current position to  $\text{pdf\_snapx\_refpos}, \text{pdf\_snappy\_refpos}$  1642⟩;
   $\text{pdf\_snappy\_comp\_node}$ :  $\text{do\_snappy\_comp}(p, \text{this\_box})$ ;
   $\text{pdf\_snappy\_node}$ :  $\text{do\_snappy}(p)$ ;
   $\text{pdf\_interword\_space\_on\_node}$ :  $\text{gen\_faked\_interword\_space} \leftarrow \text{true}$ ;
   $\text{pdf\_interword\_space\_off\_node}$ :  $\text{gen\_faked\_interword\_space} \leftarrow \text{false}$ ;
   $\text{pdf\_fake\_space\_node}$ :  $\text{pdf\_insert\_fake\_space}$ ;
   $\text{pdf\_running\_link\_off\_node}$ :  $\text{gen\_running\_link} \leftarrow \text{false}$ ;
   $\text{pdf\_running\_link\_on\_node}$ :  $\text{gen\_running\_link} \leftarrow \text{true}$ ;
othercases  $\text{out\_what}(p)$ ;
endcases
```

This code is used in section 741.

1640. ⟨Global variables 13⟩ +≡
 is_shipping_page : boolean; { set to shipping_page when pdf_ship_out starts }

1641. ⟨Save current position to $\text{pdf_last_x_pos}, \text{pdf_last_y_pos}$ 1641⟩ ≡
begin $\text{pdf_last_x_pos} \leftarrow \text{cur_h}$;
if is_shipping_page **then** $\text{pdf_last_y_pos} \leftarrow \text{cur_page_height} - \text{cur_v}$
else $\text{pdf_last_y_pos} \leftarrow \text{pdf_xform_height} + \text{pdf_xform_depth} - \text{cur_v}$;
end

This code is used in sections 1639 and 1645.

1642. ⟨Save current position to $\text{pdf_snapx_refpos}, \text{pdf_snappy_refpos}$ 1642⟩ ≡
begin $\text{pdf_snapx_refpos} \leftarrow \text{cur_h}$; $\text{pdf_snappy_refpos} \leftarrow \text{cur_v}$;
end

This code is used in sections 1639 and 1645.

1643. ⟨Output a Image node in a vlist 1643⟩ ≡
begin $\text{cur_v} \leftarrow \text{cur_v} + \text{pdf_height}(p) + \text{pdf_depth}(p)$; $\text{save_v} \leftarrow \text{cur_v}$; $\text{cur_h} \leftarrow \text{left_edge}$; $\text{out_image}(p)$;
 $\text{cur_v} \leftarrow \text{save_v}$; $\text{cur_h} \leftarrow \text{left_edge}$;
end

This code is used in section 1639.

1644. \langle Output a Form node in a vlist 1644 $\rangle \equiv$

```
begin cur_v ← cur_v + pdf_height(p); save_v ← cur_v; cur_h ← left_edge; out_form(p);
cur_v ← save_v + pdf_depth(p); cur_h ← left_edge;
end
```

This code is used in section 1639.

1645. \langle Output the whatsit node p in pdf_hlist_out 1645 $\rangle \equiv$

```
case subtype(p) of
pdf_literal_node, pdf_lateliteral_node: pdf_out_literal(p);
pdf_colorstack_node: pdf_out_colorstack(p);
pdf_setmatrix_node: pdf_out_setmatrix(p);
pdf_save_node: pdf_out_save(p);
pdf_restore_node: pdf_out_restore(p);
pdf_refobj_node: pdf_append_list(pdf_obj_objnum(p))(pdf_obj_list);
pdf_refxform_node: ⟨ Output a Form node in a hlist 1647 ⟩;
pdf_refximage_node: ⟨ Output a Image node in a hlist 1646 ⟩;
pdf_annot_node: do_annot(p, this_box, left_edge, base_line);
pdf_start_link_node: do_link(p, this_box, left_edge, base_line);
pdf_end_link_node: end_link;
pdf_dest_node: do_dest(p, this_box, left_edge, base_line);
pdf_thread_node: do_thread(p, this_box, left_edge, base_line);
pdf_start_thread_node: pdf_error("ext4", "\pdfstartthreadendedupinlist");
pdf_end_thread_node: pdf_error("ext4", "\pdfendthreadendedupinlist");
pdf_save_pos_node: ⟨ Save current position to pdf_last_x_pos, pdf_last_y_pos 1641 ⟩;
special_node, latespecial_node: pdf_special(p);
pdf_snap_ref_point_node: ⟨ Save current position to pdf_snapx_refpos, pdf_snapy_refpos 1642 ⟩;
pdf_snapy_comp_node, pdf_snapy_node: do_nothing; { snapy nodes do nothing in hlist }
pdf_interword_space_on_node: gen_faked_interword_space ← true;
pdf_interword_space_off_node: gen_faked_interword_space ← false;
pdf_fake_space_node: pdf_insert_fake_space;
pdf_running_link_off_node: gen_running_link ← false;
pdf_running_link_on_node: gen_running_link ← true;
othercases out_what(p);
endcases
```

This code is used in section 732.

1646. \langle Output a Image node in a hlist 1646 $\rangle \equiv$

```
begin cur_v ← base_line + pdf_depth(p); edge ← cur_h; out_image(p); cur_h ← edge + pdf_width(p);
cur_v ← base_line;
end
```

This code is used in section 1645.

1647. \langle Output a Form node in a hlist 1647 $\rangle \equiv$

```
begin cur_v ← base_line; edge ← cur_h; out_form(p); cur_h ← edge + pdf_width(p); cur_v ← base_line;
end
```

This code is used in section 1645.

1648. 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  $\varepsilon$ -TeX, if requested 1648) ≡
  init if (buffer[loc] = "*") ∧ (format.ident = " $\sqcup$ (INITEX)") then
    begin no_new_control_sequence ← false; {Generate all  $\varepsilon$ -TeX primitives 1649}
      incr(loc); eTeX_mode ← 1; {enter extended mode}
      {Initialize variables for  $\varepsilon$ -TeX extended mode 1813}
    end;
  tini
  if  $\neg$ no_new_control_sequence then {just entered extended mode ?}
    no_new_control_sequence ← true else
```

This code is used in section 1517.

1649. 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  $\varepsilon$ -TeX state variable}
define eTeX_version_code = eTeX_int {code for \eTeXversion}
(Generate all  $\varepsilon$ -TeX primitives 1649) ≡
  primitive("lastnodetype", last_item, last_node_type_code);
  primitive("eTeXversion", last_item, eTeX_version_code);
  primitive("eTeXrevision", convert, eTeX_revision_code);
```

See also sections 1657, 1663, 1666, 1669, 1672, 1675, 1684, 1686, 1689, 1692, 1697, 1701, 1747, 1759, 1762, 1770, 1778, 1801, 1805, 1809, 1861, and 1864.

This code is used in section 1648.

1650. {Cases of `last_item` for `print_cmd_chr` 1650} ≡
 `last_node_type_code`: `print_esc("lastnodetype")`;
 `eTeX_version_code`: `print_esc("eTeXversion")`;

See also sections 1664, 1667, 1670, 1673, 1779, 1802, and 1806.

This code is used in section 443.

1651. {Cases for fetching an integer value 1651} ≡
 `eTeX_version_code`: `cur_val ← eTeX_version`;

See also sections 1665, 1668, and 1803.

This code is used in section 450.

1652. define `eTeX_ex` ≡ (`eTeX_mode` = 1) {is this extended mode?}

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

1653. \langle Initialize table entries (done by INITEX only) 182 $\rangle +\equiv$
 $eTeX_mode \leftarrow 0;$ { initially we are in compatibility mode }
 \langle Initialize variables for ε -TEX compatibility mode 1812 \rangle

1654. \langle Dump the ε -TEX state 1654 $\rangle \equiv$
 $dump_int(eTeX_mode);$
 $\text{for } j \leftarrow 0 \text{ to } eTeX_states - 1 \text{ do } eTeX_state(j) \leftarrow 0; \{ \text{ disable all enhancements} \}$

See also section 1758.

This code is used in section 1485.

1655. \langle Undump the ε -TEX state 1655 $\rangle \equiv$
 $undump(0)(1)(eTeX_mode);$
 $\text{if } eTeX_ex \text{ then}$
 $\quad \text{begin } \langle \text{Initialize variables for } \varepsilon\text{-TEX extended mode 1813} \rangle$
 $\quad \text{end}$
 $\text{else begin } \langle \text{Initialize variables for } \varepsilon\text{-TEX compatibility mode 1812} \rangle$
 $\quad \text{end};$

This code is used in section 1486.

1656. 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.

\langle Declare ε -TEX procedures for use by main_control 1656 $\rangle \equiv$
function $eTeX_enabled(b : \text{boolean}; j : \text{quarterword}; k : \text{halfword}) : \text{boolean};$
 $\quad \text{begin if } \neg b \text{ then}$
 $\quad \quad \text{begin } print_err("Improper"); print_cmd_chr(j, k);$
 $\quad \quad help1("Sorry, this optional \varepsilon\text{-TEX feature has been disabled."); error;$
 $\quad \quad \text{end};$
 $\quad eTeX_enabled \leftarrow b;$
 $\quad \text{end};$

See also sections 1679 and 1695.

This code is used in section 991.

1657. First we implement the additional ε -TEX parameters in the table of equivalents.

\langle Generate all ε -TEX primitives 1649 $\rangle +\equiv$
 $\text{primitive}("everyeof", assign_toks, every_eof_loc);$
 $\text{primitive}("tracingassigns", assign_int, int_base + tracing_assigns_code);$
 $\text{primitive}("tracinggroups", assign_int, int_base + tracing_groups_code);$
 $\text{primitive}("tracingifs", assign_int, int_base + tracing_ifs_code);$
 $\text{primitive}("tracingscantokens", assign_int, int_base + tracing_scan_tokens_code);$
 $\text{primitive}("tracingnesting", assign_int, int_base + tracing_nesting_code);$
 $\text{primitive}("predisplaydirection", assign_int, int_base + pre_display_direction_code);$
 $\text{primitive}("lastlinefit", assign_int, int_base + last_line_fit_code);$
 $\text{primitive}("savingvdiscards", assign_int, int_base + saving_vdiscards_code);$
 $\text{primitive}("savinghyphcodes", assign_int, int_base + saving_hyph_codes_code);$

1658. **define** $every_eof \equiv equiv(every_eof_loc)$

\langle Cases of $assign_toks$ for $print_cmd_chr$ 1658 $\rangle \equiv$
 $every_eof_loc : print_esc("everyeof");$

This code is used in section 249.

1659. \langle Cases for *print_param* 1659 $\rangle \equiv$
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 1700.

This code is used in section 255.

1660. In order to handle `\everyeof` we need an array *eof_seen* of boolean variables.

\langle Global variables 13 $\rangle +\equiv$
eof_seen: **array** [1 .. *max_in_open*] **of** boolean; { has eof been seen? }

1661. The *print_group* procedure prints the current level of grouping and the name corresponding to *cur_group*.

```
< Declare  $\varepsilon$ -TEX procedures for tracing and input 306 > +≡
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;
```

1662. 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  $\varepsilon$ -TEX procedures for tracing and input 306 > +≡
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
```

1663. 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  $\varepsilon$ -TeX primitives 1649⟩ +≡
primitive("currentgrouplevel", last_item, current_group_level_code);
primitive("currentgrouptype", last_item, current_group_type_code);
```

1664. ⟨Cases of `last_item` for `print_cmd_chr` 1650⟩ +≡
`current_group_level_code`: `print_esc("currentgrouplevel")`;
`current_group_type_code`: `print_esc("currentgrouptype")`;

1665. ⟨Cases for fetching an integer value 1651⟩ +≡
`current_group_level_code`: `cur_val ← cur_level - level_one`;
`current_group_type_code`: `cur_val ← cur_group`;

1666. 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  $\varepsilon$ -TeX primitives 1649⟩ +≡
primitive("currentiflevel", last_item, current_if_level_code);
primitive("currentiftype", last_item, current_if_type_code);
primitive("currentifbranch", last_item, current_if_branch_code);
```

1667. ⟨Cases of `last_item` for `print_cmd_chr` 1650⟩ +≡
`current_if_level_code`: `print_esc("currentiflevel")`;
`current_if_type_code`: `print_esc("currentiftype")`;
`current_if_branch_code`: `print_esc("currentifbranch")`;

1668. ⟨Cases for fetching an integer value 1651⟩ +≡
`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;`

1669. 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  $\varepsilon$ -TEX primitives 1649⟩ +≡
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);
```

1670. ⟨Cases of `last_item` for `print_cmd_chr` 1650⟩ +≡
`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");`

1671. ⟨Cases for fetching a dimension value 1671⟩ ≡
`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 1674 and 1804.

This code is used in section 450.

1672. The `\parshapedimen`, `\parshapeindent`, and `\parshapelen` commands return the indent and length parameters of the current `\parshape` specification.

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

⟨Generate all  $\varepsilon$ -TEX primitives 1649⟩ +≡
primitive("parshapelen", last_item, par_shape_length_code);
primitive("parshapeindent", last_item, par_shape_indent_code);
primitive("parshapedimen", last_item, par_shape_dimen_code);
```

1673. ⟨Cases of `last_item` for `print_cmd_chr` 1650⟩ +≡
`par_shape_length_code: print_esc("parshapelen");`
`par_shape_indent_code: print_esc("parshapeindent");`
`par_shape_dimen_code: print_esc("parshapedimen");`

1674. \langle Cases for fetching a dimension value 1671 $\rangle +\equiv$

```
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;
```

1675. The \showgroups command displays all currently active grouping levels.

```
define show_groups = 4 { \showgroups }
⟨ Generate all  $\varepsilon$ -TEX primitives 1649 ⟩ +≡
primitive("showgroups", xray, show_groups);
```

1676. \langle Cases of *xray* for *print_cmd_chr* 1676 $\rangle \equiv$

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

See also sections 1685 and 1690.

This code is used in section 1470.

1677. \langle Cases for *show_whatever* 1677 $\rangle \equiv$

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

See also section 1691.

This code is used in section 1471.

1678. \langle Types in the outer block 18 $\rangle +\equiv$

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

1679. 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  $\varepsilon$ -TeX procedures for use by main_control 1656 > +≡
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("### $\downarrow$ "); 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(" $\downarrow$ ");
  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 $\downarrow$ entry")
      else print_esc("cr");
      if p ≥ a then p ← p − a;
      a ← 0; goto found;
      end;
    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;
    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 1681 ⟩;
found1: print_esc(s); ⟨ Show the box packaging info 1680 ⟩;
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;

```

1680. ⟨ Show the box packaging info 1680 ⟩ ≡

```

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 1679.

1681. \langle Show the box context 1681 $\rangle \equiv$

```

i  $\leftarrow$  saved(-4);
if i  $\neq$  0 then
  if i < box_flag then
    begin if abs(nest[p].mode_field) = vmode then j  $\leftarrow$  hmove
    else j  $\leftarrow$  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  $\geq$  global_box_flag then
      begin print_esc("global"); i  $\leftarrow$  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 1679.

1682. The *scan_general_text* procedure is much like *scan_toks(false, false)*, but will be invoked via *expand*, i.e., recursively.

\langle Declare ε -TEX procedures for scanning 1682 $\rangle \equiv$

```

procedure scan_general_text; forward;

```

See also sections 1772, 1781, and 1786.

This code is used in section 435.

1683. 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.)

\langle Declare ε -TeX procedures for token lists 1683 $\rangle \equiv$

```

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 1753.

This code is used in section 490.

1684. The *\showtokens* command displays a token list.

```

define show_tokens = 5 { \showtokens , must be odd! }

⟨ Generate all  $\varepsilon$ -TeX primitives 1649 ⟩ +≡
primitive("showtokens", xray, show_tokens);
```

1685. ⟨ Cases of *xray* for *print_cmd_chr* 1676 ⟩ +≡
show_tokens: *print_esc("showtokens")*;

1686. 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  $\varepsilon$ -TeX primitives 1649 ⟩ +≡
primitive("unexpanded", the, 1);
primitive("detokenize", the, show_tokens);
```

1687. ⟨ Cases of *the* for *print_cmd_chr* 1687 ⟩ ≡
else if *chr_code* = 1 then *print_esc("unexpanded")*
 else *print_esc("detokenize")*

This code is used in section 288.

1688. \langle Handle \unexpanded or \detokenize and return 1688 $\rangle \equiv$

```

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 491.

1689. The \showifs command displays all currently active conditionals.

```

define show_ifs = 6 { \showifs }
⟨Generate all  $\varepsilon$ -TEX primitives 1649⟩ +≡
primitive("showifs", xray, show_ifs);

```

1690. \langle Cases of xray for print_cmd_chr 1676 $\rangle +\equiv$

```

show_ifs: print_esc("showifs");

```

1691.

```

define print_if_line(♯) ≡
  if ♯ ≠ 0 then
    begin print("↳ entered↳on↳line↳"); print_int(♯);
  end

⟨Cases for show_whatever 1677⟩ +≡
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;

```

1692. The \interactionmode primitive allows to query and set the interaction mode.

```

⟨Generate all  $\varepsilon$ -TEX primitives 1649⟩ +≡
primitive("interactionmode", set_page_int, 2);

```

1693. \langle Cases of set_page_int for print_cmd_chr 1693 $\rangle \equiv$

```

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

```

This code is used in section 443.

1694. \langle Cases for 'Fetch the dead_cycles or the insert_penalties' 1694 $\rangle \equiv$

```

else if m = 2 then cur_val ← interaction

```

This code is used in section 445.

1695. \langle Declare ε -TEX procedures for use by *main_control* 1656 $\rangle + \equiv$
procedure *new_interaction*; *forward*;

1696. \langle Cases for *alter_integer* 1696 $\rangle \equiv$
else if *c* = 2 **then**
 begin if (*cur_val* < *batch_mode*) \vee (*cur_val* > *error_stop_mode*) **then**
 begin *print_err*("Bad₁interaction₁mode");
 help2("Modes₁are₁0=batch,₁1=nonstop,₁2=scroll,₁and")
 ("3=errorstop.₁Proceed,₁and₁I'll₁ignore₁this₁case."); *int_error*(*cur_val*);
 end
 else begin *cur_chr* \leftarrow *cur_val*; *new_interaction*;
 end;
 end

This code is used in section 1424.

1697. The *middle* feature of ε -TEX allows one ore several \middle delimiters to appear between \left and \right.

\langle Generate all ε -TEX primitives 1649 $\rangle + \equiv$
primitive("middle", *left_right*, *middle_noad*);

1698. \langle Cases of *left_right* for *print_cmd_chr* 1698 $\rangle \equiv$
else if *chr_code* = *middle_noad* **then** *print_esc*("middle")

This code is used in section 1367.

1699. In constructions such as

```
\hbox to \hsize{
  \hskip Opt 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 behavior 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 1699) ≡
if (((g_sign = stretching) ∧ (stretch_order(g) = g_order)) ∨ ((g_sign = shrinking) ∧ (shrink_order(g) =
g_order))) then
begin fast_delete_glue_ref(g);
if subtype(p) < a_leaders then
begin type(p) ← kern_node; width(p) ← rule_wd;
end
else begin g ← get_node(glue_spec_size);
stretch_order(g) ← filll + 1; shrink_order(g) ← filll + 1; { will never match }
width(g) ← rule_wd; stretch(g) ← 0; shrink(g) ← 0; glue_ptr(p) ← g;
end;
end
```

This code is used in sections 653, 735, and 1726.

1700. 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-X_ET* as presented by Donald E. Knuth and Pierre MacKay in *TUGboat* 8, 14–25, 1987.

In order to avoid confusion with *TeX-X_ET* the present implementation of mixed direction typesetting is called *TeX--X_ET*. It differs from *TeX-X_ET* 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--X_ET* 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--X_ET* 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 TeXX--XET enabled? }

⟨ Cases for print_param 1659 ⟩ +≡
eTeX_state_code + TeXXeT_code: print_esc("TeXXeTstate");
```

1701. ⟨ Generate all ε -TeX primitives 1649 ⟩ +≡
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*);

1702. ⟨ Cases of *valign* for print_cmd_chr 1702 ⟩ ≡
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 288.

1703. ⟨ Cases of *main_control* for *hmode* + *valign* 1703 ⟩ ≡
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 1308.

1704. An hbox with subtype dlist will never be reversed, even when embedded in right-to-left text.

\langle Display if this box is never to be reversed 1704 $\rangle \equiv$
if (*type*(*p*) = *hlist_node*) \wedge (*box_lr*(*p*) = *dlist*) **then** *print*(" $\text{\texttt{,}\text{\texttt{,}}}$ display")

This code is used in section 202.

1705. 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  $\leftarrow$  get_avail; info(temp_ptr)  $\leftarrow$  #; link(temp_ptr)  $\leftarrow$  LR_ptr;
  LR_ptr  $\leftarrow$  temp_ptr;
  end
define push_LR(#)  $\equiv$  put_LR(end_LR_type(#))
define pop_LR  $\equiv$ 
  begin temp_ptr  $\leftarrow$  LR_ptr; LR_ptr  $\leftarrow$  link(temp_ptr); free_avail(temp_ptr);
  end
 $\langle$  Global variables 13  $\rangle +\equiv$ 
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 }
```

1706. \langle Set initial values of key variables 21 $\rangle +\equiv$

LR_ptr \leftarrow null; *LR_problems* \leftarrow 0; *cur_dir* \leftarrow left_to_right;

1707. \langle Insert LR nodes at the beginning of the current line and adjust the LR stack based on LR nodes in this line 1707 $\rangle \equiv$

```
begin q  $\leftarrow$  link(temp_head);
if LR_ptr  $\neq$  null then
  begin temp_ptr  $\leftarrow$  LR_ptr; r  $\leftarrow$  q;
  repeat s  $\leftarrow$  new_math(0, begin_LR_type(info(temp_ptr))); link(s)  $\leftarrow$  r; r  $\leftarrow$  s;
    temp_ptr  $\leftarrow$  link(temp_ptr);
  until temp_ptr = null;
  link(temp_head)  $\leftarrow$  r;
  end;
while q  $\neq$  cur_break(cur_p) do
  begin if  $\neg$ is_char_node(q) then
    if type(q) = math_node then  $\langle$  Adjust the LR stack for the post_line_break routine 1708  $\rangle$ ;
    q  $\leftarrow$  link(q);
  end;
end
```

This code is used in section 1056.

1708. \langle Adjust the LR stack for the *post_line_break* routine 1708 $\rangle \equiv$

```
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 1055, 1057, and 1707.

1709. We use the fact that q now points to the node with `\rightskip` glue.

\langle Insert LR nodes at the end of the current line 1709 $\rangle \equiv$

```

if  $LR\_ptr \neq null$  then
  begin  $s \leftarrow temp\_head$ ;  $r \leftarrow link(s)$ ;
  while  $r \neq q$  do
    begin  $s \leftarrow r$ ;  $r \leftarrow link(s)$ ;
    end;
   $r \leftarrow LR\_ptr$ ;
  while  $r \neq null$  do
    begin  $temp\_ptr \leftarrow new\_math(0, info(r))$ ;  $link(s) \leftarrow temp\_ptr$ ;  $s \leftarrow temp\_ptr$ ;  $r \leftarrow link(r)$ ;
    end;
   $link(s) \leftarrow q$ ;
end

```

This code is used in section 1056.

1710. \langle Initialize the LR stack 1710 $\rangle \equiv$

```
put_LR(before) { this will never match }
```

This code is used in sections 823, 1714, and 1734.

1711. \langle Adjust the LR stack for the `hpack` routine 1711 $\rangle \equiv$

```

if  $end\_LR(p)$  then
  if  $info(LR\_ptr) = end\_LR\_type(p)$  then  $pop\_LR$ 
  else begin  $incr(LR\_problems)$ ;  $type(p) \leftarrow kern\_node$ ;  $subtype(p) \leftarrow explicit$ ;
  end
else  $push\_LR(p)$ 

```

This code is used in section 825.

1712. \langle Check for LR anomalies at the end of `hpack` 1712 $\rangle \equiv$

```

begin if  $info(LR\_ptr) \neq before$  then
  begin while  $link(q) \neq null$  do  $q \leftarrow link(q)$ ;
  repeat  $temp\_ptr \leftarrow q$ ;  $q \leftarrow new\_math(0, info(LR\_ptr))$ ;  $link(temp\_ptr) \leftarrow q$ ;
   $LR\_problems \leftarrow LR\_problems + 10000$ ;  $pop\_LR$ ;
  until  $info(LR\_ptr) = before$ ;
  end;
if  $LR\_problems > 0$  then
  begin ⟨ Report LR problems 1713 ⟩;
  goto common_ending;
  end;
 $pop\_LR$ ;
if  $LR\_ptr \neq null$  then  $confusion("LR1")$ ;
end

```

This code is used in section 823.

1713. \langle Report LR problems 1713 $\rangle \equiv$

```

begin  $print\_ln$ ;  $print\_nl(" \backslash endL \cup or \cup \backslash endR \cup problem \cup ")$ ;
 $print\_int(LR\_problems \text{ div } 10000)$ ;  $print(" \cup missing \cup ")$ ;
 $print\_int(LR\_problems \text{ mod } 10000)$ ;  $print(" \cup extra \cup ")$ ;
 $LR\_problems \leftarrow 0$ ;
end

```

This code is used in sections 1712 and 1730.

1714. \langle Initialize *hlist_out* for mixed direction typesetting 1714 $\rangle \equiv$

```

if eTeX_ex then
  begin < Initialize the LR stack 1710 >;
  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 1721);
  end

```

This code is used in sections 647 and 729.

1715. \langle Finish *hlist_out* for mixed direction typesetting 1715 $\rangle \equiv$

```

if eTeX_ex then
  begin < Check for LR anomalies at the end of hlist_out 1718 >;
  if box_lr(this_box) = dlist then cur_dir ← right_to_left;
  end

```

This code is used in sections 647 and 729.

1716. \langle Handle a math node in *hlist_out* 1716 $\rangle \equiv$

```

begin if eTeX_ex then < Adjust the LR stack for the hlist_out routine; if necessary reverse an hlist
  segment and goto reswitch 1717 >;
  cur_h ← cur_h + width(p);
end

```

This code is used in sections 650 and 732.

1717. Breaking a paragraph into lines while TeX-- $\mathbb{X}\mathbb{T}$ is disabled may result in lines with 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 1717 > ≡
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 1722 >;
  end;
type(p) ← kern_node;
end

```

This code is used in section 1716.

1718. \langle Check for LR anomalies at the end of *hlist_out* 1718 $\rangle \equiv$

```

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 1715.

1719. `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 1615 > +≡`

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;

1720. `< Cases of hlist_out that arise in mixed direction text only 1720 > ≡`
`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 sections 650 and 732.

1721. 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 1721 > ≡`
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 1714.

1722. 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 1722 > ≡`
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 1717.

1723. 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 1615 > +≡
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 ≠ 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 1724>;
  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;
```

1724. <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 1724> ≡

```
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 not is_char_node(p)
else <Move the non-char_node p to the new list 1725>
```

This code is used in section 1723.

1725. <Move the non-*char_node* *p* to the new list 1725> ≡

```
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 1726 >
  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 1724.

1726. Here we compute the effective width of a glue node as in *hlist_out*.

```
(Cases of reverse that need special treatment 1726) ≡
glue_node: begin round_glue; (Handle a glue node for mixed direction typesetting 1699);
end;
```

See also sections 1727 and 1728.

This code is used in section 1725.

1727. A ligature node is replaced by a char node.

```
(Cases of reverse that need special treatment 1726) +≡
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;
```

1728. 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 1726) +≡
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 1729};
    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;
```

1729. 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 1729} ≡
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 1728.

1730. \langle Check for LR anomalies at the end of *ship_out* 1730 $\rangle \equiv$

```

begin if LR_problems > 0 then
  begin (Report LR problems 1713);
  print_char(")");
  print_ln;
end;
if (LR_ptr ≠ null) ∨ (cur_dir ≠ left_to_right) then confusion("LR3");
end

```

This code is used in sections 666 and 750.

1731. 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.

\langle Set the value of *x* to the text direction before the display 1731 $\rangle \equiv$

```

if LR_save = null then x ← 0
else if info(LR_save) ≥ R_code then x ← -1 else x ← 1

```

This code is used in sections 1732 and 1734.

1732. \langle Prepare for display after an empty paragraph 1732 $\rangle \equiv$

```

begin pop_nest;  $\langle$  Set the value of x to the text direction before the display 1731  $\rangle$ ;
end

```

This code is used in section 1323.

1733. 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 1733 ⟩ ≡
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 1604 ⟩;
    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 1738.

This code is used in section 1316.

1734. 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 1734 ⟩ ≡
  if  $eTeX\_ex$  then ⟨ Let  $j$  be the prototype box for the display 1740 ⟩;
   $v \leftarrow shift\_amount(just\_box)$ ; ⟨ Set the value of  $x$  to the text direction before the display 1731 ⟩;
  if  $x \geq 0$  then
    begin  $p \leftarrow list\_ptr(just\_box)$ ;  $link(temp\_head) \leftarrow null$ ;
    end
  else begin  $v \leftarrow -v - width(just\_box)$ ;  $p \leftarrow new\_math(0, begin\_L\_code)$ ;  $link(temp\_head) \leftarrow p$ ;
    just_copy(list_ptr(just_box), p, new_math(0, end_L_code));  $cur\_dir \leftarrow right\_to\_left$ ;
    end;
   $v \leftarrow v + 2 * quad(cur\_font)$ ;
  if  $TeXXeT\_en$  then ⟨ Initialize the LR stack 1710 ⟩
```

This code is used in section 1324.

1735. ⟨ Finish the natural width computation 1735 ⟩ ≡

```
if  $TeXXeT\_en$  then
  begin while  $LR\_ptr \neq null$  do  $pop\_LR$ ;
  if  $LR\_problems \neq 0$  then
    begin  $w \leftarrow max\_dimen$ ;  $LR\_problems \leftarrow 0$ ;
    end;
  end;
   $cur\_dir \leftarrow left\_to\_right$ ; flush_node_list(link(temp_head))
```

This code is used in section 1324.

1736. 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 $\text{\TeX--}\mathbf{X}\mathbf{\text{\TeX}}$ is disabled, then we set $w \leftarrow max_dimen$.

```
⟨ Cases of ‘Let  $d$  be the natural width’ that need special treatment 1736 ⟩ ≡
math_node: begin  $d \leftarrow width(p)$ ;
  if  $TeXXeT\_en$  then ⟨ Adjust the LR stack for the init_math routine 1737 ⟩
  else if subtype( $p$ )  $\geq L\_code$  then
    begin  $w \leftarrow max\_dimen$ ; goto done;
    end;
  end;
edge_node: begin  $d \leftarrow width(p)$ ;  $cur\_dir \leftarrow subtype(p)$ ;
  end;
```

This code is used in section 1325.

1737. \langle Adjust the LR stack for the *init_math* routine 1737 $\rangle \equiv$

```

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  $\leftarrow$  max_dimen; goto done;
    end
  end
else begin push_LR(p);
  if LR_dir(p)  $\neq$  cur_dir then
    begin just_reverse(p); p  $\leftarrow$  temp_head;
    end;
  end
end

```

This code is used in section 1736.

1738. \langle Declare subprocedures for *init_math* 1733 $\rangle +\equiv$

```

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  $\leftarrow$  min_halfword; n  $\leftarrow$  min_halfword;
if link(temp_head) = null then
  begin just_copy(link(p), temp_head, null); q  $\leftarrow$  link(temp_head);
  end
else begin q  $\leftarrow$  link(p); link(p)  $\leftarrow$  null; flush_node_list(link(temp_head));
  end;
t  $\leftarrow$  new_edge(cur_dir, 0); l  $\leftarrow$  t; cur_dir  $\leftarrow$  reflected;
while q  $\neq$  null do
  if is_char_node(q) then
    repeat p  $\leftarrow$  q; q  $\leftarrow$  link(p); link(p)  $\leftarrow$  l; l  $\leftarrow$  p;
    until  $\neg$ is_char_node(q)
  else begin p  $\leftarrow$  q; q  $\leftarrow$  link(p);
    if type(p) = math_node then { Adjust the LR stack for the just_reverse routine 1739 };
      link(p)  $\leftarrow$  l; l  $\leftarrow$  p;
    end;
  goto done;
found: width(t)  $\leftarrow$  width(p); link(t)  $\leftarrow$  q; free_node(p, small_node_size);
done: link(temp_head)  $\leftarrow$  l;
end;

```

1739. \langle Adjust the LR stack for the *just_reverse* routine 1739 $\rangle \equiv$

```

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 if m > min_halfword then decr(m) else goto found;
      type(p) ← kern_node;
      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

```

This code is used in section 1738.

1740. 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*.

\langle Let *j* be the prototype box for the display 1740 $\rangle \equiv$

```

begin if right_skip = zero_glue then j ← new_kern(0)
else j ← new_param_glue(right_skip_code);
if left_skip = zero_glue then p ← new_kern(0)
else p ← new_param_glue(left_skip_code);
link(p) ← j; j ← new_null_box; width(j) ← width(just_box); shift_amount(j) ← shift_amount(just_box);
list_ptr(j) ← p; glue_order(j) ← glue_order(just_box); glue_sign(j) ← glue_sign(just_box);
glue_set(j) ← glue_set(just_box);
end

```

This code is used in section 1734.

1741. At the end of a displayed equation we retrieve the prototype box.

\langle Local variables for finishing a displayed formula 1376 $\rangle +\equiv$
j: pointer; { prototype box }

1742. \langle Retrieve the prototype box 1742 $\rangle \equiv$

if mode = mmode then *j* ← LR_box

This code is used in sections 1372 and 1372.

1743. \langle Flush the prototype box 1743 $\rangle \equiv$

flush_node_list(*j*)

This code is used in section 1377.

1744. 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 1744 > ≡
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 1745 };
  { Package the display line 1746 };
  end;
append_to_vlist(b);
end;
```

This code is used in section 1372.

1745. 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 1745 > ≡
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 1744.

1746. 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 ≠ 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 1746 ⟩ ≡
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 1744.

1747. The `scan_tokens` feature of ε -TEX defines the `\scantokens` primitive.

```
⟨ Generate all  $\varepsilon$ -TEX primitives 1649 ⟩ +≡
primitive("scantokens", input, 2);
```

1748. ⟨ Cases of `input` for `print_cmd_chr` 1748 ⟩ ≡
`else if chr_code = 2 then print_esc("scantokens")`

This code is used in section 403.

1749. ⟨ Cases for `input` 1749 ⟩ ≡
`else if cur_chr = 2 then pseudo_start`

This code is used in section 404.

1750. 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 }
```

1751. <Set initial values of key variables 21> +≡
pseudo_files \leftarrow null;

1752. The *pseudo_start* procedure initiates reading from a pseudo file.

```
<Declare  $\varepsilon$ -TEX procedures for expanding 1752> ≡
procedure pseudo_start; forward;
```

See also sections 1810, 1815, and 1819.

This code is used in section 388.

1753. <Declare ε -TEX procedures for token lists 1683> +≡
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  $\leftarrow$  selector; selector  $\leftarrow$  new_string; token_show(temp_head);
selector  $\leftarrow$  old_setting; flush_list(link(temp_head)); str_room(1); s  $\leftarrow$  make_string;
{ Convert string s into a new pseudo file 1754 };
flush_string; { Initiate input from new pseudo file 1755 };
end;
```

1754. \langle Convert string s into a new pseudo file 1754 $\rangle \equiv$

```

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 1753.

1755. \langle Initiate input from new pseudo file 1755 $\rangle \equiv$

```

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 1753.

1756. Here we read a line from the current pseudo file into *buffer*.

```
< Declare  $\varepsilon$ -TEX procedures for tracing and input 306 > +≡
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;
```

1757. When we are done with a pseudo file we ‘close’ it.

```
< Declare  $\varepsilon$ -TEX procedures for tracing and input 306 > +≡
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;
```

1758. < Dump the ε -TEX state 1654 > +≡
while pseudo_files ≠ null **do** pseudo_close; { flush pseudo files }

1759. < Generate all ε -TEX primitives 1649 > +≡
primitive("readline", read_to_cs, 1);

1760. < Cases of read for print_cmd_chr 1760 > ≡
else print_esc("readline")

This code is used in section 288.

1761. \langle Handle \readline and goto done 1761 $\rangle \equiv$

```

if  $j = 1$  then
  begin while  $loc \leq limit$  do { current line not yet finished }
    begin  $cur\_chr \leftarrow buffer[loc]$ ;  $incr(loc)$ ;
      if  $cur\_chr = " "$  then  $cur\_tok \leftarrow space\_token$  else  $cur\_tok \leftarrow cur\_chr + other\_token$ ;
      store_new_token( $cur\_tok$ );
    end;
  goto done;
end

```

This code is used in section 509.

1762. Here we define the additional conditionals of ε -TEX as well as the \unless prefix.

```

define if_def_code = 17 { '\ifdef' }
define if_cs_code = 18 { '\ifcsname' }
define if_font_char_code = 19 { '\iffontchar' }
define if_in_csnname_code = 20 { '\ifincsname' }
define if_pdfabs_num_code = 22 { '\ifpdfabsnum' }
{ 21 = if-pdfprimitive }
define if_pdfabs_dim_code = 23 { '\ifpdfabsdim' }

⟨Generate all  $\varepsilon$ -TEX primitives 1649⟩ +≡
primitive("unless", expand_after, 1);
primitive("ifdef", if_test, if_def_code); primitive("ifcsname", if_test, if_cs_code);
primitive("iffontchar", if_test, if_font_char_code); primitive("ifincsname", if_test, if_in_csnname_code);
primitive("ifpdfabsnum", if_test, if_pdfabs_num_code);
primitive("ifpdfabsdim", if_test, if_pdfabs_dim_code);

```

1763. \langle Cases of expandafter for print_cmd_chr 1763 $\rangle \equiv$
else print_esc("unless")

This code is used in section 288.

1764. \langle Cases of if.test for print_cmd_chr 1764 $\rangle \equiv$

```

if_def_code: print_esc("ifdef");
if_cs_code: print_esc("ifcsname");
if_font_char_code: print_esc("iffontchar");
if_in_csnname_code: print_esc("ifincsname");
if_pdfabs_num_code: print_esc("ifpdfabsnum");
if_pdfabs_dim_code: print_esc("ifpdfabsdim");

```

This code is used in section 514.

1765. The result of a boolean condition is reversed when the conditional is preceded by \unless.

\langle Negate a boolean conditional and goto reswitch 1765 $\rangle \equiv$

```

begin get_token;
if ( $cur\_cmd = if\_test$ )  $\wedge$  ( $cur\_chr \neq if\_case\_code$ ) then
  begin  $cur\_chr \leftarrow cur\_chr + unless\_code$ ; 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 391.

1766. 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 1766> ≡
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 1767 and 1769.

This code is used in section 527.

1767. 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 1766> +≡
if_cs_code: begin n ← get_avail; p ← n; { head of the list of characters }
  e ← is_in_csname; is_in_csname ← true;
  repeat get_x_token;
    if cur_cs = 0 then store_new_token(cur_tok);
  until cur_cs ≠ 0;
  if cur_cmd ≠ end_cs_name then <Complain about missing \endcsname 399>;
  <Look up the characters of list n in the hash table, and set cur_cs 1768>;
  flush_list(n); b ← (eq_type(cur_cs) ≠ undefined_cs); is_in_csname ← e;
end;
```

1768. <Look up the characters of list `n` in the hash table, and set `cur_cs` 1768> ≡

```
m ← first; p ← link(n);
while p ≠ null do
  begin if m ≥ max_buf_stack then
    begin max_buf_stack ← m + 1;
    if max_buf_stack = buf_size then overflow("buffer_size", buf_size);
    end;
    buffer[m] ← info(p) mod '400; incr(m); p ← link(p);
  end;
  if m > first + 1 then cur_cs ← id_lookup(first, m - first) { no_new_control_sequence is true }
  else if m = first then cur_cs ← null_cs { the list is empty }
  else cur_cs ← single_base + buffer[first] { the list has length one }
```

This code is used in section 1767.

1769. The conditional `\iffontchar` tests the existence of a character in a font.

```
<Cases for conditional 1766> +≡
if_in_csnname_code: b ← is_in_csnname;
if_pdfabs_dim_code, if_pdfabs_num_code: begin if this_if = if_pdfabs_num_code then
    scan_int else scan_normal_dimen;
    n ← cur_val;
    if n < 0 then negate(n);
    < Get the next non-blank non-call token 432 >;
    if (cur_tok ≥ other_token + "<") ∧ (cur_tok ≤ other_token + ">") then r ← cur_tok − other_token
    else begin print_err("Missing=inserted for"); print_cmd_chr(if_test, this_if);
        help1("I was expecting to see `<`, `=`, or `>'. Didn't."); back_error; r ← "=";
    end;
    if this_if = if_pdfabs_num_code then scan_int else scan_normal_dimen;
    if cur_val < 0 then negate(cur_val);
    case r of
        "<": b ← (n < cur_val);
        "=": b ← (n = cur_val);
        ">": b ← (n > cur_val);
    end;
    end;
if_font_char_code: begin scan_font_ident; n ← cur_val; scan_char_num;
    if (font_bc[n] ≤ cur_val) ∧ (font_ec[n] ≥ cur_val) then b ← char_exists(char_info(n)(qi(cur_val)))
    else b ← false;
end;
```

1770. 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  $\varepsilon$ -TEX primitives 1649> +≡
primitive("protected", prefix, 8);
```

1771. <Cases of *prefix* for `print_cmd_chr` 1771> ≡
else if *chr_code* = 8 then `print_esc("protected")`

This code is used in section 1387.

1772. The `get_x_or_protected` procedure is like `get_x_token` except that protected macros are not expanded.

```
<Declare  $\varepsilon$ -TEX procedures for scanning 1682> +≡
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;
```

1773. 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 }
```

1774. 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  $\varepsilon$ -TEX procedures for tracing and input 306 ) +≡
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 1775 };
    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 ← warningIssued;
  end;
end;
```

1775. 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 1775 } ≡
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 1774 and 1776.

1776. 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 306 ⟩ +≡

```

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 1775 };
    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;

```

1777. 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 306 ⟩ +≡

```

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;

```

1778. Here are the additional ε -TeX primitives for expressions.

```
<Generate all  $\varepsilon$ -TeX primitives 1649> +≡
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);
```

1779. <Cases of *last_item* for *print_cmd_chr* 1650> +≡
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")*;

1780. 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 1780> ≡
begin if m < eTeX_mu then
  begin case m of
    <Cases for fetching a glue value 1807>
  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 1808>
  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 ≥ glue_val then
      begin m ← cur_val; cur_val ← new_spec(m); delete_glue_ref(m);
      <Negate all three glue components of cur_val 457>;
      end
    else negate(cur_val);
  return;
end
```

This code is used in section 450.

1781. <Declare ε -TeX procedures for scanning 1682> +≡
procedure *scan_expr*; *forward*;

1782. The *scan_expr* procedure scans and evaluates an expression.

```

⟨ Declare procedures needed for expressions 1782 ⟩ ≡
⟨ Declare subprocedures for scan_expr 1793 ⟩
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; incr(expand_depth_count);
if expand_depth_count ≥ expand_depth then overflow("expansion_depth", expand_depth);
⟨ Scan and evaluate an expression e of type l 1783 ⟩;
decr(expand_depth_count);
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 ≥ 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 1787.

This code is used in section 487.

1783. 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 1783> ≡
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 1785>;
found: <Scan the next operator and set o 1784>;
    arith_error ← b; <Make sure that f is in the proper range 1790>;
    case s of
        <Cases for evaluation of the current term 1791>
    end; { there are no other cases }
    if o > expr_sub then s ← o else <Evaluate the current expression 1792>;
    b ← arith_error;
    if o ≠ expr_none then goto continue;
    if p ≠ null then <Pop the expression stack and goto found 1789>

```

This code is used in section 1782.

1784. <Scan the next operator and set o 1784> ≡
 <Get the next non-blank non-call token 432>;
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 1783.

1785. <Scan a factor f of type o or start a subexpression 1785> ≡
 <Get the next non-blank non-call token 432>;
if cur_tok = other_token + "(" **then** <Push the expression stack and goto restart 1788>;
 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 1783.

1786. \langle Declare ε -TEX procedures for scanning 1682 $\rangle + \equiv$
procedure *scan_normal_glue*; *forward*;
procedure *scan_mu_glue*; *forward*;

1787. Here we declare two trivial procedures in order to avoid mutually recursive procedures with parameters.

\langle Declare procedures needed for expressions 1782 $\rangle + \equiv$
procedure *scan_normal_glue*;
 begin *scan_glue(glue_val)*;
 end;
procedure *scan_mu_glue*;
 begin *scan_glue(mu_val)*;
 end;

1788. 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 being 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(#)  $\equiv$  mem[# + 1].int { saved expression so far }
define expr_t_field(#)  $\equiv$  mem[# + 2].int { saved term so far }
define expr_n_field(#)  $\equiv$  mem[# + 3].int { saved numerator }

 $\langle$  Push the expression stack and goto restart 1788  $\rangle \equiv$ 
begin q  $\leftarrow$  get_node(expr_node_size); link(q)  $\leftarrow$  p; type(q)  $\leftarrow$  l; subtype(q)  $\leftarrow$  4 * s + r;
  expr_e_field(q)  $\leftarrow$  e; expr_t_field(q)  $\leftarrow$  t; expr_n_field(q)  $\leftarrow$  n; p  $\leftarrow$  q; l  $\leftarrow$  o; goto restart;
end
```

This code is used in section 1785.

1789. \langle Pop the expression stack and **goto** *found* 1789 $\rangle \equiv$
begin *f* \leftarrow *e*; *q* \leftarrow *p*; *e* \leftarrow *expr_e_field(q)*; *t* \leftarrow *expr_t_field(q)*; *n* \leftarrow *expr_n_field(q)*; *s* \leftarrow *subtype(q)* **div** 4;
r \leftarrow *subtype(q)* **mod** 4; *l* \leftarrow *type(q)*; *p* \leftarrow *link(q)*; *free_node(q, expr_node_size)*; **goto** *found*;
end

This code is used in section 1783.

1790. 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 1790 ⟩ ≡
if ( $l = \text{int\_val}$ )  $\vee$  ( $s > \text{expr\_sub}$ ) then
begin if ( $f > \text{infinity}$ )  $\vee$  ( $f < -\text{infinity}$ ) then num_error( $f$ );
end
else if  $l = \text{dimen\_val}$  then
begin if  $\text{abs}(f) > \text{max\_dimen}$  then num_error( $f$ );
end
else begin if ( $\text{abs}(\text{width}(f)) > \text{max\_dimen}$ )  $\vee$  ( $\text{abs}(\text{stretch}(f)) > \text{max\_dimen}$ )  $\vee$ 
( $\text{abs}(\text{shrink}(f)) > \text{max\_dimen}$ ) then glue_error( $f$ );
end
```

This code is used in section 1783.

1791. 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 1791 ⟩ ≡
expr_none: if ( $l \geq \text{glue\_val}$ )  $\wedge$  ( $o \neq \text{expr\_none}$ ) then
begin  $t \leftarrow \text{new\_spec}(f)$ ; delete_glue_ref( $f$ ); normalize_glue( $t$ );
end
else  $t \leftarrow f$ ;
```

See also sections 1795, 1796, and 1798.

This code is used in section 1783.

1792. 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(#,  $r = \text{expr\_sub}$ )
define expr_a(#) ≡ expr_add_sub(#, max_dimen)
⟨ Evaluate the current expression 1792 ⟩ ≡
begin  $s \leftarrow \text{expr\_none}$ ;
if  $r = \text{expr\_none}$  then  $e \leftarrow t$ 
else if  $l = \text{int\_val}$  then  $e \leftarrow \text{expr\_add\_sub}(e, t, \text{infinity})$ 
else if  $l = \text{dimen\_val}$  then  $e \leftarrow \text{expr\_a}(e, t)$ 
else ⟨ Compute the sum or difference of two glue specs 1794 ⟩;
 $r \leftarrow o$ ;
end
```

This code is used in section 1783.

1793. 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 1793 ⟩ ≡
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 ≥ 0 then
    if y ≤ max_answer - x then a ← x + y else num_error(a)
    else if y ≥ -max_answer - x then a ← x + y else num_error(a);
    add_or_sub ← a;
  end;
```

See also sections 1797 and 1799.

This code is used in section 1782.

1794. We know that $stretch_order(e) > normal$ implies $stretch(e) \neq 0$ and $shrink_order(e) > normal$ implies $shrink(e) \neq 0$.

```
⟨ Compute the sum or difference of two glue specs 1794 ⟩ ≡
  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 1792.

1795. 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 1791 ⟩ +≡
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;
```

1796. Here we divide the term `t` by the factor `f`.

```
define expr_d(#) ≡ # ← quotient(#, f)
⟨ Cases for evaluation of the current term 1791 ⟩ +≡
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;
```

1797. The function *quotient*(*n, d*) computes the rounded quotient $q = \lfloor n/d + \frac{1}{2} \rfloor$, when *n* and *d* are positive.

\langle Declare subprocedures for *scan_expr* 1793 $\rangle +\equiv$

```
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  $\leftarrow$  false
    else begin negate(d); negative  $\leftarrow$  true;
    end;
  if n < 0 then
    begin negate(n); negative  $\leftarrow$   $\neg$ negative;
    end;
  a  $\leftarrow$  n div d; n  $\leftarrow$  n - a * d; d  $\leftarrow$  n - d; { avoid certain compiler optimizations! }
  if d + n  $\geq$  0 then incr(a);
  if negative then negate(a);
  end;
  quotient  $\leftarrow$  a;
end;
```

1798. Here the term *t* is multiplied by the quotient *n/f*.

```
define expr_s(#)  $\equiv$  #  $\leftarrow$  fract(#, n, f, max_dimen)
 $\langle$  Cases for evaluation of the current term 1791  $\rangle +\equiv$ 
expr_scale: if l = int_val then t  $\leftarrow$  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;
```

1799. Finally, the function $\text{fract}(x, n, d, \text{max_answer})$ 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 1793 ⟩ +≡
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  $2 * h \geq d$  }
    r: integer; { intermediate remainder }
    t: integer; { temp variable }
  begin if d = 0 then goto too_big;
  a  $\leftarrow$  0;
  if d > 0 then negative  $\leftarrow$  false
  else begin negate(d); negative  $\leftarrow$  true;
  end;
  if x < 0 then
    begin negate(x); negative  $\leftarrow$   $\neg$ negative;
    end
  else if x = 0 then goto done;
  if n < 0 then
    begin negate(n); negative  $\leftarrow$   $\neg$ 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  $0 < n \leq x < d$  }
  ⟨ Compute  $f = \lfloor xn/d + \frac{1}{2} \rfloor$  1800 ⟩
  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  $\leftarrow$  a;
  end;

```

1800. The loop here preserves the following invariant relations between f , x , n , and r : (i) $f + \lfloor (xn + (r + d))/d \rfloor = \lfloor xn_0/d + \frac{1}{2} \rfloor$; (ii) $-d \leq r < 0 < n \leq x < d$, 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  $f = \lfloor xn/d + \frac{1}{2} \rfloor$  1800⟩ ≡
   $f \leftarrow 0; r \leftarrow (d \text{ div } 2) - d; h \leftarrow -r;$ 
  loop begin if odd( $n$ ) then
    begin  $r \leftarrow r + x;$ 
    if  $r \geq 0$  then
      begin  $r \leftarrow r - d; incr(f);$ 
      end;
    end;
     $n \leftarrow n \text{ div } 2;$ 
    if  $n = 0$  then goto found1;
    if  $x < h$  then  $x \leftarrow x + x$ 
    else begin  $t \leftarrow x - d; x \leftarrow t + x; f \leftarrow f + n;$ 
      if  $x < n$  then
        begin if  $x = 0$  then goto found1;
         $t \leftarrow x; x \leftarrow n; n \leftarrow t;$ 
        end;
      end;
    end;
  end;
found1:
```

This code is used in section 1799.

1801. 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  $\varepsilon$ -TEX primitives 1649⟩ +≡
  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);
```

1802. ⟨Cases of `last_item` for `print_cmd_chr` 1650⟩ +≡

```

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");
```

1803. ⟨Cases for fetching an integer value 1651⟩ +≡

```

glue_stretch_order_code, glue_shrink_order_code: begin scan_normal_glue; q ← cur_val;
  if m = glue_stretch_order_code then cur_val ← stretch_order(q)
  else cur_val ← shrink_order(q);
  delete_glue_ref(q);
end;
```

1804. \langle Cases for fetching a dimension value 1671 $\rangle +\equiv$

```
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;
```

1805. 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  $\varepsilon$ -TEX primitives 1649 ⟩ +≡
primitive("mutoglue", last_item, mu_to_glue_code); primitive("gluetomu", last_item, glue_to_mu_code);
```

1806. \langle Cases of *last_item* for `print_cmd_chr` 1650 $\rangle +\equiv$

```
mu_to_glue_code: print_esc("mutoglue");
glue_to_mu_code: print_esc("gluetomu");
```

1807. \langle Cases for fetching a glue value 1807 $\rangle \equiv$

```
mu_to_glue_code: scan_mu_glue;
```

This code is used in section 1780.

1808. \langle Cases for fetching a mu value 1808 $\rangle \equiv$

```
glue_to_mu_code: scan_normal_glue;
```

This code is used in section 1780.

1809. ε -TEX (in extended mode) supports 32768 (i.e., 2^{15}) count, dimen, skip, muskip, box, and token registers. As in T_EX 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 $0 \leq n \leq 32767$ (where `\marks{0}` 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 `\topmarks{0}` is again synonymous to `\topmark`. As in T_EX 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.

\langle Generate all ε -TEX primitives 1649 $\rangle +\equiv$

```
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);
```

1810. The `scan_register_num` procedure scans a register number that must not exceed 255 in compatibility mode resp. 32767 in extended mode.

\langle Declare ε -TEX procedures for expanding 1752 $\rangle +\equiv$

```
procedure scan_register_num; forward;
```

1811. \langle Declare procedures that scan restricted classes of integers 459 $\rangle +\equiv$
procedure *scan_register_num*;

```
begin scan_int;
if (cur_val < 0)  $\vee$  (cur_val > max_reg_num) then
  begin print_err("Bad_register_code");
    help2(max_reg_help_line)("I_unchanged_this_one_to_zero."); int_error(cur_val);
    cur_val  $\leftarrow$  0;
  end;
end;
```

1812. \langle Initialize variables for ε -TEX compatibility mode 1812 $\rangle \equiv$

```
max_reg_num  $\leftarrow$  255; max_reg_help_line  $\leftarrow$  "A_register_number_must_be_between_0_and_255.";
```

This code is used in sections 1653 and 1655.

1813. \langle Initialize variables for ε -TEX extended mode 1813 $\rangle \equiv$

```
max_reg_num  $\leftarrow$  32767; max_reg_help_line  $\leftarrow$  "A_register_number_must_be_between_0_and_32767.";
```

This code is used in sections 1648 and 1655.

1814. \langle Global variables 13 $\rangle +\equiv$

```
max_reg_num: halfword; { largest allowed register number }
```

```
max_reg_help_line: str_number; { first line of help message }
```

1815. 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 $sa_index < dimen_val.limit$; it represents a skip or muskip register if and only if $dimen_val.limit \leq sa_index < mu_val.limit$; it represents a box register if and only if $mu_val.limit \leq sa_index < box_val.limit$; it represents a token list register if and only if $box_val.limit \leq sa_index < tok_val.limit$; finally it represents a mark class if and only if $tok_val.limit \leq sa_index$.

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 { 24 · (dimen_val + 1) }
define mu_val_limit = "40 { 24 · (mu_val + 1) }
define box_val_limit = "50 { 24 · (box_val + 1) }
define tok_val_limit = "60 { 24 · (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  $\varepsilon$ -TEX procedures for expanding 1752 ⟩ +≡
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;
```

1816. 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 }
```

1817. ⟨ Set initial values of key variables 21 ⟩ +≡
 sa_mark ← *null*; *sa_null.hh.lh* ← *null*; *sa_null.hh.rh* ← *null*;

1818. \langle Initialize table entries (done by INITEX only) [182](#) $\rangle +\equiv$
for $i \leftarrow int_val$ **to** tok_val **do** $sa_root[i] \leftarrow null;$

1819. 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  $\varepsilon$ -TEX procedures for expanding 1752 ⟩ +≡
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)  $\wedge$  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 1820 ⟩;
  link(cur_ptr) ← q; add_sa_ptr;
```

```
exit: end;
```

1820. 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 esoteric 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 1820⟩ ≡
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 1819.

1821. 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  $\varepsilon$ -TEX procedures for tracing and input 306 ⟩ +≡
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;

```

1822. 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;

```

1823. 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  $\varepsilon$ -TeX procedures for tracing and input 306 > +≡
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
```

1824. 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 1824 > ≡
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 412.

1825. 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 performs certain actions indicated 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) { \topmarks_n }
define sa_first_mark(#) ≡ link(# + 1) { \firstmarks_n }
define sa_bot_mark(#) ≡ info(# + 2) { \botmarks_n }
define sa_split_first_mark(#) ≡ link(# + 2) { \splitfirstmarks_n }
define sa_split_bot_mark(#) ≡ info(# + 3) { \splitbotmarks_n }

⟨ Declare the function called do_marks 1825 ⟩ ≡
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 ← 0 to 15 do
      begin get_sa_ptr;
      if cur_ptr ≠ 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 ← null;
        end;
      end
    else { q is the node for a mark class }
      begin case a of
        ⟨ Cases for do_marks 1826 ⟩
      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 ← null;
          end;
        end; do_marks ← (q = null);
      end;
    end;
  end;

```

This code is used in section 1154.

1826. At the start of the *vsplit* routine the existing *split-first-mark* and *split-bot-mark* are discarded.

```

⟨ Cases for do_marks 1826 ⟩ ≡
vsplit_init: if sa_split_first_mark(q) ≠ null then
  begin delete_token_ref(sa_split_first_mark(q)); sa_split_first_mark(q) ← null;
  delete_token_ref(sa_split_bot_mark(q)); sa_split_bot_mark(q) ← null;
  end;

```

See also sections 1828, 1829, and 1831.

This code is used in section 1825.

1827. We use again the fact that $split_first_mark = null$ if and only if $split_bot_mark = null$.

```
< Update the current marks for vsplit 1827 > ≡
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 1156.

1828. 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 $bot_mark \neq null$ implies $first_mark \neq null$; it also knows that $bot_mark = null$ implies $top_mark = first_mark = null$.

```
< Cases for do_marks 1826 > +≡
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;
```

1829. \langle Cases for do_marks 1826 $\rangle +≡$

```
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;
```

1830. \langle Update the current marks for fire_up 1830 $\rangle ≡$

```
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 1191.

1831. 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 1826) +≡
  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;
  end;
  tini
```

1832. 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 1832) ≡
  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 438.

1833. 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 1833) ≡
  begin print_esc("toks");
  if chr_code ≠ mem_bot then print_sa_num(chr_code);
  end
```

This code is used in section 288.

1834. 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 1834) ≡
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 297.

1835. 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 lower 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}
```

1836. \langle Set initial values of key variables 21 $\rangle +\equiv$
 $sa_chain \leftarrow null; sa_level \leftarrow level_zero;$

1837. 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  $\equiv$  sa_ref { location of saved item }
⟨ Declare  $\varepsilon$ -TEX procedures for tracing and input 306 ⟩  $+ \equiv$ 
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  $\neq$  sa_level then
begin check_full_save_stack; save_type(save_ptr)  $\leftarrow$  restore_sa; save_level(save_ptr)  $\leftarrow$  sa_level;
save_index(save_ptr)  $\leftarrow$  sa_chain; incr(save_ptr); sa_chain  $\leftarrow$  null; sa_level  $\leftarrow$  cur_level;
end;
 $i \leftarrow sa\_index(p)$ ;
if  $i < dimen\_val\_limit$  then
begin if sa_int( $p$ ) = 0 then
begin  $q \leftarrow get\_node(pointer\_node\_size)$ ;  $i \leftarrow tok\_val\_limit$ ;
end
else begin  $q \leftarrow get\_node(word\_node\_size)$ ; sa_int( $q$ )  $\leftarrow$  sa_int( $p$ );
end;
sa_ptr( $q$ )  $\leftarrow$  null;
end
else begin  $q \leftarrow get\_node(pointer\_node\_size)$ ; sa_ptr( $q$ )  $\leftarrow$  sa_ptr( $p$ );
end;
sa_loc( $q$ )  $\leftarrow p$ ; sa_index( $q$ )  $\leftarrow i$ ; sa_lev( $q$ )  $\leftarrow$  sa_lev( $p$ ); link( $q$ )  $\leftarrow$  sa_chain; sa_chain  $\leftarrow q$ ; add_sa_ref( $p$ );
end;
```

1838. \langle Declare ε -TEX procedures for tracing and input 306 $\rangle +\equiv$
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) \neq null then
if sa_index(p) $<$ box_val_limit then flush_node_list(sa_ptr(p))
else delete_token_ref(sa_ptr(p));
end;

1839. 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  $\varepsilon$ -TEX procedures for tracing and input 306 ⟩ +≡
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;

```

1840. 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  $\varepsilon$ -TeX procedures for tracing and input 306 > +≡
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;
```

1841. The *sa_restore* procedure restores the sparse array entries pointed at by *sa_chain*.

```
< Declare  $\varepsilon$ -TeX procedures for tracing and input 306 > +≡
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;
```

1842. 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 }
```

1843. 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 1843⟩ ≡
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;
  end
```

This code is used in section 1003.

1844. ⟨Other local variables for *try_break* 1006⟩ +≡
g: scaled; { glue stretch or shrink of test line, adjustment for last line }

1845. 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 1845⟩ ≡
begin active_short(q) ← 0; active_glue(q) ← 0;
end
```

This code is used in section 1040.

1846. 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 1846) ≡
begin if (active_short(r) = 0) ∨ (active_glue(r) ≤ 0) then goto not_found;
   { previous line was neither stretched nor shrunk, or was infinitely bad }
if (cur_active_width[3] ≠ fill_width[0]) ∨ (cur_active_width[4] ≠ fill_width[1]) ∨
   (cur_active_width[5] ≠ 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  $g \leftarrow cur\_active\_width[2]$ 
else  $g \leftarrow cur\_active\_width[6]$ ;
if  $g \leq 0$  then goto not_found; { no finite stretch resp. no shrink }
arith_error ← false;  $g \leftarrow fract(g, active\_short(r), active\_glue(r), max\_dimen)$ ;
if last_line_fit < 1000 then  $g \leftarrow fract(g, last\_line\_fit, 1000, max\_dimen)$ ;
if arith_error then
  if active_short(r) > 0 then  $g \leftarrow max\_dimen$  else  $g \leftarrow -max\_dimen$ ;
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 1847 }
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 1848 };
not_found: end
```

This code is used in section 1028.

1847. 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 1847) ≡
begin if  $g > shortfall$  then  $g \leftarrow shortfall$ ;
if  $g > 7230584$  then
  if  $cur\_active\_width[2] < 1663497$  then
    begin  $b \leftarrow inf\_bad$ ; fit_class ← very_loose_fit; goto found;
    end;
 $b \leftarrow badness(g, cur\_active\_width[2])$ ;
if  $b > 12$  then
  if  $b > 99$  then fit_class ← very_loose_fit
  else fit_class ← loose_fit
  else fit_class ← decent_fit;
  goto found;
end
```

This code is used in section 1846.

1848. { Set the value of b to the badness of the last line for shrinking, compute the corresponding fit_class,
 and **goto found** 1848 } ≡

```
begin if  $-g > cur\_active\_width[6]$  then  $g \leftarrow -cur\_active\_width[6]$ ;
 $b \leftarrow badness(-g, cur\_active\_width[6])$ ;
if  $b > 12$  then fit_class ← tight_fit else fit_class ← decent_fit;
goto found;
end
```

This code is used in section 1846.

1849. Vanishing values of *shortfall* and *g* indicate that the last line is not adjusted.

```
( Adjust the additional data for last line 1849 ) ≡
  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 1027.

1850. For each feasible break we record the shortfall and glue stretch or shrink (or adjustment).

```
( Store additional data for this feasible break 1850 ) ≡
  begin best_pl_short[fit_class] ← shortfall; best_pl_glue[fit_class] ← g;
  end
```

This code is used in section 1031.

1851. Here we save these data in the active node representing a potential line break.

```
( Store additional data in the new active node 1851 ) ≡
  begin active_short(q) ← best_pl_short[fit_class]; active_glue(q) ← best_pl_glue[fit_class];
  end
```

This code is used in section 1021.

1852. ⟨ Print additional data in the new active node 1852 ⟩ ≡

```
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 1022.

1853. Here we either reset *do_last_line_fit* or adjust the *par_fill_skip* glue.

```
( Adjust the final line of the paragraph 1853 ) ≡
  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 1039.

1854. 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 *p₁* of length 1, with *hyp_root* = *trie_r[0]* replacing *trie_root* and *lc_code(p₁)* replacing the *trie_op* code. This allows to compress and pack them together with the patterns with minimal changes to the existing code.

```
define hyp_root ≡ trie_r[0] { root of the linked trie for hyp_codes }
( Initialize table entries (done by INITEX only) 182 ) +≡
  hyp_root ← 0; hyp_start ← 0;
```

1855. \langle Store hyphenation codes for current language 1855 $\rangle \equiv$

```

begin  $c \leftarrow cur\_lang$ ;  $first\_child \leftarrow false$ ;  $p \leftarrow 0$ ;
repeat  $q \leftarrow p$ ;  $p \leftarrow trie\_r[q]$ ;
until ( $p = 0$ )  $\vee$  ( $c \leq so(trie\_c[p])$ );
if ( $p = 0$ )  $\vee$  ( $c < so(trie\_c[p])$ ) then
  ⟨ Insert a new trie node between  $q$  and  $p$ , and make  $p$  point to it 1141 ⟩;
   $q \leftarrow p$ ; { now node  $q$  represents  $cur\_lang$  }
⟨ Store all current  $lc\_code$  values 1856 ⟩;
end

```

This code is used in section 1137.

1856. 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.

\langle Store all current lc_code values 1856 $\rangle \equiv$

```

 $p \leftarrow trie\_l[q]$ ;  $first\_child \leftarrow true$ ;
for  $c \leftarrow 0$  to 255 do
  if ( $lc\_code(c) > 0$ )  $\vee$  (( $c = 255$ )  $\wedge$   $first\_child$ ) then
    begin if  $p = 0$  then ⟨ Insert a new trie node between  $q$  and  $p$ , and make  $p$  point to it 1141 ⟩
    else  $trie\_c[p] \leftarrow si(c)$ ;
     $trie\_o[p] \leftarrow qi(lc\_code(c))$ ;  $q \leftarrow p$ ;  $p \leftarrow trie\_r[q]$ ;  $first\_child \leftarrow false$ ;
    end;
  if  $first\_child$  then  $trie\_l[q] \leftarrow 0$  else  $trie\_r[q] \leftarrow 0$ 

```

This code is used in section 1855.

1857. We must avoid to “take” location 1, in order to distinguish between lc_code values and patterns.

\langle Pack all stored $hyph_codes$ 1857 $\rangle \equiv$

```

begin if  $trie\_root = 0$  then
  for  $p \leftarrow 0$  to 255 do  $trie\_min[p] \leftarrow p + 2$ ;
   $first\_fit(hyph\_root)$ ;  $trie\_pack(hyph\_root)$ ;  $hyph\_start \leftarrow trie\_ref[hyph\_root]$ ;
end

```

This code is used in section 1143.

1858. The global variable $hyph_index$ will point to the hyphenation codes for the current language.

```

define set_hyph_index ≡ { set  $hyph\_index$  for current language }
  if  $trie\_char(hyph\_start + cur\_lang) \neq qi(cur\_lang)$  then  $hyph\_index \leftarrow 0$ 
    { no hyphenation codes for  $cur\_lang$  }
  else  $hyph\_index \leftarrow trie\_link(hyph\_start + cur\_lang)$ 
define set_lc_code(#) ≡ { set  $hc[0]$  to hyphenation or lc code for # }
  if  $hyph\_index = 0$  then  $hc[0] \leftarrow lc\_code(#)$ 
  else if  $trie\_char(hyph\_index + #) \neq qi(#)$  then  $hc[0] \leftarrow 0$ 
  else  $hc[0] \leftarrow qo(trie\_op(hyph\_index + #))$ 

```

\langle Global variables 13 $\rangle +\equiv$

```

 $hyph\_start$ :  $trie\_pointer$ ; { root of the packed trie for  $hyph\_codes$  }
 $hyph\_index$ :  $trie\_pointer$ ; { pointer to hyphenation codes for  $cur\_lang$  }

```

1859. 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 }
```

1860. ⟨Set initial values of key variables 21⟩ +≡
 $page_disc \leftarrow null; split_disc \leftarrow null;$

1861. The \pagediscards and \splitediscards 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  $\varepsilon$ -TEX primitives 1649⟩ +≡
primitive("pagediscards", un_vbox, last_box_code);
primitive("splitediscards", un_vbox, vsplit_code);
```

1862. ⟨Cases of *un_vbox* for *print_cmd_chr* 1862⟩ ≡
else if *chr_code* = *last_box_code* then *print_esc*("pagediscards")
else if *chr_code* = *vsplit_code* then *print_esc*("splitediscards")

This code is used in section 1286.

1863. ⟨Handle saved items and *goto done* 1863⟩ ≡
begin *link(tail)* \leftarrow *disc_ptr[cur_chr]*; *disc_ptr[cur_chr]* \leftarrow *null*; *goto done*;
end

This code is used in section 1288.

1864. 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  $\varepsilon$ -TEX primitives 1649⟩ +≡
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);
```

1865. ⟨Cases of *set_shape* for *print_cmd_chr* 1865⟩ ≡
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 288.

1866. \langle Fetch a penalties array element 1866 $\rangle \equiv$
begin *scan_int*;
 if (*equiv*(*m*) = *null*) \vee (*cur_val* < 0) **then** *cur_val* \leftarrow 0
 else begin if *cur_val* > *penalty*(*equiv*(*m*)) **then** *cur_val* \leftarrow *penalty*(*equiv*(*m*));
 cur_val \leftarrow *penalty*(*equiv*(*m*) + *cur_val*);
 end;
end

This code is used in section 449.

1867. System-dependent changes. This section should be replaced, if necessary, by any special modifications of the program that are necessary to make T_EX 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.

1868. 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.

```
**: 37, 560
*: 192, 194, 196, 335, 382, 674, 1032, 1183, 1602
->: 316
=>: 385
???: 59
?: 83
@: 1032
@@: 1022
a: 47, 102, 122, 236, 303, 544, 545, 549, 586, 597,
  624, 678, 686, 698, 867, 898, 914, 928, 1253,
  1301, 1372, 1389, 1414, 1435, 1513, 1600, 1637,
  1679, 1782, 1793, 1797, 1799, 1825
A <box> was supposed to...: 1262
a_close: 28, 51, 351, 511, 512, 1453, 1513,
  1622, 1626
a_leaders: 167, 207, 653, 655, 662, 664, 735,
  744, 832, 847, 1249, 1250, 1251, 1256, 1326,
  1681, 1699
a_make_name_string: 551, 560, 563
a_open_in: 27, 51, 563, 1453
a_open_out: 27, 560, 1622
A_token: 471
ab_vs_cd: 122, 127
abort: 586, 589, 590, 591, 594, 595, 596, 597,
  598, 600, 602
above: 226, 1224, 1356, 1357, 1358
\above primitive: 1356
above_code: 1356, 1357, 1360, 1361
above_display_short_skip: 242, 990
\abovedisplayshortskip primitive: 244
above_display_short_skip_code: 242, 243, 244, 1381
above_display_skip: 242, 990
\abovedisplayskip primitive: 244
above_display_skip_code: 242, 243, 244, 1381, 1384
\abovewithdelims primitive: 1356
abs: 66, 125, 126, 127, 204, 229, 236, 237, 444,
  448, 474, 527, 637, 686, 690, 692, 693, 705, 706,
  839, 851, 894, 913, 933, 934, 935, 1007, 1012,
  1025, 1035, 1121, 1125, 1206, 1207, 1234, 1254,
  1256, 1258, 1261, 1271, 1288, 1298, 1305, 1327,
  1421, 1422, 1560, 1561, 1563, 1625, 1681, 1790
absorbing: 327, 328, 361, 499, 1683
acc_kern: 173, 209, 1303
accent: 226, 287, 288, 1268, 1300, 1342, 1343
\accent primitive: 287
accent_chr: 863, 872, 914, 1343
accent_noad: 863, 866, 872, 874, 909, 937,
  1343, 1364
accent_noad_size: 863, 874, 937, 1343
act_width: 1042, 1043, 1044, 1047, 1609
action procedure: 1206
active: 180, 995, 1005, 1019, 1030, 1036, 1037,
  1039, 1040, 1041, 1049, 1050, 1051
active_base: 238, 240, 270, 271, 273, 284, 285, 375,
  468, 532, 706, 1330, 1435, 1467, 1493, 1495
active_char: 225, 366, 532
active_glue: 1842, 1845, 1846, 1851, 1852, 1853
active_height: 1147, 1152, 1153
active_node_size: 1021, 1036, 1040, 1041, 1842,
  1843
active_node_size_extended: 1842, 1843
active_node_size_normal: 995, 1843
active_short: 1842, 1845, 1846, 1851, 1852, 1853
active_width: 999, 1000, 1005, 1015, 1019, 1037,
  1040, 1042, 1043, 1044, 1047, 1147
actual_looseness: 1048, 1049, 1051
add_action_ref: 1536, 1604
add_char_shrink: 828, 1015, 1018, 1042, 1043,
  1046, 1047
add_char_shrink_end: 1015
add_char_stretch: 828, 1015, 1018, 1042, 1043,
  1046, 1047
add_char_stretch_end: 1015
add_delims_to: 369
add_disc_width_to_active_width: 1015, 1045
add_disc_width_to_break_width: 1015, 1016
add_glue_ref: 221, 224, 456, 978, 1057, 1173, 1278,
  1407, 1604, 1733, 1782, 1820
add_kern_shrink: 1015, 1018, 1042, 1046, 1047
add_kern_shrink_end: 1015
add_kern_stretch: 1015, 1018, 1042, 1046, 1047
add_kern_stretch_end: 1015
add_or_sub: 1792, 1793
add_sa_ptr: 1819
add_sa_ref: 1399, 1402, 1821, 1837, 1839, 1840
add_token_ref: 221, 224, 345, 1156, 1189, 1193,
  1399, 1405, 1604, 1637, 1827, 1828, 1829, 1830
additional: 816, 817, 833, 848
```

adj_demerits: 254, 1012, 1035
\adjdemerits primitive: 256
adj_demerits_code: 254, 255, 256
adjust: 603
adjust_head: 180, 1065, 1066, 1254, 1263, 1377, 1383
adjust_interword_glue: 705, 1219
adjust_node: 160, 166, 193, 201, 220, 224, 819, 825, 831, 906, 937, 1005, 1042, 1076, 1278
adjust_pre: 160, 215, 831, 1278
adjust_ptr: 160, 215, 220, 224, 831, 1278
adjust_space_factor: 1211, 1215
adjust_tail: 819, 820, 823, 825, 831, 972, 1065, 1066, 1254, 1263, 1377
adjusted_hbox_group: 291, 1240, 1261, 1263, 1661, 1679
adv_char_width: 687, 690, 693, 726
adv_char_width_s: 687, 690, 693
adv_char_width_s_out: 687, 690, 693
adv_past: 1609
advance: 227, 287, 288, 1388, 1413, 1414, 1416
\advance primitive: 287
advance_major_tail: 1091, 1094
after: 165, 210, 1374, 1728, 1739
after_assignment: 226, 287, 288, 1446
\afterassignment primitive: 287
after_group: 226, 287, 288, 1449
\aftergroup primitive: 287
after_math: 1371, 1372
after_token: 1444, 1445, 1446, 1447
aire: 586, 587, 589, 603
align_error: 1304, 1305
align_group: 291, 944, 950, 967, 976, 1309, 1310, 1661, 1679
align_head: 180, 946, 953
align_peek: 949, 950, 961, 975, 1226, 1311
align_ptr: 946, 947, 948
align_stack_node_size: 946, 948
align_state: 88, 331, 346, 347, 348, 353, 361, 364, 369, 379, 420, 421, 422, 429, 468, 501, 508, 509, 512, 946, 947, 948, 950, 953, 959, 960, 961, 964, 965, 967, 1247, 1272, 1304, 1305
aligning: 327, 328, 361, 953, 965
alignment of rules with characters: 616
allocvffnts: 706, 715, 720
alpha: 586, 597, 599
alpha_file: 25, 27, 28, 31, 32, 50, 54, 326, 506, 551, 1522
alpha_token: 464, 466
alt_rule: 1550, 1551, 1552, 1556, 1565
alter_aux: 1420, 1421
alter_box_dimen: 1420, 1425

alter_integer: 1420, 1424
alter_page_so_far: 1420, 1423
alter_prev_graf: 1420, 1422
Ambiguous...: 1361
Amble, Ole: 1102
AmSTeX: 1511
any_mode: 1223, 1226, 1235, 1241, 1245, 1251, 1275, 1280, 1282, 1304, 1312, 1388, 1446, 1449, 1452, 1454, 1463, 1468, 1527
any_state_plus: 366, 367, 369
app_display: 1381, 1382, 1383, 1744
app_kern: 1295
app_lc_hex: 48
app_space: 1207, 1221
append_bead: 1637
append_char: 42, 48, 52, 58, 198, 213, 279, 542, 551, 706, 712, 717, 719, 726, 727, 868, 871, 1116
append_charnode_to_t: 1085, 1088
append_choices: 1349, 1350
append_dest_name: 698
append_discretionary: 1294, 1295
append_glue: 1235, 1238, 1256
append_italic_correction: 1290, 1291
append_kern: 1235, 1239
append_link: 730, 783, 1632, 1635, 1636
append_list: 160, 975, 1065, 1254
append_list_end: 160
append_nl: 686
append_normal_space: 1207
append_penalty: 1280, 1281
append_ptr: 698, 700
append_thread: 739, 1637
append_to_name: 545, 549
append_to_vlist: 855, 975, 1065, 1254, 1744
area_delimiter: 539, 541, 542, 543
Argument of \x has...: 421
arith_error: 104, 105, 106, 107, 112, 114, 474, 479, 486, 689, 1414, 1782, 1783, 1790, 1846
Arithmetic overflow: 1414, 1782
artificial_demerits: 1006, 1027, 1030, 1031, 1032
ASCII code: 17, 529
ASCII_code: 18, 19, 20, 29, 30, 31, 38, 42, 54, 58, 60, 82, 314, 363, 415, 542, 545, 549, 868, 1069, 1089, 1098, 1120, 1127, 1130, 1136, 1137, 1624
assign_dimen: 227, 266, 267, 439, 1388, 1402, 1406
assign_font_dimen: 227, 287, 288, 439, 1388, 1431
assign_font_int: 227, 439, 1388, 1431, 1432, 1433
assign_glue: 227, 244, 245, 439, 958, 1388, 1402, 1406
assign_int: 227, 256, 257, 439, 1388, 1400, 1402, 1406, 1415, 1657, 1701

assign_mu_glue: 227, 244, 245, 439, 1388, 1400, 1402, 1406, 1415
assign_toks: 227, 248, 249, 251, 345, 439, 441, 1388, 1402, 1404, 1405, 1657
assign_trace: 299, 300, 301
at: 1436
\atop primitive: 1356
atop_code: 1356, 1357, 1360
\atopwithdelims primitive: 1356
attach_fraction: 474, 479, 480, 482
attach_sign: 474, 475, 481
auto_breaking: 999, 1038, 1039, 1042, 1044
auto_expand: 705
auto_expand_font: 705, 720
auto_expand_vf: 712, 720
auto_kern: 173, 209, 705, 1005
aux: 230, 231, 234, 976, 988
aux_field: 230, 231, 236, 951
aux_save: 976, 988, 1384
avail: 136, 138, 139, 140, 141, 182, 186, 1489, 1490
AVAIL list clobbered...: 186
avl_find_obj: 1555
avl_put_obj: 698
awful_bad: 1009, 1010, 1011, 1012, 1030, 1050, 1147, 1151, 1152, 1164, 1182, 1183, 1184
axis_height: 876, 882, 912, 922, 923, 925, 938
b: 388, 490, 491, 496, 524, 549, 586, 624, 855, 881, 882, 885, 887, 891, 1006, 1147, 1171, 1376, 1425, 1466, 1656, 1744, 1782
b_close: 28, 586, 670, 712, 772, 794
b_make_name_string: 551, 558, 684
b_open_in: 27, 589
b_open_out: 27, 558, 684
back_error: 349, 399, 422, 429, 441, 468, 472, 502, 505, 529, 604, 959, 1256, 1262, 1339, 1375, 1385, 1390, 1765, 1769, 1784
back_input: 286, 303, 347, 348, 349, 392, 393, 394, 398, 401, 405, 421, 431, 433, 441, 469, 470, 474, 478, 481, 487, 552, 964, 1208, 1225, 1232, 1242, 1268, 1273, 1302, 1305, 1310, 1316, 1328, 1330, 1331, 1393, 1399, 1404, 1447, 1623, 1784, 1785
back_list: 345, 347, 359, 433, 1466
backed_up: 329, 333, 334, 336, 345, 346, 347, 1203
background: 999, 1000, 1003, 1013, 1039, 1040, 1843
backup_backup: 388
backup_head: 180, 388, 433
BAD: 315, 316
bad: 13, 14, 129, 312, 548, 1427, 1512
Bad \patterns: 1138
Bad \prevgraf: 1422
Bad character code: 460
Bad delimiter code: 463
Bad dump length: 497
Bad file offset: 497
Bad flag...: 188
Bad interaction mode: 1696
Bad link...: 200
Bad match number: 497
Bad mathchar: 462
Bad number: 461
Bad register code: 459, 1811
Bad space factor: 1421
bad_fmt: 1481, 1484, 1486, 1490, 1495, 1507
bad_pool: 51, 52, 53
bad_tfm: 586
bad_vf: 710, 712, 714, 717, 719, 725
badness: 108, 836, 843, 850, 854, 1004, 1028, 1029, 1152, 1184, 1847, 1848
\badness primitive: 442
badness_code: 442, 450
banner: 2, 61, 562, 1477
base_line: 647, 651, 652, 656, 729, 730, 733, 734, 737, 1645, 1646, 1647
base_ptr: 84, 85, 332, 333, 334, 335, 1309, 1774, 1775, 1776
baseline_skip: 242, 265, 855
\baselineskip primitive: 244
baseline_skip_code: 167, 242, 243, 244, 855
batch_mode: 73, 75, 86, 90, 92, 93, 561, 1440, 1441, 1507, 1508, 1696
\batchmode primitive: 1440
bc: 566, 567, 569, 571, 586, 591, 592, 596, 603, 706
bch_label: 586, 600, 603
bchar: 586, 600, 603, 1078, 1080, 1082, 1083, 1085, 1088, 1090, 1093, 1094, 1209, 1211, 1214, 1215, 1218
bchar_label: 575, 578, 603, 705, 706, 1086, 1093, 1211, 1218, 1500, 1501
be_careful: 112, 113, 114
before: 165, 210, 1374, 1710, 1712, 1718, 1728, 1739
before_rejected_cur_p: 999, 1039
begin: 7, 8
begin_box: 1251, 1257, 1262
begin_diagnostic: 76, 263, 306, 321, 345, 426, 427, 528, 535, 608, 666, 669, 750, 839, 851, 1002, 1039, 1164, 1169, 1183, 1188, 1299, 1471, 1474, 1662, 1677, 1691, 1823
begin_file_reading: 78, 87, 350, 509, 563, 1755
begin_group: 226, 287, 288, 1241
\begingroup primitive: 287
begin_insert_or_adjust: 1275, 1277
begin_L_code: 165, 1701, 1702, 1734

begin_LR_type: 165, 1707
begin_M: 1258
begin_M_code: 165, 1258, 1746
begin_name: 538, 541, 552, 553, 557
begin_pseudoprint: 338, 340, 341
begin_R_code: 165, 1701, 1702
begin_reflect: 1700
begin_token_list: 345, 381, 384, 412, 416, 950, 964, 965, 975, 1202, 1207, 1261, 1269, 1317, 1323, 1345, 1618
\beginL primitive: 1701
Beginning to dump...: 1508
\beginR primitive: 1701
below_display_short_skip: 242
\belowdisplayshortskip primitive: 244
below_display_short_skip_code: 242, 243, 244, 1381
below_display_skip: 242
\belowdisplayskip primitive: 244
below_display_skip_code: 242, 243, 244, 1381, 1384
best_bet: 1048, 1050, 1051, 1053, 1054, 1853
best_height_plus_depth: 1148, 1151, 1187, 1188
best_ins_ptr: 1158, 1182, 1186, 1195, 1197, 1198
best_line: 1048, 1050, 1051, 1053, 1065, 1067
best_page_break: 1157, 1182, 1190, 1191
best_pl_glue: 1842, 1850, 1851
best_pl_line: 1009, 1021, 1031
best_pl_short: 1842, 1850, 1851
best_place: 1009, 1021, 1031, 1147, 1151, 1157
best_size: 1157, 1182, 1194
beta: 586, 597, 599
bf: 720
big_op_spacing1: 877, 927
big_op_spacing2: 877, 927
big_op_spacing3: 877, 927
big_op_spacing4: 877, 927
big_op_spacing5: 877, 927
big_switch: 227, 254, 1171, 1206, 1207, 1208, 1213, 1219
BigEndian order: 566
biggest_char: 275, 281
billion: 653
bin_noad: 858, 866, 872, 874, 904, 905, 937, 1334, 1335
bin_op_penalty: 254, 937
\binoppenalty primitive: 256
bin_op_penalty_code: 254, 255, 256
blank_line: 263
bool: 496, 497
boolean: 27, 31, 37, 45, 46, 47, 76, 79, 96, 104, 106, 107, 112, 114, 183, 185, 263, 274, 303, 333, 383, 388, 389, 433, 439, 466, 474, 487, 496, 499, 524, 542, 550, 553, 575, 586, 605, 619, 647, 657,

673, 680, 686, 687, 689, 691, 693, 696, 698, 701, 702, 704, 705, 706, 720, 725, 729, 738, 749, 750, 793, 807, 817, 821, 823, 882, 895, 902, 967, 991, 999, 1001, 1004, 1005, 1006, 1053, 1077, 1084, 1127, 1137, 1145, 1166, 1189, 1209, 1229, 1232, 1257, 1269, 1283, 1338, 1372, 1389, 1414, 1459, 1481, 1513, 1522, 1537, 1550, 1555, 1628, 1640, 1656, 1660, 1661, 1662, 1756, 1774, 1776, 1782, 1793, 1797, 1799, 1819, 1825, 1842
bop: 610, 612, 613, 615, 617, 619, 666, 668
Bosshard, Hans Rudolf: 484
bot: 572
bot_mark: 408, 409, 1189, 1193, 1809, 1828
\botmark primitive: 410
bot_mark_code: 408, 410, 411, 1809
\botmarks primitive: 1809
bottom: 693
bottom_level: 291, 294, 303, 1242, 1246, 1661, 1679
bottom_line: 333
bowels: 619
box: 248, 250, 1169, 1170, 1186, 1192, 1194, 1195, 1198, 1200, 1205, 1820, 1821, 1839
\box primitive: 1249
box_base: 248, 250, 251, 273, 1255
box_code: 1249, 1250, 1257, 1285, 1288, 1861
box_context: 1253, 1254, 1255, 1256, 1257, 1261, 1262
box_end: 1253, 1257, 1262, 1264
box_error: 1169, 1170, 1192, 1205
box_flag: 1249, 1253, 1255, 1261, 1419, 1681
box_lr: 153, 643, 1704, 1714, 1715, 1745
box_max_depth: 265, 1264
\boxmaxdepth primitive: 266
box_max_depth_code: 265, 266
box_node_size: 153, 154, 220, 224, 823, 844, 891, 903, 927, 932, 1154, 1198, 1278, 1288, 1379, 1733, 1745
box_ref: 228, 250, 297, 1255
box_there: 1157, 1164, 1177, 1178
box_val: 1402, 1815, 1820, 1821, 1823, 1839
box_val_limit: 1815, 1838
\box255 is not void: 1192
bp: 484
bp: 690
brain: 1206
breadth_max: 199, 200, 216, 251, 254, 1005, 1519, 1823
break: 34
break_in: 34
break_node: 995, 1005, 1021, 1027, 1031, 1032, 1039, 1040, 1053, 1054
break_penalty: 226, 287, 288, 1280

break_type: 1005, 1013, 1021, 1022, 1035
break_width: 999, 1000, 1013, 1014, 1015, 1017, 1018, 1019, 1020, 1055
breakpoint: 1518
broken_ins: 1158, 1163, 1187, 1198
broken_penalty: 254, 1067
\brokenpenalty primitive: 256
broken_penalty_code: 254, 255, 256
broken_ptr: 1158, 1187, 1198
buf_size: 11, 30, 31, 35, 71, 129, 286, 337, 350, 353, 363, 385, 388, 400, 550, 556, 560, 1514, 1756, 1768
buffer: 30, 31, 36, 37, 45, 71, 83, 87, 88, 278, 279, 280, 286, 324, 325, 337, 340, 353, 363, 365, 374, 376, 377, 378, 382, 384, 385, 388, 400, 509, 510, 549, 550, 556, 557, 560, 564, 1517, 1519, 1648, 1756, 1761, 1768
 Buffer size exceeded: 35
build_choices: 1351, 1352
build_discretionary: 1296, 1297
build_page: 976, 988, 1165, 1171, 1203, 1232, 1238, 1254, 1269, 1272, 1278, 1281, 1323, 1378
by: 1414
byname: 1555, 1564
bypass_eoln: 31
byte: 702
byte_file: 25, 27, 28, 551, 558, 565, 680, 710, 772
b0: 128, 131, 132, 151, 239, 275, 290, 571, 572, 576, 580, 582, 590, 629, 710, 712, 714, 859, 861, 1098, 1135, 1487, 1488, 1754, 1756
b1: 128, 131, 132, 151, 239, 275, 290, 571, 572, 580, 582, 590, 629, 710, 712, 714, 859, 861, 1098, 1135, 1487, 1488, 1754, 1756
b2: 128, 131, 132, 571, 572, 580, 582, 590, 629, 710, 712, 714, 859, 861, 1487, 1488, 1754, 1756
b3: 128, 131, 132, 571, 572, 582, 590, 629, 710, 712, 714, 859, 861, 1487, 1488, 1754, 1756
c: 47, 63, 82, 162, 286, 296, 314, 363, 491, 496, 542, 545, 549, 586, 604, 608, 609, 619, 705, 817, 868, 870, 882, 885, 887, 888, 914, 925, 1070, 1089, 1130, 1136, 1137, 1171, 1189, 1264, 1279, 1288, 1295, 1314, 1329, 1333, 1359, 1421, 1423, 1424, 1425, 1453, 1457, 1466, 1515, 1679, 1777
c_leaders: 167, 208, 655, 664, 1249, 1250
\cleaders primitive: 1249
c_loc: 1089, 1093
c_node: 1295
cal_expand_ratio: 823, 825, 828, 834, 840, 1066
cal_margin_kern_var: 822
call: 228, 241, 297, 318, 388, 406, 413, 421, 422, 504, 533, 1396, 1399, 1403, 1404, 1405, 1473, 1772
call_func: 687, 692, 712, 714, 1537
cancel_boundary: 1207, 1209, 1210, 1211
cancel_glue: 1746
cancel_glue_cont: 1746
cancel_glue_cont_cont: 1746
cancel_glue_end: 1746
cancel_glue_end_end: 1746
cannot \read: 510
car_ret: 225, 250, 364, 369, 953, 956, 957, 959, 960, 961, 964, 1304
carriage_return: 22, 49, 225, 250, 258, 385
case_shift: 226, 1463, 1464, 1465
cat: 363, 376, 377, 378
cat_code: 248, 250, 254, 284, 363, 365, 376, 377, 378, 1517
\catcode primitive: 1408
cat_code_base: 248, 250, 251, 253, 1408, 1409, 1411
cc: 363, 374, 377, 712, 717, 718
cc: 484
change_box: 1154, 1257, 1288, 1548, 1821
change_if_limit: 523, 524, 535
char: 19, 26, 546, 560
\char primitive: 287
char_base: 576, 578, 580, 592, 596, 603, 705, 706, 1500, 1501
char_box: 885, 886, 887, 914
\chardef primitive: 1400
char_def_code: 1400, 1401, 1402
char_depth: 580, 673, 828, 884, 885, 888, 1671
char_depth_end: 580
char_exists: 580, 600, 603, 604, 609, 673, 705, 884, 898, 914, 916, 925, 931, 1213, 1769
char_given: 226, 439, 1112, 1207, 1215, 1268, 1302, 1329, 1332, 1400, 1401, 1402
char_height: 580, 673, 828, 884, 885, 888, 1303, 1671
char_height_end: 580
char_info: 569, 576, 580, 581, 583, 596, 600, 603, 604, 609, 648, 673, 690, 705, 717, 726, 731, 823, 828, 884, 885, 888, 890, 891, 898, 900, 914, 916, 925, 1017, 1018, 1042, 1043, 1046, 1047, 1086, 1213, 1214, 1216, 1218, 1291, 1301, 1303, 1325, 1671, 1724, 1769
char_info_end: 580
char_info_word: 567, 569, 570
char_italic: 580, 885, 890, 925, 931, 1291, 1671
char_italic_end: 580
char_kern: 583, 823, 917, 929, 1086, 1218
char_kern_end: 583
char_map_array: 707
char_move: 725, 726

char_node: 152, 161, 163, 180, 194, 574, 619, 648, 823, 928, 1057, 1084, 1206, 1291, 1316
char_num: 226, 287, 288, 1112, 1207, 1215, 1268, 1302, 1329, 1332
char_pw: 823, 825
char_shrink: 823, 1015
char_stretch: 823, 1015
char_tag: 580, 596, 604, 705, 823, 884, 886, 916, 917, 925, 928, 1086, 1216
char_used_array: 707, 708
char_warning: 608, 609, 726, 731, 898, 1213
char_width: 580, 648, 673, 690, 717, 726, 731, 823, 828, 885, 890, 891, 916, 1017, 1018, 1042, 1043, 1046, 1047, 1301, 1303, 1325, 1671, 1724
char_width_end: 580
character: 152, 161, 162, 192, 194, 224, 609, 648, 674, 705, 731, 822, 823, 825, 828, 857, 858, 859, 863, 867, 885, 891, 898, 900, 925, 928, 929, 1017, 1018, 1042, 1043, 1046, 1047, 1073, 1074, 1075, 1080, 1084, 1085, 1087, 1088, 1209, 1211, 1212, 1213, 1214, 1215, 1218, 1291, 1301, 1303, 1325, 1329, 1333, 1343, 1724, 1733
character set dependencies: 23, 49
check sum: 53, 568, 615
check_byte_range: 596, 600
check_dimensions: 902, 903, 909, 930
check_effective_tail: 1258, 1283
check_existence: 600, 601
check_expand_pars: 823, 1017, 1018, 1042, 1043, 1046, 1047
check_full_save_stack: 295, 296, 298, 302, 1837
check_image_b: 768
check_image_c: 768
check_image_i: 768
check_interrupt: 96, 346, 365, 929, 1088, 1208, 1218
check_mem: 183, 185, 1208, 1519
check_outer_validity: 358, 373, 375, 376, 379, 384, 401
check_pfdoutput: 1537, 1538, 1539, 1540, 1541, 1542, 1544, 1546, 1548, 1549, 1553, 1554, 1558, 1560, 1561, 1563, 1565, 1567, 1568, 1569, 1572, 1574, 1575, 1578, 1579, 1580, 1581, 1582, 1588, 1589, 1590, 1591, 1593, 1594, 1595, 1596, 1597, 1598, 1599
check_pfdversion: 683, 698, 792, 1553
check_shrinkage: 1001, 1003, 1044
checkpdfrestore: 727
checkpdfsave: 727
Chinese characters: 152, 612
choice_node: 864, 865, 866, 874, 906
choose_mlist: 907
chr: 19, 20, 23, 24, 1400
chr_cmd: 320, 957
chr_code: 245, 249, 257, 267, 288, 320, 403, 411, 437, 439, 443, 495, 514, 518, 957, 1161, 1231, 1237, 1249, 1250, 1267, 1286, 1293, 1321, 1335, 1348, 1357, 1367, 1387, 1398, 1401, 1409, 1429, 1433, 1439, 1441, 1451, 1456, 1465, 1467, 1470, 1473, 1526, 1687, 1693, 1698, 1702, 1748, 1771, 1832, 1833, 1861, 1862
clang: 230, 231, 988, 1211, 1269, 1378, 1624, 1625
clean_box: 896, 910, 911, 913, 914, 918, 920, 925, 926, 933, 934, 935
clear_for_error_prompt: 78, 83, 352, 368
clear_terminal: 34, 352, 556, 1518
clobbered: 185, 186, 187
CLOBBERED: 315
close: 28
close_files_and_terminate: 78, 81, 1512, 1513
\closein primitive: 1450
close_noad: 858, 866, 872, 874, 904, 937, 938, 1334, 1335
close_node: 1521, 1524, 1526, 1528, 1603, 1604, 1605, 1620, 1622, 1623
\closeout primitive: 1524
closed: 506, 507, 509, 511, 512, 527, 1453
clr: 913, 919, 921, 922, 932, 933, 934, 935
\clubpenalties primitive: 1864
club_penalties_loc: 248, 1864, 1865
club_penalties_ptr: 1067, 1864
club_penalty: 254, 1067
\clubpenalty primitive: 256
club_penalty_code: 254, 255, 256
cm: 484
cmd: 320, 712, 715, 717, 719, 725, 726, 727, 1400, 1467, 1473, 1832
cmd_length: 712, 714, 717, 719
co_backup: 388
code: 673
Color stack action is missing: 1539
colorspace: 1552
colorstack_current: 695, 727, 1539, 1603
colorstack_data: 695, 1539, 1603, 1604, 1605
colorstack_pop: 695, 727, 1539, 1603
colorstack_push: 695, 727, 1539, 1603
colorstack_set: 695, 727, 1539, 1603
colorstackcurrent: 727
colorstackpop: 727
colorstackpush: 727
colorstackset: 727
colorstackskippagestart: 727
colorstackused: 727, 1539
combine_two_deltas: 1036

comment: 225, 250, 369
common-ending: 15, 524, 526, 535, 823, 836, 842, 843, 844, 850, 853, 854, 1072, 1080, 1435, 1438, 1471, 1472, 1475, 1712
compare_strings: 497, 1537
Completed_box...: 666, 750
compress_trie: 1126, 1129
concat_tokens: 1577, 1578, 1579, 1580, 1581, 1582
cond_math_glue: 167, 207, 908, 1349
cond_ptr: 321, 350, 384, 515, 516, 521, 522, 523, 524, 526, 535, 1515, 1668, 1691, 1773, 1776, 1777
conditional: 388, 391, 524
confusion: 95, 112, 220, 224, 303, 523, 658, 693, 727, 740, 831, 845, 904, 912, 930, 937, 942, 967, 974, 976, 1017, 1018, 1042, 1046, 1047, 1053, 1145, 1150, 1177, 1246, 1258, 1363, 1378, 1389, 1528, 1603, 1604, 1605, 1620, 1637, 1712, 1725, 1730, 1745
continental_point_token: 464, 474
continue: 15, 82, 83, 84, 88, 89, 415, 418, 419, 420, 421, 423, 499, 500, 502, 725, 726, 749, 823, 882, 884, 950, 960, 991, 1005, 1008, 1027, 1073, 1083, 1086, 1087, 1088, 1171, 1178, 1782, 1783
contrib_head: 180, 233, 236, 1165, 1171, 1172, 1175, 1176, 1178, 1194, 1200, 1203
contrib_tail: 1172, 1194, 1200, 1203
contribute: 1171, 1174, 1177, 1179, 1185, 1611
conv_toks: 388, 391, 496
conventions for representing stacks: 322
convert: 228, 388, 391, 494, 495, 496, 1649
convert_to_break_width: 1019
\copy primitive: 1249
copy_code: 1249, 1250, 1257, 1285, 1286, 1288, 1859, 1861
copy_expand_params: 705, 720
copy_font_info: 706
copy_node_list: 179, 221, 222, 224, 1257, 1288, 1635, 1636, 1745
copy_to_cur_active: 1005, 1037
count: 254, 453, 666, 668, 750, 1163, 1185, 1186, 1187
\count primitive: 437
count_base: 254, 257, 260, 1402, 1415
\countdef primitive: 1400
count_def_code: 1400, 1401, 1402
count_do_snap: 1570, 1571, 1637
cp: 822, 1005
cp_skipable: 498, 1005
\cr primitive: 956
cr_code: 956, 957, 965, 967, 968
\crcr primitive: 956
cr_cr_code: 956, 961, 965
cramped: 864, 878
cramped_style: 878, 910, 913, 914
creationdate_given: 807
Creator: 807
creator_given: 807
cs: 712
cs_count: 274, 277, 279, 1496, 1497, 1514
cs_error: 1312, 1313
cs_name: 228, 287, 288, 388, 391
\csname primitive: 287
cs_token_flag: 286, 311, 312, 315, 356, 358, 359, 361, 379, 380, 387, 393, 394, 395, 398, 401, 405, 406, 407, 466, 468, 492, 532, 956, 1223, 1243, 1310, 1393, 1467, 1492, 1618
cur_active_width: 999, 1000, 1005, 1008, 1013, 1019, 1020, 1027, 1028, 1029, 1036, 1846, 1847, 1848, 1849
cur_align: 946, 947, 948, 953, 954, 955, 959, 962, 964, 965, 967, 968, 971, 972, 974
cur_area: 538, 543, 555, 556, 563, 772, 1435, 1438, 1531, 1622
cur_boundary: 292, 293, 294, 296, 304, 350, 384, 1679, 1773, 1774, 1777
cur_box: 1252, 1253, 1254, 1255, 1256, 1257, 1258, 1259, 1260, 1262, 1264, 1265, 1839
cur_break: 997, 1005, 1021, 1027, 1055, 1056, 1057, 1707
cur_c: 898, 899, 900, 914, 925, 928, 929, 931
cur_chr: 88, 286, 318, 319, 321, 354, 359, 363, 365, 370, 371, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 386, 387, 391, 394, 395, 404, 406, 407, 412, 413, 415, 429, 433, 439, 450, 454, 468, 491, 496, 498, 500, 502, 504, 505, 509, 520, 521, 524, 526, 527, 532, 533, 534, 535, 536, 552, 604, 958, 961, 965, 1112, 1114, 1139, 1207, 1211, 1213, 1215, 1223, 1227, 1236, 1238, 1239, 1244, 1251, 1257, 1261, 1268, 1270, 1271, 1279, 1283, 1284, 1288, 1295, 1302, 1306, 1318, 1320, 1329, 1330, 1332, 1333, 1336, 1337, 1338, 1349, 1359, 1369, 1389, 1390, 1391, 1395, 1396, 1399, 1402, 1403, 1404, 1405, 1406, 1410, 1411, 1412, 1415, 1421, 1423, 1424, 1425, 1426, 1430, 1431, 1443, 1453, 1457, 1466, 1471, 1515, 1528, 1530, 1623, 1674, 1688, 1696, 1703, 1749, 1761, 1765, 1772, 1863
cur_cmd: 88, 229, 286, 318, 319, 321, 354, 359, 363, 364, 365, 366, 370, 371, 373, 375, 376, 379, 380, 382, 386, 387, 388, 391, 392, 394, 395, 398, 406, 407, 412, 413, 429, 430, 432, 433, 439, 441, 454, 466, 468, 469, 470, 474, 478, 481, 487, 489, 500, 503, 504, 505, 509, 520, 527, 532, 533, 552, 604, 953, 958, 959, 960, 961,

964, 965, 967, 1112, 1138, 1206, 1207, 1215,
 1223, 1227, 1244, 1256, 1257, 1262, 1273, 1277,
 1302, 1306, 1316, 1329, 1330, 1338, 1343, 1354,
 1355, 1375, 1384, 1389, 1390, 1391, 1399, 1404,
 1405, 1406, 1414, 1415, 1430, 1448, 1623, 1683,
 1703, 1765, 1766, 1767, 1772, 1784
cur_cs: 319, 354, 355, 358, 359, 360, 363, 373,
 375, 376, 378, 379, 380, 387, 394, 395, 398,
 400, 405, 406, 407, 415, 417, 433, 498, 499,
 527, 533, 706, 950, 1223, 1330, 1393, 1396,
 1399, 1402, 1403, 1404, 1435, 1472, 1532, 1537,
 1618, 1683, 1767, 1768
cur_delta_h: 691
cur_dir: 643, 651, 654, 656, 660, 661, 665, 733,
 736, 737, 742, 743, 746, 1705, 1706, 1714,
 1715, 1717, 1720, 1722, 1728, 1730, 1734, 1735,
 1736, 1737, 1738, 1739
cur_ext: 538, 543, 555, 556, 563, 772, 1453,
 1531, 1622
cur_f: 898, 900, 914, 917, 925, 928, 929, 931
cur_fam: 254, 1329, 1333, 1343
cur_fam_code: 254, 255, 256, 1317, 1323
cur_file: 326, 351, 384, 563, 564, 1755
cur_font: 248, 250, 584, 585, 604, 1209, 1211,
 1220, 1222, 1295, 1301, 1302, 1734
cur_font_loc: 248, 250, 251, 252, 1395
cur_font_step: 823, 999, 1003, 1027
cur_g: 647, 653, 657, 662, 729, 735, 738, 744, 1637,
 1638, 1699, 1721, 1722, 1723
cur_glue: 647, 653, 657, 662, 729, 735, 738, 744,
 1637, 1638, 1699, 1721, 1722, 1723
cur_group: 292, 293, 294, 296, 303, 304, 976,
 1240, 1241, 1242, 1243, 1245, 1246, 1247, 1308,
 1309, 1318, 1320, 1369, 1370, 1371, 1372, 1378,
 1661, 1665, 1679, 1777
cur_h: 643, 645, 646, 647, 648, 650, 651, 654, 655,
 656, 657, 660, 661, 665, 691, 692, 693, 725, 726,
 727, 729, 731, 732, 733, 734, 736, 737, 738, 742,
 743, 746, 752, 1621, 1630, 1635, 1637, 1641,
 1642, 1643, 1644, 1646, 1647, 1714, 1716, 1719,
 1720, 1721, 1722, 1724, 1725, 1729
cur_h_offset: 644, 752, 755, 1628
cur_head: 946, 947, 948, 962, 975
cur_height: 1147, 1149, 1150, 1151, 1152, 1153,
 1612
cur_i: 898, 899, 900, 914, 917, 925, 928, 929, 931
cur_if: 321, 358, 515, 516, 521, 522, 1515, 1668,
 1691, 1776, 1777
cur_indent: 1053, 1066
cur_input: 35, 36, 87, 323, 324, 333, 343, 344,
 560, 1309, 1774, 1776
cur_l: 1084, 1085, 1086, 1087, 1088, 1209, 1211,
 1212, 1213, 1214, 1216, 1218
cur_lang: 1068, 1069, 1100, 1101, 1107, 1111, 1116,
 1121, 1140, 1269, 1378, 1609, 1610, 1855, 1858
cur_length: 41, 198, 200, 279, 281, 542, 551,
 645, 727, 868, 1615
cur_level: 292, 293, 294, 296, 299, 300, 302, 303,
 1482, 1515, 1661, 1665, 1679, 1777, 1837, 1839
cur_line: 1053, 1065, 1066, 1067
cur_list: 231, 234, 235, 236, 448, 1422, 1679
cur_loop: 946, 947, 948, 953, 959, 968, 969, 970
cur_mark: 318, 408, 412, 1515, 1809
cur_mlist: 895, 896, 902, 930, 1372, 1374, 1377
cur_mu: 879, 895, 906, 908, 942
cur_name: 538, 543, 555, 556, 563, 772, 1435,
 1436, 1438, 1531, 1622
cur_order: 388, 465, 473, 474, 480, 488
cur_p: 999, 1004, 1005, 1006, 1009, 1013, 1015,
 1016, 1021, 1027, 1028, 1029, 1031, 1032, 1033,
 1034, 1035, 1036, 1038, 1039, 1041, 1042, 1043,
 1044, 1045, 1048, 1053, 1054, 1055, 1056, 1057,
 1071, 1080, 1609, 1707, 1849, 1852
cur_page_height: 644, 693, 727, 752, 755, 769, 780,
 1621, 1628, 1630, 1635, 1641
cur_page_width: 644, 755, 769, 1628
cur_pos: 1637
cur_pre_head: 946, 947, 948, 962, 975
cur_pre_tail: 946, 947, 948, 962, 972, 975
cur_ptr: 412, 441, 453, 1402, 1404, 1405, 1415,
 1815, 1816, 1819, 1820, 1821, 1824, 1825,
 1827, 1830, 1831, 1839
cur_q: 1084, 1085, 1087, 1088, 1211, 1212, 1213,
 1214, 1218
cur_r: 1084, 1085, 1086, 1087, 1088, 1209, 1211,
 1214, 1215, 1216, 1218
cur_rh: 1083, 1085, 1086, 1087
cur_s: 620, 643, 647, 657, 668, 670, 729, 730,
 738, 739, 751, 1635, 1637
cur_size: 876, 877, 879, 895, 898, 899, 908, 912,
 913, 920, 922, 923, 924, 925, 933, 934, 935, 938
cur_span: 946, 947, 948, 963, 972, 974
cur_style: 879, 895, 896, 902, 903, 906, 907, 910,
 911, 913, 914, 918, 920, 921, 922, 924, 925,
 926, 930, 932, 933, 934, 935, 936, 938, 939,
 942, 1372, 1374, 1377
cur_tail: 946, 947, 948, 962, 972, 975
cur_tok: 88, 286, 303, 319, 347, 348, 349, 358, 386,
 387, 388, 392, 393, 394, 395, 398, 401, 405,
 406, 407, 418, 419, 420, 421, 423, 425, 429,
 431, 433, 466, 467, 468, 470, 471, 474, 478,
 500, 502, 503, 505, 509, 520, 529, 532, 959,
 960, 1215, 1223, 1225, 1273, 1305, 1306, 1310,
 1393, 1399, 1446, 1447, 1449, 1618, 1619, 1683,

1761, 1767, 1769, 1772, 1784, 1785
`cur_v:` 643, 646, 647, 651, 652, 656, 657, 659, 660,
 661, 663, 664, 665, 668, 692, 693, 725, 726,
 727, 729, 733, 734, 737, 738, 741, 742, 743,
 745, 746, 752, 1621, 1630, 1637, 1641, 1642,
 1643, 1644, 1646, 1647
`cur_v_offset:` 644, 752, 755, 1628
`cur_val:` 286, 287, 356, 388, 412, 436, 439, 440,
 441, 445, 446, 447, 449, 450, 451, 452, 453, 455,
 456, 457, 459, 460, 461, 462, 463, 464, 465, 466,
 468, 470, 471, 473, 474, 476, 477, 479, 481, 483,
 484, 486, 487, 488, 489, 491, 492, 497, 498, 508,
 517, 527, 529, 530, 535, 579, 604, 605, 606, 607,
 693, 705, 706, 720, 817, 956, 958, 1112, 1154,
 1207, 1215, 1238, 1239, 1251, 1255, 1260, 1277,
 1279, 1281, 1301, 1302, 1329, 1332, 1338, 1339,
 1343, 1360, 1366, 1402, 1403, 1404, 1405, 1406,
 1407, 1410, 1412, 1414, 1415, 1416, 1417, 1418,
 1419, 1421, 1422, 1423, 1424, 1425, 1426, 1431,
 1436, 1437, 1453, 1474, 1524, 1530, 1537, 1539,
 1544, 1546, 1549, 1552, 1554, 1556, 1558, 1563,
 1565, 1566, 1573, 1575, 1585, 1587, 1589, 1593,
 1625, 1651, 1665, 1668, 1671, 1674, 1683, 1688,
 1694, 1696, 1769, 1780, 1782, 1785, 1803, 1804,
 1811, 1819, 1820, 1821, 1824, 1839, 1866
`cur_val_level:` 388, 436, 439, 441, 445, 446, 447,
 449, 450, 453, 455, 456, 465, 475, 477, 481, 487,
 491, 492, 497, 1537, 1674, 1780, 1782
`cur_width:` 1053, 1066
`current_page:` 1157
`current_character_being_worked_on:` 596
`\currentgroup_level primitive:` 1663
`current_group_level_code:` 1663, 1664, 1665
`\currentgroup_type primitive:` 1663
`current_group_type_code:` 1663, 1664, 1665
`\currentifbranch primitive:` 1666
`current_if_branch_code:` 1666, 1667, 1668
`\currentiflevel primitive:` 1666
`current_if_level_code:` 1666, 1667, 1668
`\currentiftype primitive:` 1666
`current_if_type_code:` 1666, 1667, 1668
`cv_backup:` 388
`cvl_backup:` 388
`c1:` 793
`c2:` 793
`d:` 107, 194, 195, 278, 363, 466, 586, 689, 823, 844,
 855, 882, 991, 1006, 1053, 1121, 1147, 1246,
 1264, 1316, 1376, 1683, 1744, 1797, 1799
`d_fixed:` 635, 636
`danger:` 1372, 1373, 1377
`data:` 228, 250, 1395, 1410, 1412
`data structure assumptions:` 179, 182, 222, 643,
 992, 1145, 1158, 1467, 1733
`day:` 254, 259, 645, 792, 1508
`\day primitive:` 256
`day_code:` 254, 255, 256
`dd:` 484
`dd:` 689, 690
`deactivate:` 1005, 1027, 1030
`dead_cycles:` 445, 619, 620, 666, 750, 1189, 1201,
 1202, 1232, 1420, 1424
`\deadcycles primitive:` 442
`debug:` 7, 9, 78, 84, 93, 132, 183, 184, 185,
 190, 1208, 1518
`debug #:` 1518
`debug-help:` 78, 84, 93, 1518
`debugging:` 7, 84, 96, 132, 183, 200, 1208, 1518
`descent_fit:` 993, 1010, 1028, 1029, 1040, 1847, 1848
`decr:` 16, 42, 44, 64, 71, 86, 88, 89, 90, 92, 102,
 124, 138, 139, 141, 193, 195, 218, 219, 223, 235,
 263, 279, 282, 303, 304, 333, 344, 346, 347, 348,
 351, 353, 369, 378, 379, 382, 384, 420, 425, 448,
 455, 468, 496, 503, 509, 520, 535, 560, 564, 594,
 603, 628, 647, 657, 666, 670, 671, 674, 686, 690,
 698, 702, 712, 717, 719, 725, 726, 729, 738, 750,
 793, 892, 893, 979, 984, 1005, 1016, 1034, 1045,
 1059, 1092, 1093, 1107, 1108, 1117, 1121, 1125,
 1142, 1238, 1298, 1305, 1309, 1352, 1364, 1372,
 1422, 1471, 1489, 1515, 1517, 1536, 1635, 1679,
 1683, 1691, 1728, 1733, 1739, 1754, 1756, 1774,
 1775, 1776, 1777, 1780, 1782, 1819, 1821
`def:` 227, 1386, 1387, 1388, 1391, 1396
`\def primitive:` 1386
`def_code:` 227, 439, 1388, 1408, 1409, 1410
`def_family:` 227, 439, 604, 1388, 1408, 1409, 1412
`def_font:` 227, 287, 288, 439, 604, 1388, 1434
`def_ref:` 327, 328, 496, 497, 499, 508, 727, 1137,
 1279, 1396, 1404, 1457, 1466, 1532, 1534, 1537,
 1538, 1539, 1540, 1544, 1548, 1552, 1556, 1558,
 1563, 1565, 1566, 1578, 1579, 1580, 1581, 1582,
 1587, 1589, 1590, 1591, 1599, 1615, 1617, 1683
`def_tounicode:` 1587
`default_code:` 859, 873, 919, 1360
`default_hyphen_char:` 254, 603
`\defaulthyphenchar primitive:` 256
`default_hyphen_char_code:` 254, 255, 256
`default_res:` 1552
`default_rule:` 489
`default_rule_thickness:` 859, 877, 910, 911, 913,
 919, 921, 935
`default_skew_char:` 254, 603
`\defaultskewchar primitive:` 256
`default_skew_char_code:` 254, 255, 256
`defecation:` 624

define: 706, 1255, 1392, 1395, 1396, 1399, 1402,
 1403, 1406, 1410, 1412, 1426, 1435, 1839
defining: 327, 328, 361, 499, 508
del_code: 254, 258, 1338
 $\backslash\text{delcode}$ primitive: 1408
del_code_base: 254, 258, 260, 1408, 1410, 1411
delete_action_ref: 1536, 1563, 1579, 1605
delete_glue_ref: 219, 220, 297, 477, 491, 605, 705,
 908, 978, 992, 1002, 1057, 1153, 1173, 1181,
 1194, 1199, 1278, 1407, 1414, 1417, 1515,
 1605, 1780, 1782, 1790, 1791, 1794, 1803,
 1804, 1821, 1838, 1853
delete_image: 778
delete_last: 1282, 1283
delete_q: 902, 936, 939
delete_sa_ptr: 1819, 1821, 1825
delete_sa_ref: 1821, 1834, 1839, 1840, 1841
delete_token_ref: 218, 220, 297, 346, 497, 764,
 1154, 1156, 1189, 1193, 1515, 1536, 1537, 1552,
 1563, 1587, 1590, 1591, 1599, 1605, 1637, 1826,
 1827, 1828, 1830, 1831, 1838
delete_toks: 756, 763, 764, 772, 778, 789, 804,
 806, 807, 814, 815
deletions_allowed: 76, 77, 84, 85, 98, 358, 368
delim_num: 225, 287, 288, 1224, 1329, 1332, 1338
delim_ptr: 230, 231, 1363, 1369
delimited_code: 1356, 1357, 1360, 1361
delimiter: 863, 872, 938, 1369
 $\backslash\text{delimiter}$ primitive: 287
delimiter_factor: 254, 938
 $\backslash\text{delimiterfactor}$ primitive: 256
delimiter_factor_code: 254, 255, 256
delimiter_shortfall: 265, 938
 $\backslash\text{delimiter shortfall}$ primitive: 266
delimiter_shortfall_code: 265, 266
delim1: 876, 924
delim2: 876, 924
delta: 103, 902, 904, 909, 911, 912, 913, 914, 918,
 919, 921, 922, 923, 924, 925, 926, 930, 931, 932,
 935, 938, 1171, 1185, 1187, 1301, 1303
delta_node: 998, 1006, 1008, 1019, 1020, 1036,
 1037, 1041, 1050, 1051
delta_node_size: 998, 1019, 1020, 1036, 1037, 1041
delta1: 919, 922, 938
delta2: 919, 922, 938
den: 612, 614, 617
denom: 476, 484
denom_style: 878, 920
denominator: 859, 866, 873, 874, 920, 1359, 1363
denom1: 876, 920
denom2: 876, 920
deplorable: 1151, 1182

depth: 489
depth: 153, 154, 156, 157, 158, 202, 205, 206, 489,
 580, 644, 650, 652, 654, 659, 660, 663, 669, 732,
 734, 736, 741, 742, 745, 752, 755, 823, 827, 832,
 844, 846, 855, 864, 880, 882, 885, 889, 903, 906,
 907, 911, 912, 913, 921, 922, 923, 925, 926, 927,
 932, 934, 935, 944, 945, 977, 982, 986, 1005,
 1065, 1150, 1179, 1186, 1187, 1198, 1265, 1278,
 1548, 1552, 1556, 1565, 1630, 1637, 1719, 1745
depth_base: 576, 578, 580, 592, 598, 705, 706,
 1500, 1501
depth_index: 569, 580
depth_offset: 153, 442, 945, 1425
depth_threshold: 199, 200, 216, 251, 254, 868,
 1005, 1519, 1823
dest_name_entry: 698, 793, 1627, 1628
dest_names: 697, 698, 793, 804, 805, 1628
dest_names_size: 697, 698, 1513, 1628
destroy_marks: 1515, 1825, 1831
dests: 804, 1513
destxyz: 695
 $\backslash\text{detokenize}$ primitive: 1686
dig: 54, 64, 65, 67, 102, 478, 686, 702
digit_sensed: 1137, 1138, 1139
 $\backslash\text{dimexpr}$ primitive: 1778
dimen: 265, 453, 1185, 1187
 $\backslash\text{dimen}$ primitive: 437
dimen_base: 238, 254, 265, 266, 267, 268, 269,
 270, 1248, 1323
 $\backslash\text{dimendef}$ primitive: 1400
dimen_def_code: 1400, 1401, 1402
dimen_par: 265, 673
dimen_pars: 265
dimen_val: 436, 437, 439, 441, 442, 443, 444, 446,
 447, 450, 451, 453, 454, 455, 475, 481, 491,
 1415, 1674, 1778, 1779, 1785, 1790, 1792, 1795,
 1798, 1815, 1820, 1823, 1832
dimen_val_limit: 1815, 1821, 1822, 1837, 1841
Dimension too large: 486
direct_always: 497, 693, 695, 1538, 1603
direct_page: 497, 693, 695, 1538, 1603
dirty Pascal: 3, 132, 190, 200, 204, 307, 988, 1511
disc_break: 1053, 1056, 1057, 1058, 1067
disc_group: 291, 1295, 1296, 1297, 1661, 1679
disc_node: 163, 166, 193, 201, 220, 224, 674, 823,
 825, 906, 937, 993, 995, 1005, 1032, 1034, 1042,
 1057, 1091, 1217, 1258
disc_ptr: 1515, 1859, 1863
disc_width: 1015, 1046
discard_or_move: 1145
discretionary: 226, 1268, 1292, 1293, 1294
Discretionary list is too long: 1298

\discretionary primitive: 1292
 Display math...with \$\$: 1375
display_indent: 265, 976, 1316, 1323, 1377, 1744
\displayindent primitive: 266
display_indent_code: 265, 266, 1323
\displaylimits primitive: 1334
display_mlist: 865, 871, 874, 907, 1352
display_style: 864, 870, 907, 1347, 1377
\displaystyle primitive: 1347
\displaywidowpenalties primitive: 1864
display_widow_penalties_loc: 248, 1864, 1865
display_widow_penalties_ptr: 1067, 1864
display_widow_penalty: 254, 990, 1067
\displaywidowpenalty primitive: 256
display_widow_penalty_code: 254, 255, 256
display_width: 265, 1316, 1323, 1377, 1744
\displaywidth primitive: 266
display_width_code: 265, 266, 1323

: 100, 655, 664
divide: 227, 287, 288, 1388, 1413, 1414
\divide primitive: 287
divide_scaled: 687, 689, 690, 692, 693, 792, 834, 840
dist: 643, 983, 1372, 1380, 1704, 1714, 1715, 1745
do_all_eight: 999, 1005, 1008, 1013, 1019, 1020, 1036, 1037, 1040
do_all_six: 999, 1147, 1164
do_annot: 1630, 1639, 1645
do_assignments: 976, 1301, 1384, 1448
do_char: 710, 725, 726
do_dest: 1637, 1639, 1645
do_endv: 1308, 1309
do_extension: 1527, 1528, 1623
do_last_line_fit: 1021, 1022, 1027, 1028, 1031, 1039, 1040, 1842, 1843, 1853
do_link: 1560, 1635, 1645
do_marks: 1154, 1189, 1515, 1825
do_nothing: 16, 34, 57, 58, 84, 193, 220, 297, 366, 379, 497, 564, 595, 636, 638, 639, 650, 659, 687, 732, 741, 793, 825, 845, 868, 904, 909, 937, 1013, 1042, 1076, 1223, 1414, 1552, 1603, 1620, 1637, 1645
do_one_seven_eight: 999, 1016, 1045
do_pdf_font: 801
do_register_command: 1413, 1414
do_seven_eight: 999
do_snapy: 1637, 1639
do_snapy_comp: 1637, 1639
do_subst_font: 822, 823, 825, 828
do_thread: 1637, 1639, 1645
do_vf: 712, 720, 726
do_vf_packet: 721, 725, 726

doing_leaders: 619, 620, 656, 665, 737, 746, 1622, 1630, 1637
done: 15, 47, 53, 220, 303, 304, 333, 406, 415, 423, 466, 471, 474, 479, 484, 499, 500, 502, 508, 509, 520, 552, 556, 557, 563, 586, 593, 603, 642, 666, 668, 669, 698, 702, 706, 750, 751, 802, 874, 902, 914, 916, 936, 937, 950, 953, 991, 1005, 1013, 1039, 1049, 1053, 1057, 1072, 1083, 1086, 1088, 1108, 1137, 1138, 1147, 1151, 1154, 1156, 1171, 1174, 1175, 1182, 1257, 1258, 1259, 1288, 1297, 1299, 1316, 1324, 1389, 1405, 1430, 1513, 1537, 1605, 1679, 1723, 1729, 1736, 1737, 1738, 1761, 1799, 1863
done_with_noad: 902, 903, 904, 909, 930
done_with_node: 902, 903, 906, 907, 930
done1: 15, 185, 186, 415, 425, 425, 474, 478, 499, 500, 750, 804, 914, 917, 950, 959, 991, 1005, 1028, 1053, 1055, 1071, 1073, 1076, 1137, 1142, 1171, 1174, 1177, 1480, 1493, 1513
done2: 15, 185, 187, 474, 484, 485, 499, 504, 950, 960, 991, 1073, 1480, 1494
done3: 15, 991, 1074, 1075
done4: 15, 991, 1076
done5: 15, 991, 1042, 1045
done6: 15
dont_expand: 228, 277, 379, 393
double: 111, 113, 119
Double subscript: 1355
Double superscript: 1355
double_hyphen_demerits: 254, 1035
\doublehyphendemerits primitive: 256
double_hyphen_demerits_code: 254, 255, 256
Doubly free location...: 187
down_ptr: 632, 633, 634, 642
downdate_width: 1036
down1: 612, 613, 634, 636, 637, 640, 641, 643, 719, 726
down2: 612, 621, 637
down3: 612, 637
down4: 612, 637
\dp primitive: 442
dry rot: 95
ds: 712
\dump...only by INITEX: 1515
\dump primitive: 1230
dump_four_ASCII: 1487
dump_hh: 1483, 1496, 1502
dump_int: 1483, 1485, 1487, 1489, 1491, 1493, 1494, 1496, 1498, 1500, 1502, 1504, 1506, 1654
dump_qqqq: 1483, 1487, 1500
dump_wd: 1483, 1489, 1493, 1494, 1498
dumpimagemeta: 1504

dumptounicode: 1504
Duplicate pattern: 1140
dvi_buf: 621, 622, 624, 625, 634, 640, 641
dvi_buf_size: 11, 14, 621, 622, 623, 625, 626, 634, 640, 641, 670
dvi_f: 643, 645, 648, 649
dvi_file: 558, 619, 622, 624, 670
DVI files: 610
dvi_font_def: 629, 649, 671
dvi_four: 627, 629, 637, 645, 652, 661, 668, 670, 1615
dvi_gone: 621, 622, 623, 625, 639
dvi_h: 643, 645, 647, 648, 651, 652, 656, 657, 660, 665
dvi_index: 621, 622, 624
dvi_limit: 621, 622, 623, 625, 626
dvi_offset: 621, 622, 623, 625, 628, 632, 634, 640, 641, 647, 657, 668, 670
dvi_out: 625, 627, 628, 629, 630, 636, 637, 645, 647, 648, 649, 652, 657, 661, 668, 670, 1615
dvi_pop: 628, 647, 657
dvi_ptr: 621, 622, 623, 625, 626, 628, 634, 647, 657, 668, 670
dvi_ship_out: 666, 791
dvi_swap: 625
dvi_v: 643, 645, 647, 651, 656, 657, 660, 665
dvi_x: 691
dvi_y: 691
dw: 823
dyn_used: 135, 138, 139, 140, 141, 182, 667, 1489, 1490
e: 299, 301, 524, 544, 545, 556, 1376, 1389, 1414, 1661, 1662, 1744, 1782, 1839, 1840
easy_line: 995, 1011, 1023, 1024, 1026
ec: 566, 567, 569, 571, 586, 591, 592, 596, 603, 706
\edef primitive: 1386
edge: 647, 651, 654, 657, 663, 729, 733, 736, 738, 745, 1646, 1647
edge_dist: 1719, 1720, 1722, 1729
edge_node: 643, 1719, 1720, 1725, 1736
edge_node_size: 1719
ef: 823
\efcode primitive: 1432
ef_code_base: 173, 452, 1431, 1432, 1433
eight_bits: 25, 64, 130, 319, 575, 586, 597, 604, 608, 609, 622, 634, 673, 680, 686, 690, 696, 704, 705, 707, 712, 725, 823, 882, 885, 888, 1169, 1170, 1466
eject_penalty: 175, 1005, 1007, 1027, 1035, 1049, 1147, 1149, 1151, 1182, 1187, 1188
el_gordo: 111, 112, 114
\pdfelapsetime primitive: 442

elapsed_time_code: 442, 443, 450
else: 10
\else primitive: 517
else_code: 515, 517, 524, 1668
em: 481
Emergency stop: 93
emergency_stretch: 265, 1004, 1039
\emergencystretch primitive: 266
emergency_stretch_code: 265, 266
empty: 16, 447, 857, 861, 863, 868, 898, 899, 914, 925, 927, 928, 930, 931, 932, 1157, 1163, 1164, 1168, 1178, 1185, 1354, 1355, 1364
empty line at end of file: 512, 564
empty_field: 860, 861, 862, 918, 1341, 1343, 1359
empty_flag: 142, 144, 148, 168, 182, 1490
end: 7, 8, 10
End of file on the terminal: 37, 71
(\end occurred...): 1515
\end primitive: 1230
end_cs_name: 226, 287, 288, 398, 1312, 1767
\endcsname primitive: 287
end_diagnostic: 263, 306, 321, 345, 426, 427, 528, 535, 608, 666, 669, 750, 839, 851, 1002, 1039, 1164, 1169, 1183, 1188, 1299, 1476, 1662, 1823
end_file_reading: 351, 352, 382, 384, 509, 563, 1515
end_graf: 1203, 1263, 1272, 1274, 1278, 1309, 1311, 1346
end_group: 226, 287, 288, 1241
\endgroup primitive: 287
\endinput primitive: 402
end_L_code: 165, 1701, 1702, 1705, 1734
end_line_char: 87, 254, 258, 325, 340, 354, 382, 384, 509, 560, 564, 1517
\endlinechar primitive: 256
end_line_char_code: 254, 255, 256
end_line_char_inactive: 382, 384, 509, 564, 1517
end_link: 1635, 1645
end_LR: 165, 210, 1708, 1711, 1717, 1728, 1737, 1739
end_LR_type: 165, 1705, 1708, 1711, 1717, 1728, 1737, 1739
end_M: 1258
end_M_code: 165, 450, 1705, 1746
end_match: 225, 311, 313, 316, 417, 418, 420
end_match_token: 311, 415, 417, 418, 419, 420, 500, 502, 508
end_name: 538, 543, 552, 557
end_of_TEX: 6, 81, 1512
end_R_code: 165, 1701, 1705
end_reflect: 1700
end_span: 180, 944, 955, 969, 973, 977, 979
end_template: 228, 388, 401, 406, 956, 1473, 1772

end_template_token: 956, 960, 966
end_thread: 1637, 1639
end_token_list: 346, 347, 379, 416, 1203, 1515, 1618
end_write: 240, 1616, 1618
\endwrite: 1616
end_write_token: 1618, 1619
endcases: 10
\endL primitive: 1701
\endR primitive: 1701
endtemplate: 956
endv: 225, 320, 401, 406, 944, 956, 958, 967, 1224, 1308, 1309
ensure_dvi_open: 558, 645
ensure_pdf_open: 683, 684
ensure_vbox: 1170, 1186, 1195
eof: 26, 31, 52, 590, 602, 772, 1507
eof_seen: 350, 384, 1660
eln: 31, 52
eop: 610, 612, 613, 615, 668, 670
epdf_orig_x: 498, 1637
epdf_orig_y: 498, 1637
epochseconds: 680, 1517, 1555, 1584, 1586
eq_define: 299, 300, 301, 398, 958, 1248, 1392
eq_destroy: 297, 299, 301, 305
eq_level: 239, 240, 246, 250, 254, 271, 277, 286, 299, 301, 305, 956, 1154, 1493, 1616, 1820, 1821
eq_level_field: 239
eq_no: 226, 1318, 1319, 1321, 1322, 1679
\eqno primitive: 1319
eq_save: 298, 299, 300
eq_type: 228, 239, 240, 241, 246, 250, 271, 277, 286, 287, 289, 299, 301, 373, 375, 376, 379, 380, 398, 415, 417, 956, 1330, 1493, 1616, 1767
eq_type_field: 239, 297
eq_word_define: 300, 301, 1248, 1317, 1323, 1392
eqtb: 2, 133, 181, 238, 239, 240, 241, 242, 246, 248, 250, 254, 258, 260, 265, 268, 269, 270, 271, 273, 275, 284, 286, 287, 288, 289, 290, 292, 294, 296, 297, 298, 299, 300, 301, 303, 304, 305, 306, 307, 308, 311, 313, 319, 320, 327, 329, 354, 355, 376, 415, 439, 440, 499, 517, 574, 579, 706, 956, 990, 1366, 1386, 1400, 1415, 1431, 1435, 1493, 1494, 1495, 1519, 1525, 1649, 1823, 1835
eqtb_size: 238, 265, 268, 270, 271, 272, 1485, 1486, 1494, 1495
equiv: 239, 240, 241, 242, 246, 247, 248, 250, 251, 252, 253, 271, 273, 277, 286, 287, 289, 297, 299, 301, 373, 375, 376, 379, 380, 439, 440, 441, 534, 604, 706, 956, 1330, 1405, 1415, 1467, 1493, 1616, 1658, 1864, 1866
equiv_field: 239, 297, 307, 1834
err_help: 79, 248, 1461, 1462
\errhelp primitive: 248
err_help_loc: 248
\errmessage primitive: 1455
error: 72, 75, 76, 78, 79, 82, 88, 91, 93, 98, 121, 349, 360, 368, 396, 424, 434, 444, 454, 471, 480, 482, 485, 486, 497, 501, 502, 512, 526, 536, 549, 561, 587, 593, 606, 669, 899, 952, 960, 968, 1002, 1113, 1114, 1137, 1138, 1139, 1140, 1153, 1155, 1169, 1181, 1186, 1201, 1204, 1228, 1242, 1244, 1246, 1247, 1258, 1260, 1273, 1277, 1284, 1288, 1298, 1299, 1306, 1307, 1313, 1337, 1344, 1355, 1361, 1370, 1373, 1391, 1403, 1410, 1414, 1415, 1419, 1430, 1437, 1461, 1462, 1471, 1539, 1619, 1656, 1782
error_context_lines: 254, 333
\errorcontextlines primitive: 256
error_context_lines_code: 254, 255, 256
error_count: 76, 77, 82, 86, 1274, 1471
error_line: 11, 14, 54, 58, 328, 333, 337, 338, 339
error_message_issued: 76, 82, 95
error_stop_mode: 72, 73, 74, 82, 83, 93, 98, 686, 1440, 1461, 1471, 1472, 1475, 1507, 1515, 1696
\errorstopmode primitive: 1440
erstat: 27
escape: 225, 250, 366, 1517
escape_char: 254, 258, 261
\escapechar primitive: 256
escape_char_code: 254, 255, 256
escapehex: 497
escapename: 497
escapestring: 497
ETC: 314
etc: 200
eTeX_aux: 230, 231, 233, 234
eTeX_aux_field: 230, 231, 1679
eTeX_banner: 2
etex_convert_base: 494
etex_convert_codes: 494
eTeX_dim: 442, 450, 1669, 1672, 1801
eTeX_enabled: 1656, 1703
eTeX_ex: 202, 296, 299, 300, 304, 348, 562, 608, 653, 666, 735, 750, 1323, 1389, 1390, 1391, 1489, 1490, 1515, 1517, 1652, 1655, 1714, 1715, 1716, 1734
eTeX_expr: 442, 1778, 1779, 1780
eTeX_glue: 442, 450, 1805
eTeX_int: 442, 1649, 1663, 1666, 1801
etex_int_base: 254
etex_int_pars: 254
eTeX_mode: 1648, 1652, 1653, 1654, 1655
eTeX_mu: 442, 1780, 1805
etex_pen_base: 248, 250, 251

etex_pens: 248, 250, 251
eTeX_revision: 2, 498
\iTeXrevision primitive: 1649
eTeX_revision_code: 494, 495, 497, 498, 1649
eTeX_state: 1649, 1654, 1700
eTeX_state_base: 1649, 1701
eTeX_state_code: 254, 1649, 1700
eTeX_states: 2, 254, 1654
eTeX_text_offset: 329
etex_toks: 248
etex_toks_base: 248
eTeX_version: 2, 1651
\iTeXversion primitive: 1649
eTeX_version_code: 442, 1649, 1650, 1651
eTeX_version_string: 2
every_cr: 248, 950, 975
\everycr primitive: 248
every_cr_loc: 248, 249
every_cr_text: 329, 336, 950, 975
every_display: 248, 1323
\everydisplay primitive: 248
every_display_loc: 248, 249
every_display_text: 329, 336, 1323
every_eof: 384, 1658
\everyeof primitive: 1657
every_eof_loc: 248, 329, 1657, 1658
every_eof_text: 329, 336, 384
every_hbox: 248, 1261
\everyhbox primitive: 248
every_hbox_loc: 248, 249
every_hbox_text: 329, 336, 1261
every_job: 248, 1207
\everyjob primitive: 248
every_job_loc: 248, 249
every_job_text: 329, 336, 1207
every_math: 248, 1317
\everymath primitive: 248
every_math_loc: 248, 249
every_math_text: 329, 336, 1317
every_par: 248, 1269
\everypar primitive: 248
every_par_loc: 248, 249, 329, 1404
every_par_text: 329, 336, 1269
every_vbox: 248, 1261, 1345
\everyvbox primitive: 248
every_vbox_loc: 248, 249
every_vbox_text: 329, 336, 1261, 1345
ex: 481
ex_hyphen_penalty: 163, 254, 1045
\exhyphenpenalty primitive: 256
ex_hyphen_penalty_code: 254, 255, 256
ex_ratio: 823

ex_space: 226, 287, 288, 1207, 1268
exactly: 816, 817, 891, 1066, 1154, 1194, 1240, 1379, 1680
exit: 15, 16, 37, 47, 58, 59, 69, 82, 122, 143, 200, 299, 300, 314, 363, 415, 433, 439, 487, 491, 496, 523, 524, 550, 604, 609, 634, 642, 749, 793, 823, 844, 928, 967, 1005, 1072, 1111, 1121, 1125, 1154, 1171, 1189, 1207, 1232, 1257, 1283, 1288, 1291, 1297, 1329, 1337, 1352, 1389, 1414, 1448, 1481, 1515, 1518, 1661, 1772, 1819, 1821
expand: 380, 388, 392, 394, 397, 406, 407, 465, 493, 504, 524, 536, 958, 1682, 1772
expand_after: 228, 287, 288, 388, 391, 1762
\expandafter primitive: 287
expand_depth: 1782
expand_depth_count: 1782
expand_font: 705, 823
expand_font_name: 705
expand_ratio: 705
\expanded primitive: 494
expanded_code: 494, 495, 497
explicit: 173, 674, 893, 1013, 1042, 1044, 1055, 1236, 1291, 1711
expr_a: 1792, 1794
expr_add: 1783, 1784
expr_add_sub: 1792
expr_d: 1796
expr_div: 1783, 1784, 1795, 1796
expr_e_field: 1788, 1789
expr_m: 1795
expr_mult: 1783, 1784, 1795
expr_n_field: 1788, 1789
expr_node_size: 1788, 1789
expr_none: 1783, 1784, 1791, 1792
expr_s: 1798
expr_scale: 1783, 1795, 1798
expr_sub: 1783, 1784, 1790, 1792
expr_t_field: 1788, 1789
ext_bot: 572, 889, 890
ext_delimiter: 539, 541, 542, 543
ext_mid: 572, 889, 890
ext_rep: 572, 889, 890
ext_tag: 570, 595, 604, 705, 884, 886
ext_top: 572, 889, 890
ext_xn_over_d: 823, 1552, 1637
exten: 570
exten_base: 576, 578, 592, 600, 601, 603, 705, 706, 889, 1500, 1501
extensible_recipe: 567, 572
extension: 226, 1524, 1526, 1527, 1623
extensions to TeX: 2, 164, 1520
Extra \else: 536

Extra \endcsname: 1313
 Extra \fi: 536
 Extra \middle.: 1370
 Extra \or: 526, 536
 Extra \right.: 1370
 Extra }, or forgotten x: 1247
 Extra alignment tab...: 968
 Extra x: 1244
 extra_info: 945, 964, 965, 967, 968
 extra_right_brace: 1246, 1247
 extra_space: 573, 584, 1222
 extra_space_code: 573, 584
 eyes and mouth: 354
 e1: 793
 e2: 793
 f: 27, 28, 31, 112, 114, 162, 474, 551, 586, 604,
 605, 608, 609, 619, 629, 706, 720, 725, 823, 882,
 885, 887, 888, 891, 892, 893, 914, 1006, 1038,
 1246, 1291, 1301, 1316, 1389, 1435, 1782, 1799
 false: 27, 31, 37, 45, 46, 47, 51, 76, 80, 88, 89, 98,
 106, 107, 112, 115, 184, 185, 186, 187, 286, 296,
 303, 306, 321, 333, 345, 349, 350, 353, 358, 368,
 383, 384, 387, 390, 400, 426, 427, 433, 441, 451,
 453, 466, 467, 471, 473, 474, 475, 481, 486, 487,
 488, 491, 498, 511, 527, 528, 531, 533, 535, 538,
 542, 550, 552, 554, 564, 577, 589, 608, 620, 681,
 683, 685, 686, 688, 689, 692, 693, 698, 702, 705,
 706, 720, 726, 727, 749, 753, 769, 775, 793, 794,
 795, 797, 799, 801, 802, 804, 805, 807, 823, 882,
 896, 898, 930, 950, 967, 1002, 1004, 1005, 1013,
 1027, 1030, 1039, 1057, 1063, 1080, 1083, 1087,
 1088, 1128, 1131, 1137, 1138, 1139, 1140, 1143,
 1145, 1164, 1167, 1183, 1188, 1197, 1198, 1203,
 1208, 1210, 1211, 1212, 1218, 1229, 1232, 1239,
 1258, 1274, 1279, 1345, 1360, 1361, 1369, 1370,
 1372, 1377, 1404, 1405, 1414, 1436, 1448, 1457,
 1460, 1461, 1466, 1481, 1503, 1516, 1522, 1523,
 1532, 1534, 1537, 1552, 1555, 1564, 1565, 1578,
 1579, 1581, 1582, 1600, 1618, 1622, 1623, 1629,
 1630, 1639, 1645, 1648, 1656, 1662, 1682, 1756,
 1769, 1774, 1776, 1782, 1793, 1797, 1799, 1820,
 1821, 1823, 1824, 1843, 1846, 1853, 1855, 1856
 false_bchar: 1209, 1211, 1215
 fam: 857, 858, 859, 863, 867, 898, 899, 928,
 929, 1329, 1333, 1343
 \fam primitive: 256
 fam_fnt: 248, 876, 877, 883, 898, 1373
 fam_in_range: 1329, 1333, 1343
 fast_delete_glue_ref: 219, 220, 1699
 fast_get_avail: 140, 224, 397, 700, 822, 823,
 1211, 1215
 fast_store_new_token: 397, 425, 490, 492
 Fatal format file error: 1481
 fatal_error: 71, 93, 346, 382, 510, 556, 561, 958,
 965, 967, 1309
 fatal_error_stop: 76, 77, 82, 93, 1512, 1513
 fbyte: 590, 594, 597, 602
 Ferguson, Michael John: 2
 fetch: 898, 900, 914, 917, 925, 928, 931
 fetch_box: 446, 497, 531, 1154, 1257, 1288, 1425,
 1474, 1548, 1820
 fetch_effective_tail: 1258, 1259, 1283
 fetch_effective_tail_eTeX: 1258
 fewest_demerits: 1048, 1050, 1051
 ff: 498, 693, 696, 698, 766
 fget: 590, 591, 594, 597, 602
 \fi primitive: 517
 fi_code: 515, 517, 518, 520, 524, 526, 535, 536,
 1668, 1691, 1777
 fi_or_else: 228, 321, 388, 391, 515, 517, 518,
 520, 536, 1471
 fil: 480
 fil: 153, 168, 182, 195, 480, 824, 835, 841, 1379
 fil_code: 1236, 1237, 1238
 fil_glue: 180, 182, 1238
 fil_neg_code: 1236, 1238
 fil_neg_glue: 180, 182, 1238
 File ended while scanning...: 360
 File ended within \read: 512
 file_name_size: 11, 26, 545, 548, 549, 551
 file_offset: 54, 55, 57, 58, 62, 563, 666, 750,
 1458, 1755
 file_opened: 586, 587, 589
 file_warning: 384, 1777
 filename: 712
 fill: 153, 168, 182, 824, 835, 841, 1379
 fill_code: 1236, 1237, 1238
 fill_glue: 180, 182, 1232, 1238
 fill_width: 1842, 1843, 1846
 full: 153, 168, 195, 480, 824, 835, 841, 1379, 1699
 fin_align: 949, 961, 976, 1309
 fin_col: 949, 967, 1309
 fin_mlist: 1352, 1362, 1364, 1369, 1372
 fin_row: 949, 975, 1309
 fin_rule: 647, 650, 654, 657, 659, 663, 729, 732,
 736, 738, 741, 745
 final_cleanup: 1512, 1513, 1515, 1825
 final_end: 6, 35, 353, 1512, 1517
 final_hyphen_demerits: 254, 1035
 \finalhyphendemerits primitive: 256
 final_hyphen_demerits_code: 254, 255, 256
 final_pass: 1004, 1030, 1039, 1049
 final_skip: 695, 1573, 1603, 1637
 find_effective_tail: 450

find_effective_tail_eTeX: 450, 1258
find_font_dimen: 451, 605, 1220, 1431
find_obj: 1555, 1565
find_protchar_left: 821, 1005, 1063
find_protchar_right: 821, 1005, 1057
find_sa_element: 441, 453, 1402, 1404, 1405, 1415, 1816, 1819, 1820, 1821, 1824, 1827, 1830, 1839
fingers: 537
finite_shrink: 1001, 1002
fire_up: 1182, 1189, 1809, 1825, 1828
fire_up_done: 1189, 1825, 1829
fire_up_init: 1189, 1825, 1828
firm_up_the_line: 362, 384, 385, 564
first: 30, 31, 35, 36, 37, 71, 83, 87, 88, 286, 350, 351, 353, 377, 382, 384, 385, 400, 509, 557, 564, 1516, 1756, 1768
first_child: 1137, 1140, 1141, 1855, 1856
first_count: 54, 337, 338, 339
first_fit: 1130, 1134, 1143, 1857
first_indent: 1023, 1025, 1066
first_mark: 408, 409, 1189, 1193, 1809, 1828
\firstmark primitive: 410
first_mark_code: 408, 410, 411, 1809
\firstmarks primitive: 1809
first_p: 999, 1005, 1039
first_text_char: 19, 24
first_width: 1023, 1025, 1026, 1066
fit_class: 1006, 1012, 1021, 1022, 1028, 1029, 1031, 1035, 1847, 1848, 1850, 1851
fitness: 995, 1021, 1035, 1040
fix_date_and_time: 259, 1512, 1517
fix_expand_value: 705
fix_int: 682, 683, 705, 706, 792, 823, 1552, 1575
fix_language: 1211, 1624
fix_pdf_draftmode: 747, 748
fix_pdfoutput: 683, 747, 752, 791, 1513
fix_word: 567, 568, 573, 574, 597
fixed_decimal_digits: 690, 691, 692, 693, 792
fixed_gamma: 680, 683
fixed_gen_tounicode: 691, 801
fixed_image_apply_gamma: 680, 683
fixed_image_gamma: 680, 683
fixed_image_hicolor: 680, 683
fixed_inclusion_copy_font: 683, 691
fixed_pdf_draftmode: 680, 683, 684, 685, 748, 778, 794
fixed_pdf_draftmode_set: 680, 681, 748
fixed_pdf_major_version: 680, 683
fixed_pdf_minor_version: 680, 683
fixed_pdf_objcompresslevel: 680, 683, 698, 748
fixed_pdfoutput: 680, 747, 1513
fixed_pdfoutput_set: 680, 681, 747, 1513

fixed_pk_resolution: 691, 792
fixedi: 705
float: 109, 132, 204, 653, 662, 735, 744, 985, 1638
float_constant: 109, 204, 647, 653, 657, 729, 738, 1301, 1303, 1637
float_cost: 158, 206, 1185, 1278
floating_penalty: 158, 254, 1246, 1278
\floatingpenalty primitive: 256
floating_penalty_code: 254, 255, 256
flush_char: 42, 198, 213, 868, 871
flush_jbig2_page0_objects: 794
flush_list: 141, 218, 346, 398, 422, 433, 727, 764, 765, 977, 1080, 1137, 1274, 1457, 1475, 1615, 1617, 1688, 1753, 1767
flush_math: 894, 952, 1373
flush_node_list: 217, 220, 297, 667, 772, 874, 894, 907, 908, 918, 976, 992, 1055, 1059, 1080, 1095, 1145, 1154, 1169, 1176, 1200, 1203, 1256, 1258, 1283, 1298, 1299, 1384, 1515, 1565, 1623, 1635, 1636, 1727, 1735, 1738, 1743, 1838
flush_str: 497, 705, 706, 718, 726, 727, 769, 772, 807, 1537, 1552, 1555, 1563, 1565, 1587, 1630
flush_string: 44, 286, 563, 792, 1438, 1457, 1508, 1555, 1753
flush_whatsit_node: 772, 783, 786
flushable: 673, 1555
flushable_string: 1435, 1438
fm: 1257, 1258, 1283
fm_entry_ptr: 707, 708
fmem_ptr: 451, 575, 578, 592, 595, 596, 603, 605, 606, 607, 705, 706, 1498, 1499, 1501, 1514
fmt_file: 550, 1483, 1484, 1486, 1507, 1508, 1509, 1517
fnt_def1: 612, 613, 629, 715
fnt_def2: 612
fnt_def3: 612
fnt_def4: 612
fnt_num_0: 612, 613, 649, 719, 726
fnt1: 612, 613, 649, 719, 726
fnt2: 612
fnt3: 612
fnt4: 612
font: 152, 161, 162, 192, 194, 211, 224, 574, 609, 648, 674, 705, 731, 822, 823, 825, 828, 857, 885, 891, 900, 1017, 1018, 1042, 1043, 1046, 1047, 1073, 1074, 1075, 1080, 1085, 1088, 1211, 1215, 1291, 1325, 1724, 1733
font metric files: 565
font parameters: 876, 877
Font x has only...: 606
Font x=xx not loadable...: 587
Font x=xx not loaded...: 593

\font primitive: [287](#)
font_area: [575](#), [578](#), [603](#), [629](#), [630](#), [705](#), [706](#),
[1438](#), [1500](#), [1501](#)
font_base: [11](#), [12](#), [129](#), [152](#), [192](#), [194](#), [240](#), [250](#),
[574](#), [577](#), [629](#), [649](#), [671](#), [673](#), [674](#), [705](#), [801](#),
[1438](#), [1498](#), [1499](#), [1514](#)
font_bc: [575](#), [578](#), [603](#), [604](#), [609](#), [673](#), [705](#), [706](#),
[884](#), [898](#), [1213](#), [1500](#), [1501](#), [1671](#), [1769](#)
font_bchar: [575](#), [578](#), [603](#), [705](#), [706](#), [1074](#), [1075](#),
[1092](#), [1209](#), [1211](#), [1500](#), [1501](#)
\fontchardp primitive: [1669](#)
font_char_dp_code: [1669](#), [1670](#), [1671](#)
\fontcharht primitive: [1669](#)
font_char_ht_code: [1669](#), [1670](#), [1671](#)
\fontcharic primitive: [1669](#)
font_char_ic_code: [1669](#), [1670](#), [1671](#)
\fontcharwd primitive: [1669](#)
font_char_wd_code: [1669](#), [1670](#), [1671](#)
font_check: [575](#), [594](#), [629](#), [712](#), [714](#), [1500](#), [1501](#)
\fontdimen primitive: [287](#)
font_dsize: [192](#), [498](#), [575](#), [578](#), [594](#), [629](#), [705](#), [706](#),
[712](#), [714](#), [1438](#), [1439](#), [1500](#), [1501](#)
font_ec: [575](#), [578](#), [603](#), [604](#), [609](#), [673](#), [705](#), [706](#),
[884](#), [898](#), [1213](#), [1500](#), [1501](#), [1671](#), [1769](#)
font_expand_ratio: [821](#), [823](#), [825](#), [828](#), [834](#), [840](#)
font_false_bchar: [575](#), [578](#), [603](#), [705](#), [706](#), [1209](#),
[1211](#), [1500](#), [1501](#)
font_glue: [575](#), [578](#), [603](#), [605](#), [705](#), [706](#), [1220](#),
[1500](#), [1501](#)
font_id_base: [240](#), [252](#), [274](#), [441](#), [574](#), [706](#), [1435](#)
font_id_text: [192](#), [252](#), [274](#), [606](#), [705](#), [706](#), [1435](#),
[1500](#)
font_in_short_display: [191](#), [192](#), [211](#), [674](#), [839](#),
[1040](#), [1519](#)
font_index: [574](#), [575](#), [586](#), [823](#), [1083](#), [1209](#), [1389](#)
font_info: [11](#), [451](#), [574](#), [575](#), [576](#), [578](#), [580](#), [583](#),
[584](#), [586](#), [592](#), [595](#), [598](#), [600](#), [601](#), [602](#), [605](#), [607](#),
[705](#), [706](#), [823](#), [876](#), [877](#), [889](#), [917](#), [928](#), [1086](#),
[1209](#), [1216](#), [1220](#), [1389](#), [1431](#), [1498](#), [1499](#), [1519](#)
font_max: [11](#), [129](#), [192](#), [194](#), [574](#), [577](#), [592](#), [674](#),
[705](#), [706](#), [1499](#), [1514](#)
font_mem_size: [11](#), [574](#), [592](#), [607](#), [705](#), [706](#),
[1499](#), [1514](#)
font_name: [192](#), [498](#), [575](#), [578](#), [603](#), [608](#), [629](#),
[630](#), [693](#), [705](#), [706](#), [710](#), [712](#), [713](#), [714](#), [717](#),
[1438](#), [1439](#), [1500](#), [1501](#)
\fontname primitive: [494](#)
font_name_code: [494](#), [495](#), [497](#), [498](#)
font_params: [575](#), [578](#), [603](#), [605](#), [606](#), [607](#), [705](#),
[706](#), [1373](#), [1500](#), [1501](#)
font_ptr: [575](#), [578](#), [592](#), [603](#), [605](#), [671](#), [705](#), [706](#),
[801](#), [1438](#), [1498](#), [1499](#), [1514](#)

font_shrink: [823](#), [825](#), [828](#), [840](#)
font_size: [192](#), [498](#), [575](#), [578](#), [594](#), [629](#), [692](#), [693](#),
[705](#), [706](#), [712](#), [717](#), [726](#), [1438](#), [1439](#), [1500](#), [1501](#)
font_step: [705](#)
font_stretch: [823](#), [825](#), [828](#), [834](#)
font_used: [497](#), [575](#), [577](#), [649](#), [671](#), [692](#), [693](#),
[801](#), [1587](#)
fontnum: [692](#)
FONTx: [706](#), [1435](#)
for accent: [209](#)
Forbidden control sequence...: [360](#)
force_eof: [353](#), [383](#), [384](#), [404](#)
format_area_length: [546](#), [550](#)
format_default_length: [546](#), [548](#), [549](#), [550](#)
format_ext_length: [546](#), [549](#), [550](#)
format_extension: [546](#), [555](#), [1508](#)
format_ident: [35](#), [61](#), [562](#), [1477](#), [1478](#), [1479](#), [1506](#),
[1507](#), [1508](#), [1517](#), [1648](#)
forward: [78](#), [236](#), [303](#), [362](#), [388](#), [435](#), [646](#), [703](#),
[728](#), [868](#), [869](#), [896](#), [950](#), [976](#), [1682](#), [1695](#),
[1752](#), [1781](#), [1786](#), [1810](#)
found: [15](#), [143](#), [146](#), [147](#), [278](#), [281](#), [363](#), [376](#), [378](#),
[415](#), [418](#), [420](#), [474](#), [481](#), [499](#), [501](#), [503](#), [550](#),
[634](#), [636](#), [639](#), [640](#), [641](#), [693](#), [705](#), [817](#), [882](#),
[884](#), [896](#), [1005](#), [1027](#), [1072](#), [1100](#), [1108](#), [1111](#),
[1118](#), [1130](#), [1132](#), [1316](#), [1324](#), [1325](#), [1326](#), [1414](#),
[1415](#), [1679](#), [1683](#), [1733](#), [1738](#), [1739](#), [1782](#), [1783](#),
[1789](#), [1799](#), [1847](#), [1848](#)
found1: [15](#), [693](#), [1072](#), [1079](#), [1480](#), [1493](#), [1679](#),
[1799](#), [1800](#)
found2: [15](#), [1072](#), [1080](#), [1480](#), [1494](#), [1679](#)
four_cases: [710](#), [719](#), [726](#)
four_choices: [131](#)
four_quarters: [131](#), [439](#), [574](#), [575](#), [580](#), [581](#), [586](#),
[712](#), [823](#), [859](#), [860](#), [882](#), [885](#), [888](#), [900](#), [914](#), [925](#),
[1083](#), [1209](#), [1301](#), [1480](#), [1481](#), [1753](#), [1756](#)
fract: [1798](#), [1799](#), [1846](#)
fraction: [110](#), [112](#)
fraction_four: [110](#), [111](#), [116](#), [119](#), [120](#)
fraction_half: [111](#), [116](#), [127](#)
fraction_noad: [110](#), [859](#), [863](#), [866](#), [874](#), [909](#),
[937](#), [1356](#), [1359](#)
fraction_noad_size: [859](#), [874](#), [937](#), [1359](#)
fraction_one: [110](#), [111](#), [112](#), [113](#), [114](#), [124](#), [125](#)
fraction_rule: [880](#), [881](#), [911](#), [923](#)
free: [183](#), [185](#), [186](#), [187](#), [188](#), [189](#)
free_avail: [139](#), [220](#), [222](#), [235](#), [426](#), [478](#), [497](#),
[822](#), [948](#), [1092](#), [1213](#), [1288](#), [1404](#), [1466](#), [1577](#),
[1683](#), [1705](#), [1757](#)
free_node: [148](#), [219](#), [220](#), [297](#), [522](#), [642](#), [823](#), [831](#),
[874](#), [891](#), [897](#), [903](#), [927](#), [929](#), [932](#), [936](#), [948](#), [979](#),
[1036](#), [1037](#), [1041](#), [1080](#), [1087](#), [1154](#), [1196](#), [1198](#),

1199, 1214, 1278, 1288, 1364, 1365, 1379, 1515,
 1536, 1605, 1722, 1725, 1727, 1729, 1738, 1745,
 1756, 1757, 1789, 1821, 1825, 1841
`freeze_page_specs`: 1164, 1178, 1185
`frozen_control_sequence`: 240, 277, 1393, 1492,
 1496, 1497
`frozen_cr`: 240, 361, 956, 1310
`frozen_dont_expand`: 240, 277, 393
`frozen_end_group`: 240, 287, 1243
`frozen_end_template`: 240, 401, 956
`frozen_endv`: 240, 401, 406, 956
`frozen_fi`: 240, 358, 517
`frozen_null_font`: 240, 284, 285, 579
`frozen_primitive`: 240, 277, 394, 466
`frozen_protection`: 240, 1393, 1394
`frozen_relax`: 240, 287, 395, 405
`frozen_right`: 240, 1243, 1366
`fs`: 705, 712, 725
 Fuchs, David Raymond: 2, 610, 618
`\futurelet primitive`: 1397
`g`: 47, 200, 586, 619, 823, 844, 882, 892, 1844
`g_order`: 647, 653, 657, 662, 729, 735, 738, 744,
 1637, 1638, 1699, 1723
`g_sign`: 647, 653, 657, 662, 729, 735, 738, 744,
 1637, 1638, 1699, 1723
`gap_amount`: 1637
`garbage`: 180, 493, 496, 497, 1137, 1361, 1370, 1457
`garbage_warning`: 794
`\gdef primitive`: 1386
`gen_faked_interword_space`: 693, 1628, 1629,
 1639, 1645
`gen_running_link`: 730, 1628, 1629, 1639, 1645
`geq_define`: 301, 958, 1392
`geq_word_define`: 301, 310, 1190, 1392
`get`: 26, 29, 31, 33, 511, 564, 590, 1484
`get_auto_kern`: 173, 705, 1211, 1212, 1216, 1295
`get_avail`: 138, 140, 222, 223, 234, 347, 348, 359,
 361, 393, 394, 397, 398, 478, 499, 508, 609,
 885, 948, 959, 960, 970, 1085, 1088, 1115,
 1242, 1243, 1396, 1404, 1618, 1683, 1688, 1705,
 1727, 1733, 1754, 1767
`get_chardepth`: 673
`get_charheight`: 673
`get_charwidth`: 673
`get_ef_code`: 452, 823
`get_expand_font`: 705
`get_fontbase`: 673
`get_image_group_ref`: 1637
`get_kern`: 823, 825
`get_kn_ac_code`: 452, 705
`get_kn_bc_code`: 452, 705
`get_kn_bs_code`: 452, 705
`get_lp_code`: 452, 823
`get_microinterval`: 450, 1555
`get_next`: 76, 319, 354, 358, 362, 363, 379, 382,
 386, 387, 388, 393, 406, 407, 413, 415, 504, 520,
 527, 533, 816, 1215, 1223, 1304, 1766
`get_next_char`: 793
`get_node`: 143, 149, 154, 157, 162, 163, 165, 169,
 170, 171, 174, 176, 224, 521, 634, 823, 844,
 862, 864, 865, 892, 948, 974, 1019, 1020, 1021,
 1040, 1091, 1186, 1278, 1279, 1341, 1343, 1359,
 1426, 1427, 1529, 1556, 1573, 1604, 1637, 1699,
 1719, 1733, 1754, 1788, 1815, 1820, 1837
`get_nulcs`: 673
`get_nullfont`: 673
`get_nullptr`: 673
`get_obj`: 498, 752, 1555, 1630, 1637
`get_pdf_compress_level`: 673
`get_pdf OMIT_charset`: 673
`get_pdf suppress_ptex_info`: 673
`get_pdf suppress_warning_dup_map`: 673
`get_pdf suppress_warning_page_group`: 673
`get_pk_char_width`: 690
`get_preamble_token`: 958, 959, 960
`get_ptex_use_underscore`: 673, 807
`get_quad`: 673
`get_r_token`: 706, 1393, 1396, 1399, 1402, 1403,
 1435
`get_resname_prefix`: 792
`get_rp_code`: 452, 823
`get_sa_ptr`: 1819, 1825, 1831
`get_sh_bs_code`: 452, 705
`get_slant`: 673
`get_st_bs_code`: 452, 705
`get_strings_started`: 47, 51, 1512
`get_tag_code`: 452, 604
`get_tex_dimen`: 673
`get_tex_int`: 673
`get_token`: 76, 78, 88, 386, 387, 392, 393, 394, 395,
 418, 425, 468, 478, 497, 499, 500, 502, 503, 505,
 509, 958, 1204, 1316, 1393, 1399, 1430, 1446,
 1449, 1472, 1618, 1619, 1683, 1765, 1772
`get_vpos`: 1637
`get_x_height`: 673
`get_x_or_protected`: 961, 967, 1772
`get_x_token`: 386, 388, 398, 406, 407, 428, 430,
 432, 433, 469, 470, 471, 478, 491, 505, 532,
 552, 956, 1112, 1138, 1206, 1207, 1316, 1375,
 1415, 1623, 1767, 1772
`get_x_token_or_active_char`: 532
`getc`: 712, 772
`getcreationdate`: 497
`getfiledump`: 497

getfilemoddate: 497
getfilesize: 497
getllx: 1630, 1635
gettly: 1630, 1635
getmatch: 497
getmd5sum: 497
geturx: 1630, 1635
getury: 1630, 1635
give_err_help: 78, 89, 90, 1462
global: 1392, 1396, 1419, 1839
global definitions: 239, 301, 305, 1840
\global primitive: 1386
global_box_flag: 1249, 1255, 1419, 1681
global_defs: 254, 958, 1392, 1396
\globaldefs primitive: 256
global_defs_code: 254, 255, 256
glue_base: 238, 240, 242, 244, 245, 246, 247, 270, 958
glue_break: 1053, 1057
glue_error: 1790
\glueexpr primitive: 1778
glue_node: 167, 170, 171, 193, 201, 220, 224, 450, 498, 650, 659, 674, 705, 732, 741, 825, 845, 906, 908, 937, 992, 993, 1005, 1013, 1032, 1038, 1042, 1055, 1057, 1076, 1080, 1145, 1149, 1150, 1165, 1173, 1174, 1177, 1284, 1285, 1286, 1325, 1380, 1637, 1726, 1733, 1746
glue_offset: 153, 177, 204
glue_ord: 168, 473, 647, 657, 729, 738, 818, 823, 844, 967, 1637, 1723
glue_order: 153, 154, 177, 203, 204, 647, 657, 729, 738, 833, 834, 840, 848, 849, 852, 945, 972, 977, 983, 985, 986, 987, 1326, 1637, 1723, 1740
glue_par: 242, 942
glue_pars: 242
glue_ptr: 167, 170, 171, 193, 207, 208, 220, 224, 450, 653, 662, 705, 735, 744, 832, 847, 855, 908, 962, 969, 971, 978, 979, 985, 992, 1005, 1014, 1044, 1057, 1146, 1153, 1173, 1178, 1181, 1326, 1638, 1699, 1733, 1746, 1843, 1853
glue_ratio: 109, 128, 131, 153, 204
glue_ref: 228, 246, 297, 958, 1406, 1414
glue_ref_count: 168, 169, 170, 171, 172, 182, 219, 221, 246, 942, 1221, 1238
glue_set: 153, 154, 177, 204, 653, 662, 735, 744, 833, 834, 840, 848, 849, 852, 983, 985, 986, 987, 1326, 1638, 1699, 1740
glue_shrink: 177, 203, 972, 975, 977, 986, 987
\glueshrink primitive: 1801
glue_shrink_code: 1801, 1802, 1804
\glueshrinkorder primitive: 1801
glue_shrink_order_code: 1801, 1802, 1803
glue_sign: 153, 154, 177, 203, 204, 647, 657, 729, 738, 833, 834, 840, 848, 849, 852, 945, 972, 977, 983, 985, 986, 987, 1326, 1637, 1723, 1740
glue_spec_size: 168, 169, 180, 182, 219, 892, 1699
glue_stretch: 177, 203, 972, 975, 977, 986, 987
\gluestretch primitive: 1801
glue_stretch_code: 1801, 1802, 1804
\gluestretchorder primitive: 1801
glue_stretch_order_code: 1801, 1802, 1803
glue_temp: 647, 653, 657, 662, 729, 735, 738, 744, 1637, 1638, 1723
\gluetomu primitive: 1805
glue_to_mu_code: 1805, 1806, 1808
glue_val: 436, 437, 439, 442, 443, 450, 453, 455, 456, 477, 487, 491, 958, 1238, 1406, 1414, 1415, 1416, 1418, 1573, 1778, 1779, 1780, 1782, 1785, 1787, 1791, 1796, 1815, 1823, 1832
glyph_to_unicode: 1587, 1592
goal height: 1163, 1164
goto: 35, 81
gr: 128, 131, 132, 153
group_code: 291, 293, 296, 817, 1314, 1679
group_trace: 296, 304, 1662
group_warning: 304, 1774
groupref: 1637
grp_stack: 304, 350, 353, 384, 1773, 1774, 1777
gsa_def: 1839, 1840
gsa_w_def: 1839, 1840
gubed: 7
 Guibas, Leonidas Ioannis: 2
g1: 1376, 1381
g2: 1376, 1381, 1383, 1637
h: 222, 278, 281, 823, 844, 914, 1106, 1111, 1121, 1125, 1130, 1143, 1147, 1154, 1171, 1264, 1269, 1301, 1615, 1733, 1799
h_offset: 265, 644, 645, 669, 755
\hoffset primitive: 266
h_offset_code: 265, 266
ha: 1069, 1073, 1077, 1080, 1089
half: 100, 882, 912, 913, 914, 921, 922, 925, 926, 1380
half_buf: 621, 622, 623, 625, 626
half_error_line: 11, 14, 333, 337, 338, 339
halfp: 111, 116, 120, 125
halfword: 108, 128, 131, 133, 148, 286, 299, 301, 302, 303, 319, 320, 322, 355, 363, 388, 415, 439, 490, 499, 508, 575, 586, 604, 705, 706, 727, 857, 967, 976, 997, 1005, 1006, 1009, 1023, 1048, 1053, 1069, 1078, 1083, 1084, 1154, 1209, 1257, 1279, 1389, 1421, 1444, 1466, 1615, 1628, 1656, 1683, 1723, 1738, 1814, 1819, 1822, 1839, 1840
halign: 226, 287, 288, 1272, 1308

\halign primitive: 287
handle_right_brace: 1245, 1246
hang_after: 254, 258, 1023, 1025, 1248, 1327
\hangafter primitive: 256
hang_after_code: 254, 255, 256, 1248
hang_indent: 265, 1023, 1024, 1025, 1248, 1327
\hangindent primitive: 266
hang_indent_code: 265, 266, 1248
hanging indentation: 1023
hasfmentry: 801
hash: 252, 274, 276, 278, 279, 281, 1496, 1497
hash_base: 238, 240, 274, 276, 278, 284, 285, 394,
 395, 527, 706, 1223, 1435, 1492, 1496, 1497
hash_brace: 499, 502
hash_is_full: 274, 279
hash_prime: 12, 14, 278, 280, 1485, 1486
hash_size: 12, 14, 240, 279, 280, 1514
hash_used: 274, 277, 279, 1496, 1497
hasspacechar: 693
hb: 1069, 1074, 1075, 1077, 1080
hbadness: 254, 836, 842, 843
\hbadness primitive: 256
hbadness_code: 254, 255, 256
\hbox primitive: 1249
hbox_group: 291, 296, 1261, 1263, 1661, 1679
hc: 1069, 1070, 1073, 1074, 1075, 1077, 1078,
 1096, 1097, 1100, 1107, 1108, 1111, 1114, 1116,
 1137, 1139, 1140, 1142, 1858
hchar: 1082, 1083, 1085, 1086
hd: 823, 828, 882, 884, 885, 888
head: 230, 231, 233, 234, 235, 450, 894, 952, 972,
 975, 981, 988, 990, 992, 1203, 1232, 1258, 1264,
 1269, 1274, 1278, 1283, 1291, 1297, 1299, 1323,
 1337, 1346, 1354, 1359, 1362, 1363, 1365, 1369
head_field: 230, 231, 236
head_for_vmode: 1272, 1273
head_tab: 693, 695, 696, 697, 698, 789, 790, 795,
 796, 797, 798, 799, 800, 801, 802, 803, 1504,
 1505, 1545, 1600
head_tab_max: 695, 696, 697
header: 568
Hedrick, Charles Locke: 3
height: 153, 154, 156, 157, 158, 202, 205, 206, 489,
 498, 580, 644, 650, 652, 654, 657, 659, 660, 663,
 665, 668, 669, 732, 734, 736, 738, 739, 741, 742,
 745, 746, 752, 755, 823, 827, 832, 846, 848, 855,
 880, 882, 885, 887, 889, 903, 906, 911, 912, 913,
 914, 915, 918, 921, 922, 923, 925, 926, 927, 932,
 933, 935, 944, 945, 972, 977, 980, 982, 983, 985,
 986, 987, 1005, 1065, 1146, 1150, 1158, 1163,
 1178, 1179, 1185, 1186, 1187, 1198, 1265, 1278,
 1548, 1552, 1556, 1565, 1630, 1637, 1639, 1745
height: 489
height_base: 576, 578, 580, 592, 598, 705, 706,
 1500, 1501
height_depth: 580, 673, 828, 884, 885, 888,
 1303, 1671
height_index: 569, 580
height_offset: 153, 442, 443, 945, 1425
height_plus_depth: 888, 890
held over for next output: 1163
help_line: 79, 89, 90, 358, 1284, 1390, 1391
help_ptr: 79, 80, 89, 90
help0: 79, 1430, 1471
help1: 79, 93, 95, 310, 434, 454, 480, 512, 526,
 529, 536, 1137, 1138, 1139, 1140, 1244, 1258,
 1277, 1299, 1310, 1313, 1337, 1355, 1370, 1390,
 1391, 1410, 1415, 1421, 1422, 1436, 1461, 1482,
 1656, 1765, 1769, 1784
help2: 72, 79, 88, 89, 94, 95, 121, 310, 368, 399,
 459, 460, 461, 462, 463, 468, 471, 486, 497,
 501, 502, 604, 606, 669, 683, 1113, 1114, 1155,
 1192, 1204, 1225, 1246, 1258, 1260, 1273, 1284,
 1298, 1307, 1344, 1375, 1385, 1403, 1414, 1419,
 1437, 1539, 1619, 1696, 1782, 1811
help3: 72, 79, 98, 358, 422, 441, 472, 505, 952,
 959, 960, 968, 1170, 1186, 1201, 1205, 1256,
 1262, 1288, 1305, 1361, 1373, 1471, 1539
help4: 79, 89, 360, 424, 429, 444, 482, 593, 899,
 1153, 1181, 1228, 1461
help5: 79, 396, 587, 1002, 1242, 1247, 1306,
 1393, 1471
help6: 79, 421, 485, 1306, 1339
Here is how much...: 1514
hex_dig1: 1819
hex_dig2: 1819
hex_dig3: 1819
hex_dig4: 1819, 1821, 1822
hex_to_cur_chr: 374, 377
hex_token: 464, 470
hf: 1069, 1073, 1074, 1075, 1080, 1085, 1086,
 1087, 1088, 1092, 1093
\hfil primitive: 1236
\hfilneg primitive: 1236
\hfill primitive: 1236
hfuzz: 265, 842
\hfuzz primitive: 266
hfuzz_code: 265, 266
hh: 128, 131, 132, 136, 151, 200, 231, 237, 239,
 275, 290, 862, 918, 1341, 1343, 1359, 1364,
 1483, 1484, 1817
hi: 130, 250, 1410, 1754
hi_mem_min: 134, 136, 138, 143, 144, 152, 182,
 183, 185, 186, 189, 190, 194, 315, 667, 1489,

1490, 1514
hi_mem_stat_min: [180](#), [182](#), [1490](#)
hi_mem_stat_usage: [180](#), [182](#)
history: [76](#), [77](#), [82](#), [93](#), [95](#), [263](#), [686](#), [1512](#), [1513](#),
[1515](#), [1774](#), [1776](#), [1777](#)
hlist_node: [153](#), [154](#), [155](#), [156](#), [166](#), [177](#), [193](#), [201](#),
[202](#), [220](#), [224](#), [497](#), [531](#), [643](#), [646](#), [647](#), [650](#), [659](#),
[729](#), [732](#), [741](#), [816](#), [823](#), [825](#), [845](#), [857](#), [983](#), [986](#),
[990](#), [1005](#), [1017](#), [1018](#), [1042](#), [1046](#), [1047](#), [1145](#),
[1150](#), [1170](#), [1177](#), [1252](#), [1258](#), [1265](#), [1288](#), [1325](#),
[1381](#), [1635](#), [1636](#), [1637](#), [1704](#), [1725](#), [1733](#)
hlist_out: [619](#), [642](#), [643](#), [646](#), [647](#), [648](#), [651](#),
[656](#), [657](#), [660](#), [665](#), [666](#), [668](#), [727](#), [729](#), [869](#),
[1620](#), [1699](#), [1726](#)
hlist_stack: [173](#), [821](#), [1005](#)
hlist_stack_level: [821](#), [1005](#)
hlp1: [79](#)
hlp2: [79](#)
hlp3: [79](#)
hlp4: [79](#)
hlp5: [79](#)
hlp6: [79](#)
hmode: [229](#), [236](#), [442](#), [527](#), [962](#), [963](#), [972](#), [975](#),
[1207](#), [1223](#), [1224](#), [1226](#), [1234](#), [1235](#), [1249](#), [1251](#),
[1254](#), [1257](#), [1261](#), [1264](#), [1269](#), [1270](#), [1271](#), [1272](#),
[1274](#), [1275](#), [1287](#), [1288](#), [1290](#), [1294](#), [1295](#), [1297](#),
[1300](#), [1308](#), [1315](#), [1378](#), [1421](#), [1625](#), [1679](#)
hmove: [226](#), [1226](#), [1249](#), [1250](#), [1251](#), [1681](#)
hn: [1069](#), [1074](#), [1075](#), [1076](#), [1079](#), [1089](#), [1090](#),
[1092](#), [1093](#), [1094](#), [1096](#), [1100](#), [1107](#), [1108](#)
ho: [130](#), [253](#), [440](#), [1329](#), [1332](#), [1756](#), [1757](#)
hold_head: [180](#), [328](#), [955](#), [959](#), [960](#), [970](#), [984](#), [1082](#),
[1083](#), [1090](#), [1091](#), [1092](#), [1093](#), [1094](#), [1191](#), [1194](#)
holding_inserts: [254](#), [1191](#)
\holdinginserts primitive: [256](#)
holding_inserts_code: [254](#), [255](#), [256](#)
hpack: [180](#), [254](#), [816](#), [817](#), [818](#), [819](#), [823](#), [834](#), [837](#),
[885](#), [891](#), [896](#), [903](#), [913](#), [924](#), [930](#), [932](#), [972](#),
[975](#), [980](#), [982](#), [1066](#), [1240](#), [1264](#), [1303](#), [1372](#),
[1377](#), [1379](#), [1382](#), [1705](#), [1736](#), [1746](#)
hrule: [226](#), [287](#), [288](#), [489](#), [1224](#), [1234](#), [1262](#),
[1272](#), [1273](#)
\hrule primitive: [287](#)
hsize: [265](#), [1023](#), [1024](#), [1025](#), [1232](#), [1327](#)
\hsize primitive: [266](#)
hsize_code: [265](#), [266](#)
hskip: [226](#), [1235](#), [1236](#), [1237](#), [1256](#), [1268](#)
\hskip primitive: [1236](#)
\hss primitive: [1236](#)
\ht primitive: [442](#)
hu: [1069](#), [1070](#), [1074](#), [1075](#), [1078](#), [1080](#), [1082](#),
[1084](#), [1085](#), [1087](#), [1088](#), [1089](#), [1092](#), [1093](#)
Huge page...: [669](#)
hyf: [1077](#), [1079](#), [1082](#), [1085](#), [1086](#), [1090](#), [1091](#),
[1096](#), [1097](#), [1100](#), [1101](#), [1109](#), [1137](#), [1138](#),
[1139](#), [1140](#), [1142](#)
hyf_bchar: [1069](#), [1074](#), [1075](#), [1080](#)
hyf_char: [1069](#), [1073](#), [1090](#), [1092](#)
hyf_distance: [1097](#), [1098](#), [1099](#), [1101](#), [1120](#), [1121](#),
[1122](#), [1502](#), [1503](#)
hyf_next: [1097](#), [1098](#), [1101](#), [1120](#), [1121](#), [1122](#),
[1502](#), [1503](#)
hyf_node: [1089](#), [1092](#)
hyf_num: [1097](#), [1098](#), [1101](#), [1120](#), [1121](#), [1122](#),
[1502](#), [1503](#)
hypf_codes: [1854](#), [1858](#)
hypf_count: [1103](#), [1105](#), [1117](#), [1502](#), [1503](#), [1514](#)
hypf_data: [227](#), [1388](#), [1428](#), [1429](#), [1430](#)
hypf_index: [1111](#), [1856](#), [1858](#)
hypf_list: [1103](#), [1105](#), [1106](#), [1109](#), [1110](#), [1111](#),
[1117](#), [1118](#), [1502](#), [1503](#)
hypf_pointer: [1102](#), [1103](#), [1104](#), [1106](#), [1111](#)
hypf_root: [1129](#), [1135](#), [1143](#), [1854](#), [1857](#)
hypf_size: [12](#), [1102](#), [1105](#), [1107](#), [1110](#), [1116](#), [1117](#),
[1485](#), [1486](#), [1502](#), [1503](#), [1514](#)
hypf_start: [1502](#), [1503](#), [1854](#), [1857](#), [1858](#)
hypf_word: [1103](#), [1105](#), [1106](#), [1108](#), [1111](#), [1117](#),
[1118](#), [1502](#), [1503](#)
hyphen_char: [173](#), [452](#), [575](#), [578](#), [603](#), [705](#), [706](#),
[1068](#), [1073](#), [1212](#), [1295](#), [1431](#), [1500](#), [1501](#)
\hyphenchar primitive: [1432](#)
hyphen_passed: [1082](#), [1083](#), [1086](#), [1090](#), [1091](#)
hyphen_penalty: [163](#), [254](#), [1045](#)
\hyphenpenalty primitive: [256](#)
hyphen_penalty_code: [254](#), [255](#), [256](#)
hyphenate: [1071](#), [1072](#)
hyphenated: [995](#), [996](#), [1005](#), [1022](#), [1035](#), [1045](#), [1049](#)
Hyphenation trie...: [1502](#)
\hyphenation primitive: [1428](#)
i: [19](#), [125](#), [337](#), [439](#), [496](#), [604](#), [614](#), [699](#), [702](#), [705](#),
[706](#), [712](#), [725](#), [727](#), [729](#), [750](#), [793](#), [823](#), [914](#),
[925](#), [1078](#), [1301](#), [1528](#), [1679](#), [1774](#), [1776](#), [1777](#),
[1815](#), [1819](#), [1821](#), [1825](#), [1837](#)
I can't find file x: [556](#)
I can't find PLAIN...: [550](#)
I can't go on...: [95](#)
I can't read TEX.POOL: [51](#)
I can't write on file x: [556](#)
id: [1564](#)
id_byte: [614](#), [645](#), [670](#)
id_lookup: [278](#), [286](#), [378](#), [400](#), [1768](#)
ident_val: [436](#), [441](#), [491](#), [492](#)
\ifcase primitive: [513](#)
if_case_code: [513](#), [514](#), [527](#), [1765](#)

if_cat_code: 513, 514, 527
\ifcat primitive: 513
\if primitive: 513
if_char_code: 513, 527, 532
if_code: 513, 521, 536
if_cs_code: 1762, 1764, 1767
\ifcsname primitive: 1762
if_cur_ptr_is_null_then_return_or_goto: 1819
if_def_code: 1762, 1764, 1766
\ifdefined primitive: 1762
\ifdim primitive: 513
if_dim_code: 513, 514, 527
\ifeof primitive: 513
if_eof_code: 513, 514, 527
\iffalse primitive: 513
if_false_code: 513, 514, 527
\iffontchar primitive: 1762
if_font_char_code: 1762, 1764, 1769
\ifhbox primitive: 513
if_hbox_code: 513, 514, 527, 531
\ifhmode primitive: 513
if_hmode_code: 513, 514, 527
\ifincsname primitive: 1762
if_in_csnname_code: 1762, 1764, 1769
\ifinner primitive: 513
if_inner_code: 513, 514, 527
\ifnum primitive: 513
if_int_code: 513, 514, 527, 529
if_limit: 515, 516, 521, 522, 523, 524, 536, 1668, 1691, 1777
if_line: 321, 515, 516, 521, 522, 1515, 1691, 1776, 1777
if_line_field: 515, 521, 522, 1515, 1691, 1777
\ifmmode primitive: 513
if_mmode_code: 513, 514, 527
if_node_size: 515, 521, 522, 1515
\ifodd primitive: 513
if_odd_code: 513, 514, 527
\ifpdfabsdim primitive: 1762
if_pdfabs_dim_code: 1762, 1764, 1769
\ifpdfabsnum primitive: 1762
if_pdfabs_num_code: 1762, 1764, 1769
if_pdfprimitive: 1762
\ifpdfprimitive primitive: 513
if_pdfprimitive_code: 513, 514, 527
if_stack: 350, 353, 384, 522, 1773, 1776, 1777
if_test: 228, 321, 358, 388, 391, 513, 514, 520, 524, 529, 1515, 1691, 1762, 1765, 1769, 1776, 1777
\iftrue primitive: 513
if_true_code: 513, 514, 527
\ifvbox primitive: 513
if_vbox_code: 513, 514, 527
\ifvmode primitive: 513
if_vmode_code: 513, 514, 527
\ifvoid primitive: 513
if_void_code: 513, 514, 527, 531
if_warning: 522, 1776
\ifx primitive: 513
ifx_code: 513, 514, 527
ignore: 225, 250, 354, 367
ignore_depth: 230, 233, 1064
ignore_spaces: 226, 277, 287, 288, 394, 439, 1223
\ignorespaces primitive: 287
Illegal magnification...: 310, 1436
Illegal math \disc...: 1298
Illegal parameter number...: 505
Illegal unit of measure: 480, 482, 485
image: 1552, 1637
image_colordepth: 1552
image_height: 498, 1552, 1637
image_orig_x: 1628
image_orig_y: 1628
image_pages: 1552
image_rotate: 1552, 1637
image_width: 498, 1552, 1637
image_x_res: 1552
image_y_res: 1552
img_h: 1637
img_w: 1637
\immediate primitive: 1524
immediate_code: 1524, 1526, 1528
IMPOSSIBLE: 284
Improper \beginL: 1703
Improper \beginR: 1703
Improper \endL: 1703
Improper \endR: 1703
Improper \halign...: 952
Improper \hyphenation...: 1113
Improper \prevdepth: 444
Improper \setbox: 1419
Improper \spacefactor: 444
Improper ‘at’ size...: 1437
Improper alphabetic constant: 468
Improper discretionary list: 1299
in: 484
in_open: 304, 326, 335, 350, 351, 353, 384, 522, 1774, 1776, 1777
in_state_record: 322, 323
in_stream: 226, 1450, 1451, 1452
inaccessible: 1394
Incompatible glue units: 434
Incompatible list...: 1288
Incompatible magnification: 310

incompleat_noad: 230, 231, 894, 952, 1314, 1356, 1359, 1360, 1362, 1363
Incomplete \if...: 358
incr: 16, 31, 37, 42, 43, 45, 46, 53, 58, 59, 60, 65, 67, 70, 71, 82, 90, 98, 113, 138, 140, 170, 171, 188, 200, 221, 234, 279, 296, 298, 302, 316, 321, 333, 334, 343, 347, 348, 350, 365, 369, 374, 376, 377, 378, 379, 382, 384, 400, 418, 421, 423, 425, 426, 429, 433, 468, 478, 480, 490, 501, 502, 503, 520, 543, 545, 550, 557, 563, 607, 625, 647, 657, 668, 670, 674, 680, 686, 689, 693, 698, 699, 702, 705, 706, 715, 719, 720, 725, 726, 727, 729, 738, 749, 751, 788, 793, 802, 803, 805, 817, 823, 890, 974, 1021, 1053, 1074, 1075, 1087, 1088, 1091, 1092, 1100, 1107, 1108, 1114, 1116, 1117, 1118, 1121, 1131, 1133, 1139, 1140, 1141, 1163, 1199, 1202, 1212, 1216, 1247, 1295, 1297, 1299, 1305, 1320, 1331, 1350, 1352, 1493, 1494, 1496, 1517, 1536, 1537, 1544, 1548, 1552, 1562, 1587, 1635, 1637, 1648, 1668, 1679, 1683, 1691, 1711, 1717, 1728, 1739, 1754, 1755, 1761, 1768, 1782, 1797, 1800, 1819, 1821, 1837
\indent primitive: 1266
indent_in_hmode: 1266, 1270, 1271
indented: 1269
index: 322, 324, 325, 326, 329, 335, 350, 351, 353, 384
index_field: 322, 324, 1309, 1775
index_node_size: 1815, 1821, 1825
inf: 473, 474, 479
inf_bad: 108, 175, 1027, 1028, 1029, 1032, 1039, 1151, 1182, 1194, 1847
inf_dest_names_size: 695, 697
inf_obj_tab_size: 695, 697
inf_pdf_mem_size: 675, 677
inf_pdf_os_buf_size: 679, 681
inf_penalty: 175, 937, 943, 992, 1005, 1007, 1151, 1182, 1190, 1381, 1383
inf_pk_dpi: 695
Infinite glue shrinkage...: 1002, 1153, 1181, 1186
infinity: 471, 1790, 1792, 1798
info: 136, 142, 144, 158, 159, 173, 182, 190, 218, 251, 297, 313, 315, 347, 348, 359, 361, 379, 380, 393, 394, 397, 400, 415, 417, 418, 419, 420, 423, 426, 449, 478, 492, 504, 534, 632, 635, 636, 637, 638, 639, 640, 641, 642, 693, 695, 700, 766, 767, 771, 773, 775, 779, 781, 782, 783, 784, 786, 857, 865, 868, 869, 874, 896, 910, 911, 912, 913, 914, 918, 925, 930, 944, 945, 948, 955, 959, 960, 966, 969, 970, 973, 974, 977, 979, 997, 1023, 1024, 1102, 1109, 1115, 1158, 1243, 1254, 1271, 1327, 1329, 1346, 1359, 1363, 1364, 1369, 1396, 1404, 1426, 1427, 1467, 1473, 1490, 1519, 1521, 1618, 1636, 1637, 1674, 1705, 1707, 1708, 1709, 1711, 1712, 1717, 1718, 1723, 1728, 1731, 1737, 1739, 1750, 1754, 1756, 1757, 1768, 1772, 1815, 1819, 1820, 1824, 1825
init: 8, 47, 50, 149, 286, 1068, 1111, 1119, 1120, 1124, 1127, 1430, 1480, 1503, 1512, 1515, 1516, 1648, 1831
init_align: 949, 950, 1308
init_col: 949, 961, 964, 967
init_cur_lang: 992, 1068, 1069
init_font_base: 705
init_lhyf: 992, 1068, 1069
init_lft: 1077, 1080, 1082, 1085
init_lig: 1077, 1080, 1082, 1085
init_list: 1077, 1080, 1082, 1085
init_math: 1315, 1316, 1705
init_pdf_output: 687, 688, 750
init_pool_ptr: 39, 42, 1488, 1512, 1514
init_prim: 1512, 1516
init_r_hyf: 992, 1068, 1069
init_randoms: 125, 1517, 1585
init_row: 949, 961, 962
init_span: 949, 962, 963, 967
init_start_time: 1584
init_str_ptr: 39, 43, 543, 1488, 1512, 1514
init_terminal: 37, 353
init_trie: 1068, 1143, 1502
INITEX: 8, 11, 12, 47, 50, 134, 1477, 1511, 1825, 1831
initialize: 4, 1512, 1517
inner loop: 31, 112, 113, 116, 130, 138, 139, 140, 141, 143, 145, 146, 148, 220, 346, 347, 363, 364, 365, 379, 387, 406, 425, 433, 580, 624, 638, 648, 825, 828, 829, 1008, 1011, 1027, 1028, 1043, 1207, 1211, 1212, 1213, 1216, 1219
inner_noad: 858, 859, 866, 872, 874, 909, 937, 940, 1334, 1335, 1369
input: 228, 388, 391, 402, 403, 1747
\input primitive: 402
input_file: 326
\inputlineno primitive: 442
input_line_no_code: 442, 443, 450
input_ln: 30, 31, 37, 58, 71, 384, 511, 512, 564
input_ptr: 323, 333, 334, 343, 344, 352, 353, 382, 560, 1309, 1515, 1774, 1776
input_stack: 84, 85, 323, 333, 343, 344, 560, 1309, 1774, 1775, 1776
ins_disc: 1209, 1210, 1212
ins_error: 349, 358, 421, 1225, 1305, 1310, 1393
ins_list: 345, 361, 493, 496, 497, 1242, 1618

ins_node: 158, 166, 193, 201, 220, 224, 819, 825, 906, 937, 1005, 1042, 1076, 1145, 1150, 1158, 1163, 1177, 1191, 1278
ins_node_size: 158, 220, 224, 1199, 1278
ins_ptr: 158, 206, 220, 224, 1187, 1197, 1198, 1278
ins_the_toks: 388, 391, 493
insert: 226, 287, 288, 1275
insert>: 87
\insert primitive: 287
insert_before_tail: 232, 1217
insert_dollar_sign: 1223, 1225
insert_group: 291, 1246, 1277, 1278, 1661, 1679
insert_penalties: 445, 1159, 1167, 1182, 1185, 1187, 1191, 1199, 1203, 1420, 1424
\insertpenalties primitive: 442
insert_relax: 404, 405, 536
insert_token: 290, 302, 304
inserted: 329, 336, 345, 346, 349, 405, 1273
inserting: 1158, 1186
Insertions can only...: 1170
inserts_only: 1157, 1164, 1185
int: 128, 131, 132, 158, 160, 175, 204, 231, 237, 254, 258, 260, 296, 300, 301, 439, 440, 515, 632, 695, 710, 901, 945, 948, 995, 1415, 1426, 1483, 1484, 1486, 1494, 1649, 1788, 1820
int_base: 238, 248, 250, 254, 256, 257, 258, 260, 270, 271, 272, 290, 305, 310, 1190, 1248, 1317, 1323, 1493, 1649, 1657
int_error: 91, 310, 459, 460, 461, 462, 463, 497, 683, 1421, 1422, 1436, 1696, 1811
int_par: 254, 673
int_pars: 254
int_val: 436, 437, 439, 440, 442, 443, 444, 445, 448, 449, 450, 452, 453, 454, 455, 465, 466, 475, 487, 491, 497, 1402, 1414, 1415, 1416, 1418, 1489, 1490, 1537, 1778, 1779, 1780, 1783, 1785, 1790, 1792, 1795, 1798, 1815, 1816, 1818, 1823, 1832
integer: 3, 13, 19, 45, 47, 54, 59, 60, 63, 66, 67, 69, 82, 91, 94, 96, 100, 101, 102, 105, 106, 107, 108, 109, 110, 112, 114, 117, 119, 122, 124, 125, 126, 127, 128, 131, 135, 143, 176, 181, 190, 191, 192, 194, 195, 196, 199, 200, 229, 230, 236, 243, 255, 264, 265, 274, 278, 281, 284, 286, 300, 301, 308, 314, 320, 321, 326, 330, 331, 333, 337, 388, 436, 439, 466, 474, 476, 496, 508, 515, 519, 520, 524, 544, 545, 549, 575, 576, 586, 597, 604, 605, 608, 619, 622, 627, 628, 634, 642, 643, 647, 657, 666, 673, 674, 676, 678, 680, 682, 686, 687, 689, 690, 691, 692, 693, 694, 696, 698, 700, 701, 702, 705, 706, 707, 708, 710, 712, 720, 725, 727, 750, 772, 774, 778, 793, 795, 797, 817, 818, 821, 823, 837, 867, 870, 875, 882, 892, 893, 902, 914, 928, 940, 999, 1004, 1005, 1006, 1009, 1048, 1053, 1069, 1089, 1099, 1143, 1147, 1157, 1159, 1171, 1189, 1207, 1209, 1246, 1253, 1257, 1262, 1269, 1295, 1297, 1316, 1329, 1333, 1372, 1389, 1414, 1471, 1480, 1481, 1505, 1511, 1513, 1518, 1528, 1543, 1545, 1547, 1550, 1552, 1555, 1556, 1557, 1559, 1562, 1564, 1570, 1583, 1600, 1615, 1617, 1627, 1628, 1630, 1632, 1637, 1679, 1705, 1744, 1753, 1756, 1777, 1782, 1793, 1797, 1799, 1839, 1840
\interlinepenalties primitive: 1864
inter_line_penalties_loc: 248, 1248, 1864, 1865
inter_line_penalties_ptr: 1067, 1248, 1864
inter_line_penalty: 254, 1067
\interlinepenalty primitive: 256
inter_line_penalty_code: 254, 255, 256
interaction: 71, 72, 73, 74, 75, 82, 83, 84, 86, 90, 92, 93, 98, 382, 385, 510, 556, 686, 1443, 1461, 1471, 1472, 1475, 1506, 1507, 1508, 1515, 1694
\interactionmode primitive: 1692
internal_font_number: 192, 574, 575, 576, 586, 604, 605, 608, 609, 629, 643, 673, 686, 690, 691, 692, 693, 705, 706, 710, 712, 720, 725, 821, 823, 882, 885, 887, 888, 891, 900, 914, 1006, 1038, 1069, 1209, 1291, 1301, 1316, 1389, 1435, 1587
interrupt: 96, 97, 98, 1208
Interruption: 98
interwoven alignment preambles...: 346, 958, 965, 967, 1309
int0: 694, 695, 1504, 1505
int1: 694, 695, 1504, 1505
int2: 694, 695, 1505
int3: 694, 695, 1504, 1505
int4: 694, 695, 1504, 1505
Invalid code: 1410
Invalid negative color stack number: 1539
invalid_char: 225, 250, 366
invalid_code: 22, 24, 250
is_char_node: 152, 192, 201, 220, 223, 450, 498, 648, 658, 674, 705, 731, 740, 823, 825, 845, 891, 896, 897, 932, 981, 992, 1005, 1013, 1017, 1018, 1042, 1043, 1044, 1046, 1047, 1055, 1057, 1073, 1074, 1076, 1080, 1213, 1217, 1218, 1258, 1283, 1288, 1291, 1299, 1325, 1380, 1637, 1707, 1724, 1733, 1738
is_empty: 142, 145, 187, 188
is_error: 1537
is_hex: 374, 377
is_hex_char: 702
is_hex_string: 702
is_in_csnname: 389, 390, 398, 1767, 1769
is_letterspaced_font: 706
is_names: 804, 805, 1513

is_obj_scheduled: 695, 1630, 1635
is_obj_written: 695, 773, 775, 779, 784, 812, 813, 814
is_pdf_image: 498, 1552, 1637
is_png_image: 1637
is_root: 802, 803, 1513
is_running: 156, 194, 652, 661, 730, 734, 739, 743, 982, 1552, 1630, 1635, 1637
is_shipping_page: 750, 1630, 1635, 1637, 1640, 1641
is_unless: 524
is_valid_char: 604, 673, 705, 717, 726, 731
isscalable: 690, 693
issue_message: 1454, 1457
ital_corr: 226, 287, 288, 1289, 1290
italic correction: 569
italic_base: 576, 578, 580, 592, 598, 705, 706, 1500, 1501
italic_index: 569
its_all_over: 1223, 1232, 1515
i1: 1537
i2: 1537
j: 45, 46, 59, 60, 69, 70, 125, 278, 281, 286, 337, 388, 496, 508, 545, 549, 550, 666, 686, 693, 749, 1070, 1078, 1083, 1111, 1143, 1316, 1389, 1480, 1481, 1528, 1617, 1620, 1656, 1679, 1741, 1744
j_random: 110, 124, 126, 127
Japanese characters: 152, 612
Jensen, Kathleen: 10
jj: 125
job aborted: 382
job aborted, file error...: 556
job_name: 92, 497, 498, 553, 554, 555, 558, 560, 563, 684, 1435, 1508, 1515
\jobname primitive: 494
job_name_code: 494, 496, 497, 498
jump_out: 81, 82, 84, 93
just_box: 990, 1065, 1066, 1326, 1734, 1740
just_copy: 1733, 1734, 1738
just_open: 506, 509, 1453
just_reverse: 1737, 1738
j1: 793, 1537
j2: 793, 1537
k: 45, 46, 47, 64, 65, 67, 69, 71, 102, 119, 124, 125, 181, 278, 281, 286, 363, 385, 433, 476, 490, 545, 549, 551, 556, 560, 586, 614, 624, 629, 634, 666, 686, 693, 702, 705, 706, 712, 823, 881, 1083, 1106, 1111, 1137, 1143, 1257, 1389, 1480, 1481, 1518, 1528, 1545, 1552, 1562, 1615, 1637, 1656, 1815
kern: 226, 571, 1235, 1236, 1237
\kern primitive: 1236
kern_base: 576, 578, 583, 592, 600, 603, 705, 706, 1500, 1501
kern_base_offset: 583, 592, 600, 705
kern_break: 1042
kern_flag: 571, 823, 917, 929, 1086, 1218
kern_node: 173, 174, 201, 220, 224, 450, 650, 659, 674, 705, 732, 741, 825, 845, 897, 906, 908, 937, 1005, 1013, 1017, 1018, 1032, 1042, 1044, 1046, 1047, 1055, 1057, 1073, 1074, 1076, 1145, 1149, 1150, 1153, 1173, 1174, 1177, 1181, 1284, 1285, 1286, 1299, 1325, 1637, 1699, 1711, 1717, 1725, 1728, 1733, 1739
kern_shrink: 823, 825, 1015
kern_stretch: 823, 825, 1015
kk: 476, 478, 749
kn: 705
\knacode primitive: 1432
kn_ac_code_base: 173, 452, 1431, 1432, 1433
\knbccode primitive: 1432
kn_bc_code_base: 173, 452, 1431, 1432, 1433
\knbscode primitive: 1432
kn_bs_code_base: 173, 452, 1431, 1432, 1433
Knuth, Donald Ervin: 2, 86, 869, 989, 1068, 1102, 1174, 1332, 1618, 1679, 1700
kpse_init_prog: 792
kpse_pk_format: 792
kpse_set_program_enabled: 792
kpse_src_compile: 792
l: 47, 127, 278, 281, 286, 298, 303, 314, 321, 337, 520, 523, 560, 628, 642, 823, 844, 1005, 1006, 1078, 1121, 1130, 1137, 1316, 1372, 1414, 1471, 1480, 1518, 1624, 1679, 1723, 1738, 1753, 1777, 1782, 1825
L_code: 165, 193, 210, 1042, 1073, 1076, 1717, 1718, 1736, 1737
l_hyf: 1068, 1069, 1071, 1076, 1079, 1100, 1609, 1610
language: 254, 1111, 1211, 1624
\language primitive: 256
language_code: 254, 255, 256
language_node: 1521, 1603, 1604, 1605, 1609, 1610, 1620, 1624, 1625
large_attempt: 882
large_char: 859, 867, 873, 882, 1338
large_fam: 859, 867, 873, 882, 1338
last: 30, 31, 35, 36, 37, 71, 83, 87, 88, 353, 382, 385, 509, 550, 557, 1756
last_active: 995, 996, 1008, 1011, 1020, 1030, 1036, 1037, 1039, 1040, 1041, 1049, 1050, 1051
last_attr: 1600
last_badness: 450, 818, 820, 823, 836, 840, 843, 844, 850, 852, 854

last_bop: 619, 620, 668, 670
\lastbox primitive: 1249
last_box_code: 1249, 1250, 1257, 1515, 1859, 1861, 1862
last_glue: 450, 1159, 1168, 1173, 1194, 1284, 1515
last_insptr: 1158, 1182, 1185, 1195, 1197
last_item: 226, 439, 442, 443, 1226, 1649, 1663, 1666, 1669, 1672, 1778, 1801, 1805
last_kern: 450, 1159, 1168, 1173
\lastkern primitive: 442
last_leftmost_char: 821, 822, 823, 1063
last_line_fill: 992, 1842, 1843, 1853
last_line_fit: 254, 1842, 1843, 1846
\lastlinefit primitive: 1657
last_line_fit_code: 254, 1657, 1659
last_node_type: 450, 1159, 1168, 1173
\lastnodetype primitive: 1649
last_node_type_code: 442, 450, 1649, 1650
last_nonblank: 31
last_penalty: 450, 1159, 1168, 1173
\lastpenalty primitive: 442
last_pos: 1637
last_rightmost_char: 821, 822, 823, 1057
\lastskip primitive: 442
last_special_line: 1023, 1024, 1025, 1026, 1066
last_text_char: 19, 24
last_thread: 739, 754, 1628, 1637
last_tokens_string: 686, 708, 709, 1563, 1630
latespecial_node: 727, 1521, 1534, 1603, 1604, 1605, 1615, 1620, 1639, 1645
lc: 823
lc_code: 248, 250, 1068, 1139, 1854, 1856, 1857, 1858
\lccode primitive: 1408
lc_code_base: 248, 253, 1408, 1409, 1464, 1465, 1466
leader_box: 647, 654, 656, 657, 663, 665, 729, 736, 737, 738, 745, 746
leader_flag: 1249, 1251, 1256, 1262, 1681
leader_ht: 657, 663, 664, 665, 738, 745, 746
leader_ptr: 167, 170, 171, 208, 220, 224, 654, 663, 736, 745, 832, 847, 992, 1256, 1733
leader_ship: 226, 1249, 1250, 1251, 1681
leader_wd: 647, 654, 655, 656, 729, 736, 737
leaders: 1622
 Leaders not followed by...: 1256
\leaders primitive: 1249
least_cost: 1147, 1151, 1157
least_page_cost: 1157, 1164, 1182, 1183
left: 693
\left primitive: 1366

left_brace: 225, 311, 316, 320, 369, 379, 429, 499, 953, 1241, 1328, 1404
left_brace_limit: 311, 347, 348, 418, 420, 425, 502
left_brace_token: 311, 429, 1305, 1404, 1618
left_delimiter: 859, 872, 873, 913, 924, 1341, 1359, 1360
left_edge: 647, 655, 657, 660, 661, 665, 729, 730, 738, 739, 742, 743, 746, 1639, 1643, 1644, 1645, 1719, 1720, 1722
left_hyphen_min: 254, 1269, 1378, 1624, 1625
\lefthyphenmin primitive: 256
left_hyphen_min_code: 254, 255, 256
\leftmarginkern primitive: 494
left_margin_kern_code: 494, 495, 497, 498
left_noad: 230, 863, 866, 872, 874, 901, 903, 904, 909, 936, 937, 938, 1363, 1366, 1367, 1369, 1679
left_pw: 822, 823, 1005, 1063
left_right: 226, 1224, 1366, 1367, 1368, 1697
left_side: 173, 201, 498, 823, 1063
left_skip: 242, 1003, 1056, 1063, 1740, 1843
\leftskip primitive: 244
left_skip_code: 242, 243, 244, 498, 1063, 1740, 1746
left_to_right: 643, 1706, 1714, 1730, 1735
len: 693
length: 40, 46, 278, 281, 563, 629, 686, 693, 702, 706, 712, 727, 749, 793, 801, 807, 1108, 1118, 1458, 1630
length of lines: 1023
\leqno primitive: 1319
let: 227, 1388, 1397, 1398, 1399
\let primitive: 1397
letter: 225, 250, 284, 311, 313, 316, 320, 369, 376, 378, 1112, 1138, 1206, 1207, 1215, 1268, 1302, 1329, 1332, 1338
letter_space_font: 706
letter_token: 311, 471
letterspace_font: 227, 287, 288, 439, 604, 1388, 1434
\letterspacefont primitive: 287
level: 436, 439, 441, 444, 454, 487, 1780
level_boundary: 290, 292, 296, 304
level_one: 239, 246, 250, 272, 277, 286, 294, 299, 300, 301, 302, 303, 305, 956, 1482, 1515, 1616, 1665, 1820, 1840, 1841
level_zero: 239, 240, 294, 298, 302, 1836
lf: 566, 586, 591, 592, 602, 603, 705, 706, 720
lft_hit: 1083, 1084, 1085, 1087, 1088, 1210, 1212, 1218
lh: 128, 131, 132, 136, 231, 237, 274, 275, 566, 567, 586, 591, 592, 594, 861, 1127, 1817
 Liang, Franklin Mark: 2, 1096
libpdffinish: 794

lig_char: 161, 162, 211, 224, 705, 823, 825, 826, 1017, 1018, 1042, 1046, 1047, 1075, 1080, 1291, 1727, 1733
lig_kern: 570, 571, 575
lig_kern_base: 576, 578, 583, 592, 598, 600, 603, 705, 706, 1500, 1501
lig_kern_command: 567, 571
lig_kern_restart: 583, 823, 917, 928, 1086, 1216
lig_kern_restart_end: 583
lig_kern_start: 583, 823, 917, 928, 1086, 1216
lig_ptr: 161, 162, 193, 211, 220, 224, 823, 1073, 1075, 1080, 1084, 1087, 1088, 1214, 1218, 1727
lig_stack: 1084, 1085, 1087, 1088, 1209, 1211, 1212, 1213, 1214, 1215, 1218
lig_tag: 570, 595, 604, 705, 823, 917, 928, 1086, 1216
lig_trick: 180, 648, 731, 826
ligature_node: 161, 162, 166, 193, 201, 220, 224, 650, 705, 732, 823, 825, 928, 1017, 1018, 1042, 1046, 1047, 1073, 1074, 1076, 1080, 1291, 1299, 1325, 1727, 1733
ligature_present: 1083, 1084, 1085, 1087, 1088, 1210, 1212, 1214, 1218
limit: 322, 324, 325, 329, 340, 350, 352, 353, 365, 370, 372, 373, 374, 376, 377, 378, 382, 384, 385, 509, 512, 563, 564, 1517, 1755, 1761
Limit controls must follow . . .: 1337
limit_field: 35, 87, 322, 324, 560
limit_switch: 226, 1224, 1334, 1335, 1336
limits: 858, 872, 909, 925, 1334, 1335
\limits primitive: 1334
line: 84, 234, 296, 321, 326, 335, 350, 351, 353, 384, 450, 520, 521, 564, 839, 851, 1202, 1755
line_break: 180, 990, 991, 999, 1004, 1015, 1024, 1038, 1039, 1042, 1052, 1071, 1111, 1144, 1147, 1159, 1274, 1323
line_diff: 1048, 1051
line_number: 995, 996, 1009, 1011, 1021, 1022, 1026, 1040, 1048, 1050, 1051
line_penalty: 254, 1035
\linepenalty primitive: 256
line_penalty_code: 254, 255, 256
line_skip: 242, 265
\lineskip primitive: 244
line_skip_code: 167, 170, 242, 243, 244, 855
line_skip_limit: 265, 855
\lineskiplimit primitive: 266
line_skip_limit_code: 265, 266
line_stack: 326, 335, 350, 351
line_width: 1006, 1026, 1027
link: 136, 138, 139, 140, 141, 142, 143, 144, 148, 151, 152, 153, 158, 159, 160, 161, 168, 182, 186, 190, 192, 193, 194, 200, 220, 222, 230, 232, 236, 241, 251, 314, 317, 321, 328, 341, 345, 348, 361, 379, 380, 388, 393, 394, 397, 400, 415, 416, 417, 420, 422, 423, 426, 433, 450, 478, 490, 492, 493, 496, 497, 498, 504, 515, 521, 522, 523, 534, 632, 634, 636, 638, 642, 648, 650, 658, 674, 686, 693, 695, 700, 705, 727, 731, 732, 740, 766, 767, 771, 772, 773, 775, 779, 781, 782, 783, 784, 786, 823, 825, 826, 828, 831, 842, 845, 855, 857, 865, 881, 887, 891, 894, 895, 896, 897, 903, 907, 908, 911, 913, 914, 915, 923, 924, 927, 928, 929, 930, 931, 932, 935, 936, 937, 942, 943, 946, 948, 954, 955, 959, 960, 962, 966, 967, 969, 970, 971, 972, 973, 974, 975, 977, 978, 979, 980, 981, 982, 983, 984, 985, 988, 990, 992, 995, 997, 998, 1005, 1006, 1013, 1016, 1019, 1020, 1021, 1030, 1033, 1034, 1036, 1037, 1038, 1039, 1040, 1041, 1042, 1043, 1045, 1049, 1050, 1051, 1053, 1055, 1056, 1057, 1058, 1059, 1060, 1061, 1062, 1063, 1067, 1071, 1073, 1074, 1075, 1076, 1080, 1082, 1083, 1084, 1085, 1087, 1088, 1090, 1091, 1092, 1093, 1094, 1095, 1109, 1115, 1117, 1137, 1145, 1146, 1147, 1150, 1156, 1157, 1158, 1163, 1165, 1168, 1171, 1175, 1176, 1177, 1178, 1182, 1185, 1186, 1191, 1194, 1195, 1196, 1197, 1198, 1199, 1200, 1203, 1212, 1213, 1214, 1217, 1218, 1219, 1221, 1242, 1243, 1254, 1258, 1264, 1269, 1278, 1279, 1288, 1295, 1297, 1298, 1299, 1301, 1303, 1324, 1333, 1346, 1359, 1362, 1363, 1364, 1365, 1369, 1372, 1374, 1377, 1382, 1383, 1384, 1396, 1404, 1457, 1466, 1473, 1475, 1489, 1490, 1515, 1519, 1521, 1529, 1545, 1565, 1573, 1577, 1615, 1618, 1623, 1636, 1637, 1668, 1683, 1688, 1691, 1705, 1707, 1709, 1712, 1721, 1722, 1724, 1725, 1727, 1729, 1733, 1734, 1735, 1738, 1740, 1745, 1746, 1753, 1754, 1756, 1757, 1768, 1772, 1776, 1777, 1788, 1789, 1815, 1819, 1820, 1821, 1822, 1823, 1824, 1825, 1828, 1837, 1841, 1863
link_node: 730, 1632, 1635, 1636
list_offset: 153, 647, 729, 823, 945, 1195
list_ptr: 153, 154, 202, 220, 224, 498, 647, 651, 657, 660, 729, 733, 738, 742, 823, 834, 839, 840, 844, 849, 852, 885, 887, 891, 897, 915, 923, 927, 983, 1005, 1154, 1156, 1198, 1265, 1278, 1288, 1377, 1733, 1734, 1740, 1745, 1746
list_state_record: 230, 231
list_tag: 570, 595, 596, 604, 705, 884, 916, 925
literal: 693, 726, 727
literal_mode: 693, 727
ll: 1130, 1133
llink: 142, 144, 145, 147, 148, 149, 163, 167, 182, 187, 948, 995, 997, 1490

ln: 686
lo_mem_max: 134, 138, 143, 144, 182, 183, 185, 187, 188, 189, 190, 196, 667, 1489, 1490, 1501, 1514
lo_mem_stat_max: 180, 182, 453, 1399, 1415, 1490, 1832, 1834
load_expand_font: 705
load_fmt_file: 1481, 1517
loc: 36, 37, 87, 322, 324, 325, 329, 334, 336, 340, 341, 345, 347, 348, 350, 352, 353, 365, 370, 372, 373, 374, 376, 378, 379, 380, 382, 384, 393, 394, 416, 509, 550, 563, 564, 1203, 1204, 1517, 1648, 1755, 1761
loc_field: 35, 36, 322, 324, 1309
local_base: 238, 242, 246, 248, 270
location: 632, 634, 639, 640, 641, 642
log_file: 54, 56, 75, 560, 1513
log_name: 558, 560, 1513
log_only: 54, 57, 58, 62, 75, 98, 382, 560, 1508, 1617
log_opened: 92, 93, 553, 554, 560, 561, 1443, 1513, 1514
Logarithm...replaced by 0: 121
\long primitive: 1386
long_call: 228, 297, 388, 413, 415, 418, 425, 1473
long_char: 710, 712, 717
long_help_seen: 1459, 1460, 1461
long_outer_call: 228, 297, 388, 413, 415, 1473
long_state: 361, 413, 417, 418, 421, 422, 425
longinteger: 65, 680, 685, 686, 694, 696, 702
loop: 15, 16
Loose \hbox...: 836
Loose \vbox...: 850
loose_fit: 993, 1010, 1028, 1847
looseness: 254, 1024, 1049, 1051, 1248
\looseness primitive: 256
looseness_code: 254, 255, 256, 1248
\lower primitive: 1249
\lowercase primitive: 1464
lp: 822, 1005
\lpcode primitive: 1432
lp_code_base: 173, 452, 1431, 1432, 1433
lq: 619, 655, 664
lr: 619, 655, 664
LR_box: 230, 231, 1323, 1384, 1742
LR_dir: 1717, 1728, 1737, 1739
LR_problems: 1705, 1706, 1711, 1712, 1713, 1717, 1718, 1723, 1728, 1730, 1735, 1739
LR_ptr: 1053, 1705, 1706, 1707, 1708, 1709, 1711, 1712, 1717, 1718, 1723, 1728, 1730, 1735, 1737, 1739
LR_save: 230, 231, 1053, 1274, 1731

lx: 647, 654, 655, 656, 657, 663, 664, 665, 729, 736, 737, 738, 745, 746
m: 47, 65, 176, 229, 236, 314, 337, 415, 439, 466, 508, 524, 604, 823, 844, 882, 892, 893, 1257, 1283, 1372, 1471, 1518, 1555, 1679, 1723, 1738, 1753
M_code: 165
m_exp: 110
m_log: 110, 119, 121, 127
mac_param: 225, 313, 316, 320, 369, 500, 503, 505, 959, 960, 1223
MacKay, Pierre: 1700
macro: 329, 336, 341, 345, 346, 416
macro_call: 313, 388, 406, 408, 413, 414, 415, 417
macro_def: 499, 503
mag: 254, 258, 310, 483, 612, 614, 615, 617, 645, 670, 690, 758
\mag primitive: 256
mag_code: 254, 255, 256, 310
mag_set: 308, 309, 310
magic_offset: 940, 941, 942
main_control: 286, 1206, 1207, 1209, 1218, 1219, 1230, 1232, 1233, 1234, 1235, 1304, 1312, 1386, 1468, 1512, 1517, 1524, 1527
main_f: 1209, 1211, 1212, 1213, 1214, 1215, 1216, 1218
main_i: 1209, 1213, 1214, 1216, 1218
main_j: 1209, 1216, 1218
main_k: 1209, 1211, 1216, 1218, 1220
main_lig_loop: 1207, 1211, 1214, 1215, 1216, 1218
main_loop: 1207
main_loop_looking: 1207, 1211, 1213, 1214, 1215
main_loop_move: 1207, 1211, 1213, 1217, 1218
main_loop_move_lig: 1207, 1211, 1213, 1214
main_loop_wrapup: 1207, 1211, 1216, 1218
main_p: 1209, 1212, 1214, 1218, 1219, 1220, 1221, 1222
main_s: 1209, 1211
major_tail: 1089, 1091, 1094, 1095
make Accent: 1300, 1301
make_box: 226, 1249, 1250, 1251, 1257, 1262
make_cstring: 792
make font_copy: 706, 1434
make_frac: 110, 112, 127
make_fraction: 110, 909, 910, 919, 1799
make_left_right: 937, 938
make_mark: 1275, 1279
make_math Accent: 909, 914
make_name_string: 551
make_op: 909, 925
make_ord: 909, 928
make_over: 909, 910

make_radical: 909, 910, 913
make_scripts: 930, 932
make_string: 43, 48, 52, 279, 496, 543, 551, 686, 705, 706, 712, 718, 726, 727, 1116, 1435, 1457, 1508, 1513, 1753
make_under: 909, 911
make_vcenter: 909, 912
margin: 1630
margin_char: 173, 220, 224, 823, 825, 1288
margin_kern_node: 173, 201, 220, 224, 498, 650, 732, 823, 825, 1288, 1325
margin_kern_node_size: 173, 220, 224, 823, 1288
margin_kern_shrink: 822, 1005, 1027
margin_kern_stretch: 822, 1005, 1027
mark: 226, 287, 288, 1275, 1809
\mark primitive: 287
mark_class: 159, 214, 1156, 1191, 1279, 1827, 1830
mark_class_node_size: 1820, 1825
mark_node: 159, 166, 193, 201, 220, 224, 819, 825, 906, 937, 1005, 1042, 1076, 1145, 1150, 1156, 1177, 1191, 1279
mark_ptr: 159, 214, 220, 224, 1156, 1193, 1279, 1827, 1830
mark_text: 329, 336, 345, 412
mark_val: 1815, 1816, 1820, 1824, 1827, 1830
\marks primitive: 1809
marks_code: 318, 408, 411, 412, 1809
mastication: 363
match: 225, 311, 313, 314, 316, 417, 418
match_chr: 314, 316, 415, 417, 426
match_token: 311, 417, 418, 419, 420, 502
matching: 327, 328, 361, 417
matchstrings: 497
Math formula deleted...: 1373
math_ac: 1342, 1343
math Accent: 226, 287, 288, 1224, 1342
\mathaccent primitive: 287
\mathbin primitive: 1334
math_char: 857, 868, 896, 898, 900, 914, 917, 925, 928, 929, 930, 1329, 1333, 1343
\mathchar primitive: 287
\mathchardef primitive: 1400
math_char_def_code: 1400, 1401, 1402
math_char_num: 226, 287, 288, 1224, 1329, 1332
math_choice: 226, 287, 288, 1224, 1349
\mathchoice primitive: 287
math_choice_group: 291, 1350, 1351, 1352, 1661, 1679
\mathclose primitive: 1334
math_code: 248, 250, 254, 440, 1329, 1332
\mathcode primitive: 1408

math_code_base: 248, 253, 440, 1408, 1409, 1410, 1411
math_comp: 226, 1224, 1334, 1335, 1336
math_font_base: 248, 250, 252, 1408, 1409
math_fraction: 1358, 1359
math_given: 226, 439, 1224, 1329, 1332, 1400, 1401, 1402
math_glue: 892, 908, 942
math_group: 291, 1314, 1328, 1331, 1364, 1661, 1679
\mathinner primitive: 1334
math_kern: 893, 906
math_left_group: 230, 291, 1243, 1246, 1247, 1328, 1369, 1661, 1679
math_left_right: 1368, 1369
math_limit_switch: 1336, 1337
math_node: 165, 166, 193, 201, 220, 224, 450, 650, 732, 825, 993, 1005, 1013, 1042, 1055, 1057, 1073, 1076, 1258, 1700, 1707, 1728, 1733, 1736, 1738
\mathop primitive: 1334
\mathopen primitive: 1334
\mathord primitive: 1334
\mathpunct primitive: 1334
math_quad: 876, 879, 1377
math Radical: 1340, 1341
\mathrel primitive: 1334
math_shift: 225, 311, 316, 320, 369, 1268, 1315, 1316, 1371, 1375, 1384
math shift_group: 291, 1243, 1246, 1247, 1308, 1317, 1318, 1320, 1323, 1370, 1371, 1372, 1378, 1661, 1679
math shift_token: 311, 1225, 1243
math_spacing: 940, 941
math_style: 226, 1224, 1347, 1348, 1349
math_surround: 265, 1374
\mathsurround primitive: 266
math_surround_code: 265, 266
math_text_char: 857, 928, 929, 930, 931
math_type: 857, 859, 863, 868, 874, 896, 898, 899, 910, 911, 913, 914, 917, 918, 925, 927, 928, 929, 930, 931, 932, 1254, 1271, 1329, 1333, 1343, 1346, 1354, 1359, 1363, 1364, 1369
math_x_height: 876, 913, 933, 934, 935
math_x: 877
mathsy: 876
mathsy_end: 876
matrixrecalculate: 1635
matrixtransformrect: 1630
matrixused: 1630, 1635, 1637
max: 682, 727
max_answer: 105, 1793, 1799

max_buf_stack: 30, 31, 353, 400, 1514, 1756, 1768
max_char_code: 225, 325, 363, 366, 1411
max_command: 227, 228, 229, 237, 380, 388, 392, 394, 406, 407, 504, 958, 1772
max_d: 902, 903, 906, 936, 937, 938
max_dead_cycles: 254, 258, 1189
\maxdeadcycles primitive: 256
max_dead_cycles_code: 254, 255, 256
max_depth: 265, 1157, 1164
\maxdepth primitive: 266
max_depth_code: 265, 266
max_dimen: 447, 486, 669, 844, 1187, 1194, 1323, 1324, 1326, 1637, 1735, 1736, 1737, 1790, 1792, 1798, 1846
max_expand: 705
max_group_code: 291
max_h: 619, 620, 669, 670, 902, 903, 906, 936, 937, 938
max_halfword: 11, 14, 128, 129, 131, 142, 143, 144, 149, 150, 311, 312, 450, 783, 786, 996, 1024, 1026, 1159, 1168, 1173, 1194, 1284, 1427, 1501, 1503, 1515, 1565, 1566, 1636, 1637
max_hlist_stack: 173, 821, 1005
max_in_open: 11, 14, 326, 350, 1660, 1773, 1774, 1776
max_in_stack: 323, 343, 353, 1514
max_integer: 687, 689, 1555
max_internal: 227, 439, 466, 474, 481, 487
max_len: 693
max_nest_stack: 231, 233, 234, 1514
max_non_prefixed_command: 226, 1389, 1448
max_param_stack: 330, 353, 416, 1514
max_print_line: 11, 14, 54, 58, 72, 194, 563, 666, 750, 1458, 1755
max_push: 619, 620, 647, 657, 670
max_quarterword: 11, 128, 129, 131, 173, 296, 973, 974, 1121, 1298, 1503
max_reg_help_line: 1811, 1812, 1813, 1814
max_reg_num: 1811, 1812, 1813, 1814
max_save_stack: 293, 294, 295, 1514
max_selector: 54, 264, 333, 491, 496, 560, 666, 705, 706, 712, 727, 1435, 1457, 1615, 1617, 1753
max_shrink_ratio: 823, 999, 1003, 1027
max_stretch_ratio: 823, 999, 1003, 1027
max_strings: 11, 38, 43, 129, 543, 551, 1488, 1514
max_v: 619, 620, 669, 670
maxdimen: 110
\meaning primitive: 494
meaning_code: 494, 495, 497, 498
med_mu_skip: 242
\medmuskip primitive: 244
med_mu_skip_code: 242, 243, 244, 942

mediabox_given: 750, 769
mem: 11, 12, 133, 134, 136, 142, 144, 149, 151, 152, 153, 158, 160, 168, 169, 175, 177, 180, 181, 182, 183, 185, 190, 200, 204, 221, 223, 224, 239, 242, 297, 313, 413, 446, 515, 632, 695, 826, 856, 857, 859, 862, 863, 896, 901, 918, 929, 945, 946, 948, 973, 992, 994, 995, 998, 999, 1008, 1019, 1020, 1023, 1024, 1026, 1036, 1037, 1066, 1102, 1327, 1329, 1338, 1341, 1343, 1359, 1364, 1425, 1426, 1489, 1490, 1519, 1674, 1727, 1733, 1754, 1756, 1788, 1815, 1820, 1842
mem_bot: 11, 12, 14, 129, 134, 143, 144, 180, 182, 287, 437, 441, 453, 1399, 1404, 1405, 1415, 1485, 1486, 1489, 1490, 1832, 1833, 1834
mem_end: 134, 136, 138, 182, 183, 185, 186, 189, 190, 192, 194, 200, 315, 674, 1489, 1490, 1514
mem_max: 11, 12, 14, 128, 129, 134, 138, 142, 143, 183, 184
mem_min: 11, 12, 129, 134, 138, 143, 183, 184, 185, 187, 188, 189, 190, 192, 196, 200, 674, 1427, 1490, 1514
mem_top: 11, 12, 14, 129, 134, 180, 182, 1427, 1485, 1486, 1490
Memory usage...: 667
memory_word: 128, 131, 132, 134, 200, 230, 236, 239, 271, 290, 293, 297, 574, 575, 710, 976, 1483, 1816
message: 226, 1454, 1455, 1456
\message primitive: 1455
METAFONT: 616
microseconds: 680, 1517, 1555, 1584, 1586
mid: 572
mid_line: 87, 325, 350, 366, 369, 374, 375, 376
middle: 1697
\middle primitive: 1697
middle_noad: 230, 863, 1369, 1370, 1697, 1698
min: 682
min_bp_val: 691, 692, 693, 792
min_font_val: 691
min_halfword: 11, 128, 129, 130, 131, 133, 248, 1204, 1501, 1503, 1534, 1723, 1728, 1738, 1739
min_internal: 226, 439, 466, 474, 481, 487
min_quarterword: 12, 128, 129, 130, 131, 152, 154, 158, 203, 239, 296, 575, 576, 580, 582, 583, 592, 603, 823, 844, 861, 873, 883, 889, 890, 972, 977, 979, 984, 1097, 1100, 1101, 1120, 1121, 1122, 1123, 1135, 1140, 1141, 1142, 1171, 1189, 1501, 1502, 1503
minimal_demerits: 1009, 1010, 1012, 1021, 1031, 1842
minimum_demerits: 1009, 1010, 1011, 1012, 1030, 1031

minor_tail: 1089, 1092, 1093
minus: 488
Misplaced &: 1306
Misplaced \cr: 1306
Misplaced \noalign: 1307
Misplaced \omit: 1307
Misplaced \span: 1306
Missing) inserted: 1784
Missing = inserted: 529, 1769
Missing # inserted...: 959
Missing \$ inserted: 1225, 1243
Missing \cr inserted: 1310
Missing \endcsname...: 399
Missing \endgroup inserted: 1243
Missing \right. inserted: 1243
Missing { inserted: 429, 501, 1305
Missing } inserted: 1243, 1305
Missing ‘to’ inserted: 1260
Missing ‘to’...: 1403
Missing \$\$ inserted: 1385
Missing character: 608
Missing control...: 1393
Missing delimiter...: 1339
Missing font identifier: 604
Missing number...: 441, 472
mkern: 226, 1224, 1235, 1236, 1237
\mkern primitive: 1236
ml_field: 230, 231, 236
mlist: 902, 936
mlist_penalties: 895, 896, 902, 930, 1372, 1374, 1377
mlist_to_hlist: 869, 895, 896, 901, 902, 910, 930, 936, 1372, 1374, 1377
mm: 484
mmode: 229, 230, 231, 236, 527, 894, 951, 952, 976, 983, 988, 1207, 1223, 1224, 1226, 1234, 1235, 1251, 1258, 1270, 1275, 1287, 1288, 1290, 1294, 1298, 1308, 1314, 1318, 1323, 1328, 1332, 1336, 1340, 1342, 1345, 1349, 1353, 1358, 1368, 1371, 1372, 1679, 1742
moddate_given: 807
mode: 229, 230, 231, 233, 234, 321, 444, 448, 450, 527, 727, 894, 951, 952, 961, 962, 963, 972, 975, 980, 983, 984, 985, 988, 1202, 1206, 1207, 1211, 1212, 1227, 1229, 1234, 1254, 1256, 1258, 1261, 1264, 1269, 1271, 1272, 1273, 1274, 1277, 1281, 1283, 1288, 1295, 1297, 1298, 1314, 1316, 1323, 1345, 1372, 1374, 1378, 1421, 1560, 1561, 1615, 1617, 1618, 1625, 1742
mode_field: 230, 231, 236, 448, 976, 983, 1422, 1679, 1681
mode_line: 230, 231, 233, 234, 326, 980, 991, 1202
month: 254, 259, 645, 792, 1508
\month primitive: 256
month_code: 254, 255, 256
months: 560, 562
more_name: 538, 542, 552, 557
\moveleft primitive: 1249
move_past: 647, 650, 653, 657, 659, 662, 729, 732, 735, 738, 741, 744
\overright primitive: 1249
movement: 634, 636, 643
movement_node_size: 632, 634, 642
msg: 712
mskip: 226, 1224, 1235, 1236, 1237
\mskip primitive: 1236
mskip_code: 1236, 1238
mstate: 634, 638, 639
mtype: 4
mu: 473, 474, 475, 479, 481, 487, 488
mu: 482
mu_error: 434, 455, 475, 481, 487, 1780
\muexpr primitive: 1778
mu_glue: 167, 173, 209, 450, 893, 908, 1236, 1238, 1239
mu_mult: 892, 893
mu_skip: 242, 453
\muskip primitive: 437
mu_skip_base: 242, 245, 247, 1402, 1415
\muskipdef primitive: 1400
mu_skip_def_code: 1400, 1401, 1402
\mutogluue primitive: 1805
mu_to_glue_code: 1805, 1806, 1807
mu_val: 436, 437, 439, 450, 453, 455, 456, 475, 477, 481, 487, 491, 1238, 1402, 1406, 1414, 1415, 1778, 1779, 1780, 1787, 1815, 1820, 1823
mu_val_limit: 1815, 1821, 1838
mult_and_add: 105
mult_integers: 105, 1418, 1795
multiply: 227, 287, 288, 1388, 1413, 1414, 1418
\multiply primitive: 287
Must increase the x: 1481
must_end_string: 693
must_insert_space: 693
n: 47, 65, 66, 67, 69, 91, 94, 105, 106, 107, 112, 114, 170, 172, 192, 200, 243, 255, 265, 270, 314, 320, 321, 337, 415, 508, 524, 544, 545, 549, 605, 674, 689, 882, 892, 893, 967, 976, 1083, 1111, 1121, 1154, 1169, 1170, 1171, 1189, 1257, 1297, 1316, 1389, 1453, 1471, 1518, 1723, 1738, 1782, 1797, 1799, 1819, 1822
name: 322, 324, 325, 326, 329, 333, 335, 336, 345, 350, 351, 353, 359, 382, 384, 416, 509, 563, 1755
name_field: 84, 85, 322, 324, 1774, 1775

name_in_progress: 404, 552, 553, 554, 1436
name_length: 26, 51, 545, 549, 551
name_of_file: 26, 27, 51, 545, 549, 551, 556
name_tree_kids_max: 695, 805
named: 1552
names_head: 804, 805, 1513
names_tail: 804, 1513
names_tree: 804, 806, 1513
natural: 816, 881, 891, 896, 903, 911, 913, 914, 924, 930, 932, 935, 972, 975, 982, 1154, 1198, 1278, 1303, 1372, 1377, 1382, 1746
nc: 484
nd: 484
nd: 566, 567, 586, 591, 592, 595
ne: 566, 567, 586, 591, 592, 595
neg: 705
negate: 16, 65, 103, 105, 106, 107, 112, 115, 123, 456, 457, 466, 474, 487, 686, 689, 951, 1585, 1769, 1780, 1793, 1797, 1799
negative: 106, 112, 114, 115, 439, 456, 466, 467, 474, 487, 1780, 1793, 1797, 1799
nest: 230, 231, 234, 235, 236, 237, 439, 448, 951, 976, 983, 1172, 1422, 1679, 1681
nest_ptr: 231, 233, 234, 235, 236, 448, 951, 976, 983, 1172, 1194, 1200, 1269, 1278, 1323, 1378, 1422, 1679
nest_size: 11, 231, 234, 236, 439, 1422, 1514, 1679
nesting_level: 730, 1632, 1635
new_annot_whatsit: 1556, 1558, 1560, 1567, 1568
new_character: 609, 931, 1092, 1295, 1301, 1302
new_choice: 865, 1350
new_delta_from_break_width: 1020
new_delta_to_break_width: 1019
new_disc: 163, 1212, 1295
new_edge: 1719, 1722, 1738
new_font: 706, 1434, 1435
new_font_type: 703, 705, 720, 726
new_glue: 171, 172, 891, 942, 962, 969, 971, 985, 1219, 1221, 1232, 1238, 1349
new_graf: 1268, 1269
new_hlist: 901, 903, 919, 924, 925, 926, 930, 932, 938, 943
new_hyph_exceptions: 1111, 1430
new_index: 1815, 1816, 1819
new_interaction: 1442, 1443, 1695, 1696
new_kern: 174, 705, 881, 891, 911, 914, 915, 923, 927, 929, 931, 935, 1087, 1218, 1239, 1290, 1291, 1303, 1382, 1721, 1740, 1746
new_letterspaced_font: 706, 1434
new_lig_item: 162, 1088, 1218
new_ligature: 162, 1087, 1212
new_line: 325, 353, 365, 366, 367, 369, 509, 563

new_line_char: 59, 254, 262, 1513, 1515, 1754
\newlinechar primitive: 256
new_line_char_code: 254, 255, 256
new_margin_kern: 823, 1057, 1063
new_math: 165, 1374, 1703, 1707, 1709, 1712, 1723, 1734, 1746
new_noad: 862, 896, 918, 929, 1254, 1271, 1328, 1333, 1336, 1346, 1355, 1369
new_null_box: 154, 882, 885, 889, 896, 923, 926, 955, 969, 985, 1195, 1232, 1269, 1271, 1740
new_param_glue: 170, 172, 855, 954, 992, 1062, 1063, 1219, 1221, 1269, 1381, 1383, 1384, 1740
new_patterns: 1137, 1430
new_penalty: 176, 943, 992, 1067, 1232, 1281, 1381, 1383, 1384
new_randoms: 110, 124, 125
new_rule: 157, 489, 842, 880, 1552
new_save_level: 296, 817, 950, 961, 967, 1202, 1241, 1277, 1295, 1297, 1314
new_skip_param: 172, 855, 1146, 1178, 1746
new_snap_node: 1573, 1574
new_spec: 169, 172, 456, 488, 705, 1002, 1153, 1181, 1220, 1221, 1417, 1418, 1780, 1790, 1791, 1853
new_string: 54, 57, 58, 491, 496, 645, 686, 705, 706, 712, 727, 1435, 1457, 1508, 1615, 1688, 1753
new_style: 864, 1349
new_trie_op: 1120, 1121, 1122, 1142
new_vf_packet: 706, 716
new_whatsit: 1529, 1530, 1534, 1538, 1539, 1540, 1541, 1542, 1546, 1549, 1554, 1556, 1561, 1565, 1569, 1572, 1575, 1576, 1594, 1595, 1596, 1597, 1598, 1624, 1625
new_write_whatsit: 1530, 1531, 1532, 1533
newcolorstack: 497
next: 274, 276, 278, 279
next_break: 1053, 1054
next_char: 571, 823, 917, 929, 1086, 1216
next_char_p: 999
next_p: 647, 650, 654, 657, 658, 659, 661, 663, 729, 732, 736, 738, 740, 741, 743, 745, 1723, 1725
next_pos: 1637
next_random: 124, 126, 127
nh: 566, 567, 586, 591, 592, 595
ni: 566, 567, 586, 591, 592, 595, 705
nil: 16
nk: 566, 567, 586, 591, 592, 600, 705
nl: 59, 566, 567, 571, 586, 591, 592, 595, 600, 603, 1753, 1754
nn: 333, 334
No pages of output: 670, 794
no_align: 226, 287, 288, 961, 1304

\noalign primitive: 287
 no_align_error: 1304, 1307
 no_align_group: 291, 944, 961, 1311, 1661, 1679
 no_boundary: 226, 287, 288, 1207, 1215, 1223, 1268
 \noboundary primitive: 287
 no_break_yet: 1005, 1012, 1013
 no_expand: 228, 287, 288, 388, 391
 \noexpand primitive: 287
 no_expand_flag: 380, 504, 532
 \noindent primitive: 1266
 no_lig_code: 173, 452, 1431, 1432, 1433
 \pdfnoligatures primitive: 1432
 no_limits: 858, 1334, 1335
 \nolimits primitive: 1334
 no_new_control_sequence: 274, 276, 278, 281, 286, 387, 400, 1516, 1648, 1768
 no_print: 54, 57, 58, 75, 98
 no_shrink_error_yet: 1001, 1002, 1003
 no_tag: 570, 595
 no_zip: 680, 681, 685
 noad_size: 857, 862, 874, 929, 937, 1364, 1365
 node_list_display: 198, 202, 206, 208, 213, 215
 node_r_stays_active: 1006, 1027, 1030
 node_size: 142, 144, 145, 146, 148, 182, 187, 1489, 1490
 nom: 586, 587, 589, 603
 non_address: 575, 578, 603, 1086, 1093, 1211
 non_char: 575, 578, 603, 705, 1074, 1075, 1078, 1085, 1086, 1087, 1088, 1092, 1093, 1094, 1209, 1211, 1212, 1215, 1216, 1218, 1295, 1501
 non_discardable: 166, 1005, 1055
 non_existent_path: 705, 706
 non_math: 1224, 1241, 1322
 non_script: 226, 287, 288, 1224, 1349
 \nonscript primitive: 287, 908
 none_seen: 638, 639
 NONEXISTENT: 284
 Nonletter: 1139
 nonnegative_integer: 69, 101, 107, 689
 nonstop_mode: 73, 86, 382, 385, 510, 1440, 1441
 \nonstopmode primitive: 1440
 nop: 610, 612, 613, 615, 617, 717, 719
 norm_min: 1269, 1378, 1624, 1625
 norm_rand: 110, 127, 498
 normal: 153, 154, 167, 168, 171, 173, 174, 182, 195, 204, 207, 209, 327, 353, 358, 393, 394, 465, 474, 497, 499, 506, 508, 511, 515, 516, 527, 533, 647, 653, 657, 662, 729, 735, 738, 744, 824, 825, 833, 834, 835, 836, 840, 841, 842, 843, 848, 849, 850, 852, 853, 854, 858, 862, 872, 892, 908, 925, 953, 977, 986, 987, 1001, 1002, 1005, 1017, 1018, 1042, 1046, 1047, 1073, 1074, 1076, 1153, 1165, 1181, 1186, 1223, 1334, 1341, 1343, 1359, 1379, 1397, 1398, 1399, 1417, 1471, 1637, 1638, 1683, 1723, 1766, 1791, 1794, 1843
 \pdfnormaldeviate primitive: 494
 normal_deviate_code: 494, 495, 497, 498
 normal_paragraph: 950, 961, 963, 1202, 1248, 1261, 1272, 1274, 1277, 1345
 normalize_glue: 1791, 1794
 normalize_selector: 78, 92, 93, 94, 95, 686, 1039
 Not a letter: 1114
 not_found: 15, 45, 46, 474, 481, 586, 596, 634, 638, 639, 686, 1005, 1072, 1107, 1108, 1111, 1118, 1130, 1132, 1147, 1149, 1150, 1316, 1324, 1612, 1733, 1819, 1846
 not_found1: 15, 1111, 1819
 not_found2: 15, 1819
 not_found3: 15, 1819
 not_found4: 15, 1819
 notexpanded:: 277
 np: 566, 567, 586, 591, 592, 602, 603
 nucleus: 857, 858, 859, 862, 863, 866, 872, 874, 896, 901, 910, 911, 912, 913, 914, 917, 918, 925, 926, 928, 929, 930, 931, 1254, 1271, 1328, 1329, 1333, 1336, 1341, 1343, 1346, 1364, 1369
 null: 133, 134, 136, 138, 140, 141, 143, 144, 153, 154, 162, 163, 167, 168, 169, 170, 171, 172, 182, 186, 187, 193, 194, 200, 218, 219, 220, 222, 228, 230, 233, 234, 236, 237, 240, 241, 250, 251, 297, 314, 317, 321, 328, 329, 334, 336, 347, 353, 379, 380, 384, 397, 400, 408, 409, 412, 416, 417, 418, 423, 426, 433, 436, 441, 446, 449, 453, 478, 490, 492, 497, 498, 499, 504, 508, 515, 516, 523, 531, 534, 575, 578, 603, 605, 609, 633, 638, 642, 647, 651, 657, 660, 673, 674, 686, 693, 700, 705, 727, 729, 733, 738, 739, 742, 753, 754, 756, 763, 764, 766, 767, 769, 771, 772, 773, 775, 778, 779, 781, 782, 783, 784, 786, 792, 795, 797, 803, 804, 806, 807, 814, 815, 820, 822, 823, 825, 830, 831, 834, 840, 842, 844, 849, 852, 857, 861, 865, 868, 891, 894, 895, 896, 897, 902, 907, 908, 928, 930, 931, 932, 936, 937, 942, 943, 947, 950, 952, 953, 959, 960, 965, 966, 967, 968, 970, 972, 973, 975, 977, 980, 981, 982, 983, 988, 997, 1003, 1005, 1013, 1016, 1022, 1023, 1024, 1026, 1027, 1028, 1032, 1033, 1034, 1035, 1039, 1040, 1041, 1043, 1045, 1048, 1053, 1054, 1055, 1057, 1058, 1059, 1060, 1061, 1063, 1065, 1066, 1067, 1071, 1073, 1075, 1080, 1083, 1084, 1085, 1087, 1088, 1090, 1091, 1092, 1093, 1094, 1095, 1105, 1109, 1112, 1145, 1146, 1147, 1149, 1150, 1154, 1155, 1156, 1158, 1168, 1169, 1170, 1171, 1175, 1176, 1177, 1186, 1187, 1188, 1189, 1191, 1192, 1193, 1194,

1195, 1197, 1198, 1199, 1200, 1203, 1204, 1205,
 1207, 1209, 1211, 1212, 1213, 1214, 1215, 1217,
 1218, 1220, 1221, 1248, 1252, 1253, 1254, 1257,
 1258, 1261, 1265, 1269, 1274, 1288, 1295, 1299,
 1301, 1302, 1309, 1314, 1317, 1323, 1324, 1327,
 1345, 1352, 1354, 1359, 1362, 1363, 1364, 1372,
 1374, 1377, 1380, 1383, 1384, 1404, 1405, 1425,
 1426, 1461, 1466, 1474, 1489, 1490, 1515, 1519,
 1533, 1534, 1536, 1544, 1545, 1548, 1551, 1552,
 1555, 1556, 1565, 1573, 1577, 1603, 1604, 1605,
 1615, 1616, 1623, 1629, 1630, 1636, 1637, 1668,
 1674, 1683, 1691, 1706, 1707, 1708, 1709, 1712,
 1721, 1723, 1725, 1730, 1731, 1733, 1734, 1735,
 1738, 1745, 1746, 1751, 1756, 1757, 1758, 1768,
 1782, 1783, 1784, 1809, 1815, 1816, 1817, 1818,
 1819, 1820, 1821, 1823, 1824, 1825, 1826, 1827,
 1828, 1829, 1830, 1831, 1832, 1836, 1837, 1838,
 1841, 1849, 1852, 1860, 1863, 1866
null delimiter: 258, 1243
null_character: 581, 582, 898, 899
null_code: 22, 250
null_cs: 240, 284, 285, 376, 400, 673, 693, 705,
 706, 712, 1435, 1768
null_delimiter: 860, 861, 1359
null_delimiter_space: 265, 882
\nulldelimiterspace primitive: 266
nullDelimiterSpace_code: 265, 266
null_flag: 156, 157, 489, 827, 955, 969, 977, 1552
null_font: 192, 250, 497, 578, 579, 586, 604, 645,
 673, 674, 693, 697, 705, 706, 712, 720, 823,
 839, 882, 883, 898, 1040, 1435, 1498, 1499,
 1519, 1587, 1589, 1593
\nullfont primitive: 579
null_list: 14, 180, 406, 956
num: 476, 484, 612, 614, 617
num_error: 1790, 1793, 1797, 1799
\numexpr primitive: 1778
num_style: 878, 920
Number too big: 471
\number primitive: 494
number_code: 494, 495, 496, 497, 498
numerator: 859, 866, 873, 874, 920, 1359, 1363
num1: 876, 920
num2: 876, 920
num3: 876, 920
nw: 566, 567, 586, 591, 592, 595, 705, 706
nx_plus_y: 105, 481, 892, 1418, 1795
o: 286, 634, 823, 844, 967, 976, 1782
obj_annot_ptr: 695, 781, 782, 783, 1558, 1560,
 1630, 1635, 1636
obj_aux: 695, 698, 752, 799, 805
obj_bead_attr: 695, 1600, 1637
obj_bead_data: 695, 786, 1637
obj_bead_next: 695, 1600, 1637
obj_bead_page: 695, 1600, 1637
obj_bead_prev: 695, 1600, 1637
obj_bead_ptr: 695, 1637
obj_bead_rect: 695, 786, 1600
obj_data_ptr: 695, 1544, 1548, 1552, 1623
obj_dest_ptr: 695, 784, 795, 797, 1555, 1565, 1637
obj_entry: 694, 695, 696, 698
obj_info: 498, 693, 695, 698, 767, 794, 795, 797,
 799, 801, 802, 803, 804, 805, 1600, 1637
obj_link: 693, 695, 698, 789, 790, 796, 798, 799,
 800, 801, 802, 803, 804, 805, 812, 813, 814, 1545
obj_obj_data: 695, 772, 1544, 1603
obj_obj_is_file: 695, 772, 1544, 1603
obj_obj_is_stream: 695, 772, 1544, 1603
obj_obj_stream_attr: 695, 772, 1544, 1603
obj_offset: 695, 698, 813, 814, 815
obj_os_idx: 695, 698, 814
obj_outline_action_objnum: 695, 789, 1563
obj_outline_attr: 695, 789, 1563
obj_outline_count: 695, 788, 789, 1563, 1637
obj_outline_first: 695, 789, 1563, 1637
obj_outline_last: 695, 789, 1563, 1637
obj_outline_next: 695, 788, 789, 1563, 1637
obj_outline_parent: 695, 788, 789, 1563, 1637
obj_outline_prev: 695, 789, 1562, 1563
obj_outline_ptr: 695, 1563
obj_outline_title: 695, 789, 1563
obj_ptr: 693, 696, 697, 698, 752, 770, 786, 788,
 790, 804, 806, 813, 814, 1504, 1505, 1513, 1544,
 1548, 1552, 1555, 1558, 1563, 1579, 1600, 1636
obj_tab: 694, 695, 696, 697, 698, 804, 1504, 1505
obj_tab_size: 694, 696, 697, 698, 1504, 1505, 1513
obj_thread_first: 695, 1600, 1637
obj_type_dest: 695, 698, 796, 1555, 1565, 1630,
 1637
obj_type_font: 693, 695, 801
obj_type_obj: 695, 1504, 1505, 1544, 1546
obj_type_others: 695, 698, 752, 786, 788, 790, 804,
 806, 807, 814, 1563, 1579, 1600, 1636
obj_type_outline: 695, 789, 1563
obj_type_page: 498, 695, 698, 752, 795, 797, 799,
 800, 802, 1600, 1630
obj_type_pages: 695, 770, 800, 802, 803
obj_type_struct_dest: 695, 798, 1555, 1565, 1630,
 1637
obj_type_thread: 695, 790, 1630, 1637
obj_type_xform: 497, 695, 1504, 1505, 1548, 1549
obj_type_ximage: 497, 695, 1504, 1505, 1552, 1554
obj_xform_attr: 695, 756, 1548
obj_xform_box: 695, 775, 1548, 1623

obj_xform_depth: 695, 1548, 1549, 1603, 1637
obj_xform_height: 695, 1548, 1549, 1603
obj_xform_resources: 695, 763, 1548
obj_xform_width: 695, 1548, 1549, 1603
obj_ximage_attr: 695, 778, 1552
obj_ximage_data: 497, 695, 767, 778, 1552, 1637
obj_ximage_depth: 695, 1552, 1554, 1603
obj_ximage_height: 695, 1552, 1554, 1603
obj_ximage_width: 695, 1552, 1554, 1603
objname: 698, 793, 805, 1627
objnum: 698, 805, 1627
octal_token: 464, 470
odd: 62, 100, 116, 165, 211, 530, 604, 702, 934, 1042, 1075, 1079, 1085, 1086, 1090, 1091, 1389, 1396, 1426, 1473, 1688, 1800, 1819, 1824
off_save: 1241, 1242, 1272, 1273, 1308, 1309, 1318, 1370, 1371
OK: 1476
OK_so_far: 466, 471
OK_to_interrupt: 88, 96, 97, 98, 349, 1208
old_l: 1005, 1011, 1026
old_mode: 727, 1615, 1617, 1618
old_rover: 149
old_setting: 263, 264, 333, 334, 491, 496, 560, 608, 645, 666, 686, 705, 706, 712, 727, 1435, 1457, 1615, 1617, 1688, 1753
omit: 226, 287, 288, 964, 965, 1304
\omit primitive: 287
omit_error: 1304, 1307
omit_template: 180, 965, 966
one_bp: 672, 687, 688, 693
one_hundred_bp: 687, 688, 690, 692, 693, 792, 1637
one_hundred_inch: 672, 687, 688, 1552
Only one # is allowed...: 960
op_byte: 571, 583, 705, 823, 917, 929, 1086, 1088, 1218
op_noad: 858, 866, 872, 874, 902, 904, 909, 925, 937, 1334, 1335, 1337
op_start: 1097, 1098, 1101, 1122, 1503
open_area: 1521, 1531, 1603, 1622
open_ext: 1521, 1531, 1603, 1622
open_fmt_file: 550, 1517
\openin primitive: 1450
open_log_file: 78, 92, 382, 497, 558, 560, 561, 563, 684, 1435, 1515
open_name: 1521, 1531, 1603, 1622
open_noad: 858, 866, 872, 874, 904, 909, 936, 937, 938, 1334, 1335
open_node: 1521, 1524, 1526, 1528, 1603, 1604, 1605, 1620
open_node_size: 1521, 1531, 1604, 1605
open_or_close_in: 1452, 1453

\openout primitive: 1524
open_parens: 326, 353, 384, 563, 1515, 1755
open_subentries: 788, 1637
\or primitive: 517
or_code: 515, 517, 518, 526, 535, 1471, 1668
ord: 20
ord_noad: 857, 858, 862, 863, 866, 872, 874, 904, 905, 928, 929, 937, 940, 941, 1253, 1333, 1334, 1335, 1364
order: 195
oriental characters: 152, 612
orig_char_info: 604, 705
other_A_token: 471
other_char: 225, 250, 311, 313, 316, 320, 369, 471, 490, 552, 1112, 1138, 1207, 1215, 1268, 1302, 1329, 1332, 1338
other_token: 311, 431, 464, 467, 471, 490, 529, 1243, 1399, 1761, 1769, 1784, 1785
othercases: 10
others: 10, 1620
Ugh...clobbered: 1512
out_form: 1637, 1644, 1647
out_image: 1637, 1643, 1646
out_param: 225, 311, 313, 316, 379
out_param_token: 311, 505
out_thread: 790, 1600
out_what: 1613, 1614, 1620, 1623, 1639, 1645
\outer primitive: 1386
outer_call: 228, 297, 361, 373, 375, 376, 379, 388, 413, 417, 422, 956, 1330, 1473, 1616
outer_doing_leaders: 647, 656, 657, 665, 729, 737, 738, 746
outline_list_count: 1562, 1563
outlines: 788, 806, 1513
output: 4
Output loop...: 1201
Output routine didn't use...: 1205
Output written on x: 670, 794
\output primitive: 248
output_active: 447, 839, 851, 1163, 1166, 1167, 1171, 1182, 1202, 1203
output_file_name: 558, 559, 670, 684, 794, 814, 815
output_group: 291, 1202, 1278, 1661, 1679
output_one_char: 726, 731
output_penalty: 254
\outputpenalty primitive: 256
output_penalty_code: 254, 255, 256, 1190
output_routine: 248, 1189, 1202
output_routine_loc: 248, 249, 250, 329, 345, 1404
output_text: 329, 336, 345, 1202, 1203
\over primitive: 1356
over_code: 1356, 1357, 1360

over_noad: 863, 866, 872, 874, 909, 937, 1334
\overwithdelims primitive: 1356
overbar: 881, 910, 913
overflow: 35, 42, 43, 94, 138, 143, 234, 279, 282, 286, 295, 296, 343, 350, 400, 416, 543, 607, 678, 680, 686, 698, 705, 706, 719, 725, 1117, 1121, 1131, 1141, 1513, 1635, 1768, 1782
overflow in arithmetic: 9, 104
Overfull \hbox{...}: 842
Overfull \vbox{...}: 853
overfull boxes: 1030
overfull_rule: 265, 842, 976, 980
\overfullrule primitive: 266
overfull_rule_code: 265, 266
\overline primitive: 1334
p: 112, 114, 138, 141, 143, 148, 149, 154, 157, 162, 163, 165, 169, 170, 171, 172, 174, 176, 185, 190, 192, 194, 196, 200, 216, 218, 219, 220, 222, 236, 278, 281, 284, 285, 298, 299, 300, 301, 303, 306, 314, 317, 321, 328, 337, 345, 347, 358, 388, 415, 433, 439, 476, 490, 491, 499, 508, 523, 524, 609, 634, 642, 647, 657, 666, 674, 693, 729, 738, 823, 844, 855, 862, 864, 865, 867, 868, 880, 881, 885, 887, 891, 892, 893, 896, 902, 911, 914, 919, 925, 928, 932, 948, 950, 963, 967, 975, 976, 1002, 1083, 1111, 1125, 1126, 1130, 1134, 1136, 1137, 1143, 1145, 1147, 1170, 1171, 1189, 1242, 1246, 1253, 1257, 1264, 1271, 1279, 1283, 1288, 1291, 1297, 1301, 1316, 1329, 1333, 1338, 1352, 1354, 1362, 1369, 1372, 1389, 1414, 1422, 1466, 1471, 1480, 1481, 1528, 1529, 1545, 1556, 1573, 1577, 1602, 1615, 1617, 1620, 1635, 1636, 1637, 1679, 1683, 1719, 1723, 1733, 1738, 1744, 1753, 1756, 1757, 1777, 1782, 1821, 1823, 1837, 1838, 1839, 1840, 1841

p-1: 1854
pack_begin_line: 837, 838, 839, 851, 980, 991
pack_buffered_name: 549, 550
pack_cur_name: 555, 556, 563, 772, 1453, 1622
pack_file_name: 545, 555, 563, 589, 713
pack_job_name: 555, 558, 560, 684, 1508
pack_lig: 1212
package: 1263, 1264
packed_ASCII_code: 38, 39, 793, 1124
packet_byte: 725, 726
packet_length: 712, 717, 719
packet_read_signed: 725
packet_read_unsigned: 725, 726
packet_scaled: 725, 726
page: 326, 1552
page_contents: 447, 1157, 1163, 1164, 1168, 1177, 1178, 1185

page_depth: 1159, 1164, 1168, 1179, 1180, 1181, 1185, 1187, 1611
\pagedepth primitive: 1160
page_disc: 1176, 1200, 1203, 1859, 1860
\pagediscards primitive: 1861
\pagefilstretch primitive: 1160
\pagefillstretch primitive: 1160
\pagefilllstretch primitive: 1160
page_goal: 1157, 1159, 1163, 1164, 1182, 1183, 1184, 1185, 1186, 1187
\pagegoal primitive: 1160
page_head: 180, 233, 1157, 1163, 1165, 1168, 1191, 1194, 1200, 1203, 1232
page_ins_head: 180, 498, 1158, 1163, 1182, 1185, 1195, 1196, 1197
page_ins_node_size: 1158, 1186, 1196
page_loc: 666, 668
page_max_depth: 1157, 1159, 1164, 1168, 1180, 1194
page_shrink: 1159, 1162, 1181, 1184, 1185, 1186
\pageshrink primitive: 1160
page_so_far: 447, 1159, 1162, 1164, 1181, 1184, 1186, 1423
page_stack: 326
\pagestretch primitive: 1160
page_tail: 233, 1157, 1163, 1168, 1175, 1177, 1194, 1200, 1203, 1232
page_total: 1159, 1162, 1179, 1180, 1181, 1184, 1185, 1187, 1611
\pagetotal primitive: 1160
pagebox: 1552
pages_tail: 696, 800, 802
pages_tree_kids_max: 695, 770, 794, 802, 803
panicking: 183, 184, 1208, 1519
\par primitive: 356
par_end: 225, 356, 357, 1224, 1272
par_fill_skip: 242, 992, 1842, 1843, 1846, 1853
\parfillskip primitive: 244
par_fill_skip_code: 242, 243, 244, 992
par_indent: 265, 1269, 1271
\parindent primitive: 266
par_indent_code: 265, 266
par_loc: 355, 356, 373, 1491, 1492
\parshape primitive: 287
\parshapedimen primitive: 1672
par_shape_dimen_code: 1672, 1673, 1674
\parshapeindent primitive: 1672
par_shape_indent_code: 1672, 1673, 1674
\parshapelen primitive: 1672
par_shape_length_code: 1672, 1673, 1674
par_shape_loc: 248, 250, 251, 287, 288, 449, 1248, 1426

par_shape_ptr: 248, 250, 251, 449, 990, 1023, 1024, 1026, 1066, 1248, 1327, 1427, 1674
par_skip: 242, 1269
\parskip primitive: 244
par_skip_code: 242, 243, 244, 1269
par_token: 355, 356, 361, 418, 421, 425, 1273, 1492
Paragraph ended before...: 422
param: 568, 573, 584
param_base: 576, 578, 584, 592, 601, 602, 603, 605, 607, 705, 706, 876, 877, 1220, 1500, 1501
param_end: 584
param_ptr: 330, 345, 346, 353, 416
param_size: 11, 330, 416, 1514
param_stack: 329, 330, 346, 381, 414, 415, 416
param_start: 329, 345, 346, 381
parameter: 329, 336, 381
parameters for symbols: 876, 877
Parameters...consecutively: 502
parent_box: 1630, 1635, 1636, 1637
Pascal-H: 3, 4, 9, 10, 27, 28, 33, 34
Pascal: 1, 10, 869, 940
pass_number: 997, 1021, 1040
pass_text: 388, 520, 526, 535, 536
passive: 997, 1021, 1022, 1040, 1041
passive_node_size: 997, 1021, 1041
Patterns can be...: 1430
\patterns primitive: 1428
pause_for_instructions: 96, 98
pausing: 254, 385
\pausing primitive: 256
pausing_code: 254, 255, 256
pc: 484
pdf_action_file: 695, 1536, 1556, 1603, 1630
pdf_action_goto: 695, 1556, 1603, 1630
pdf_action_id: 695, 1536, 1556, 1603, 1630
pdf_action_named_id: 695, 1536, 1556, 1603, 1630
pdf_action_new_window: 695, 1556, 1630
pdf_action_page: 695, 1536, 1556, 1603, 1630
pdf_action_page_tokens: 695, 1536, 1556, 1603, 1630
pdf_action_refcount: 695, 1536, 1556
pdf_action_size: 695, 1536, 1556
pdf_action_struct_id: 695, 1536, 1556, 1630
pdf_action_thread: 695, 1556, 1603, 1630
pdf_action_type: 695, 782, 1536, 1556, 1603, 1630
pdf_action_user: 695, 782, 1536, 1556, 1603, 1630
pdf_action_user_tokens: 695, 1536, 1556, 1603, 1630
pdf_adjust_interword_glue: 254, 1207, 1219
\pdfadjustinterwordglue primitive: 256
pdf_adjust_interword_glue_code: 254, 255, 256
pdf_adjust_spacing: 254, 999, 1003, 1017, 1018, 1027, 1042, 1043, 1046, 1047, 1066
\pdfadjustspacing primitive: 256
pdf_adjust_spacing_code: 254, 255, 256
\pdfannot primitive: 1524
pdf_annot_data: 695, 781, 1558, 1603, 1604, 1605
pdf_annot_list: 754, 765, 771, 781, 1628, 1630
pdf_annot_node: 781, 1524, 1526, 1528, 1558, 1603, 1604, 1605, 1639, 1645
pdf_annot_node_size: 695, 1558, 1560, 1604, 1605
pdf_annot_objnum: 695, 1558, 1630
pdf_append_kern: 254, 705
\pdfappendkern primitive: 256
pdf_append_kern_code: 254, 255, 256
pdf_append_list: 693, 696, 698, 1630, 1635, 1636, 1637, 1639, 1645
pdf_append_list_arg: 696, 698
pdf_append_list_end: 698
pdf_bead_list: 754, 765, 771, 786, 1628, 1637
pdf_begin_dict: 698, 699, 752, 762, 769, 772, 778, 781, 782, 784, 789, 803, 805, 1600
pdf_begin_obj: 698, 772, 784, 795, 797
pdf_begin_stream: 685, 699, 757, 772, 814
pdf_begin_string: 693, 726
pdf_begin_text: 693
pdf_bottom: 695, 781, 782, 785, 1630, 1635, 1637
pdf_box_spec_art: 696, 697, 1552
pdf_box_spec_bleed: 696, 697, 1552
pdf_box_spec_crop: 696, 697, 1552
pdf_box_spec_media: 696, 697, 1552
pdf_box_spec_trim: 696, 697, 1552
pdf_buf: 680, 681, 685, 686, 698, 699, 772
pdf_buf_size: 680, 681, 686, 698, 699
\pdfcatalog primitive: 1524
pdf_catalog_code: 1524, 1526, 1528
pdf_catalog_openaction: 806, 1579, 1628, 1629
pdf_catalog_toks: 806, 1579, 1628, 1629
pdf_char_marked: 801
pdf_char_used: 708
pdf_check_obj: 497, 1545, 1546, 1549, 1554
pdf_check_vf_cur_val: 497, 703, 720, 1587
\pdfcolorstack primitive: 1524
pdf_colorstack_cmd: 695, 727, 1539, 1603, 1604, 1605
pdf_colorstack_data: 695, 727, 1539, 1603, 1604, 1605
pdf_colorstack_getter_node_size: 695, 1539, 1604, 1605
\pdfcolorstackinit primitive: 494
pdf_colorstack_init_code: 494, 495, 497, 498
pdf_colorstack_node: 1524, 1526, 1528, 1539, 1603, 1604, 1605, 1639, 1645

pdf_colorstack_node_size: [695](#)
pdf_colorstack_setter_node_size: [695](#), [1539](#), [1604](#), [1605](#)
pdf_colorstack_stack: [695](#), [727](#), [1539](#), [1603](#)
pdf_compress_level: [254](#), [672](#), [673](#), [685](#), [698](#), [748](#)
\pdfcompresslevel primitive: [256](#)
pdf_compress_level_code: [254](#), [255](#), [256](#)
pdf_copy_font: [227](#), [287](#), [288](#), [439](#), [604](#), [1388](#), [1434](#)
\pdfcopyfont primitive: [287](#)
pdf_create_obj: [693](#), [698](#), [770](#), [804](#), [1544](#), [1548](#), [1552](#), [1555](#), [1563](#), [1636](#)
\pdfcreationdate primitive: [494](#)
pdf_creation_date_code: [494](#), [495](#), [497](#)
pdf_cur_form: [752](#), [756](#), [763](#), [775](#), [1623](#), [1628](#)
pdf_cur_Tm_a: [690](#), [691](#), [692](#), [693](#)
pdf_decimal_digits: [254](#), [672](#), [792](#)
\pdfdecimaldigits primitive: [256](#)
pdf_decimal_digits_code: [254](#), [255](#), [256](#)
pdf_delta_h: [690](#), [691](#), [692](#), [693](#)
pdf_depth: [695](#), [1549](#), [1554](#), [1556](#), [1565](#), [1601](#), [1603](#), [1604](#), [1606](#), [1607](#), [1611](#), [1612](#), [1630](#), [1635](#), [1636](#), [1637](#), [1643](#), [1644](#), [1646](#)
\pdfdest primitive: [1524](#)
pdf_dest_fit: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_fitb: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_fitbh: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_fitbv: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_fith: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_fitr: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_fitv: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_id: [695](#), [1565](#), [1603](#), [1604](#), [1605](#), [1637](#)
pdf_dest_list: [754](#), [765](#), [784](#), [1628](#), [1637](#)
pdf_dest_margin: [265](#), [1637](#)
\pdfdestmargin primitive: [266](#)
pdf_dest_margin_code: [265](#), [266](#)
pdf_dest_named_id: [695](#), [784](#), [1565](#), [1603](#), [1604](#), [1605](#), [1637](#)
pdf_dest_names_ptr: [698](#), [804](#), [805](#), [1513](#), [1628](#), [1629](#)
pdf_dest_node: [695](#), [1524](#), [1526](#), [1528](#), [1565](#), [1603](#), [1604](#), [1605](#), [1639](#), [1645](#)
pdf_dest_node_size: [695](#), [1565](#), [1604](#), [1605](#)
pdf_dest_objnum: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_type: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_xyz: [695](#), [784](#), [1565](#), [1603](#), [1637](#)
pdf_dest_xyz_zoom: [695](#), [784](#), [1565](#), [1603](#)
pdf_doing_string: [691](#), [692](#), [693](#)
pdf_doing_text: [691](#), [693](#)
pdf_draftmode: [254](#), [672](#), [683](#), [748](#), [1517](#)
\pdfdraftmode primitive: [256](#)
pdf_draftmode_code: [254](#), [255](#), [256](#)
pdf_draftmode_option: [691](#), [1517](#)

pdf_draftmode_value: [691](#), [1517](#)
pdf_dummy_font: [691](#), [693](#), [697](#)
pdf_each_line_depth: [265](#), [1064](#), [1065](#)
\pdfeachlinedepth primitive: [266](#)
pdf_each_line_depth_code: [265](#), [266](#)
pdf_each_line_height: [265](#), [1064](#), [1065](#)
\pdfeachlineheight primitive: [266](#)
pdf_each_line_height_code: [265](#), [266](#)
pdf_end_dict: [698](#), [762](#), [769](#), [781](#), [782](#), [784](#), [788](#), [789](#), [803](#), [804](#), [805](#), [806](#), [807](#), [1600](#)
\pdfendlink primitive: [1524](#)
pdf_end_link_node: [1524](#), [1526](#), [1528](#), [1561](#), [1603](#), [1604](#), [1605](#), [1639](#), [1645](#)
pdf_end_obj: [685](#), [698](#), [772](#), [784](#), [786](#), [790](#), [795](#), [797](#), [1563](#), [1579](#)
pdf_end_stream: [685](#), [699](#), [760](#), [772](#), [814](#)
pdf_end_string: [692](#), [693](#)
pdf_end_string_nl: [692](#), [693](#)
pdf_end_text: [693](#), [760](#), [1637](#)
\pdfendthread primitive: [1524](#)
pdf_end_thread_node: [1524](#), [1526](#), [1528](#), [1569](#), [1603](#), [1604](#), [1605](#), [1639](#), [1645](#)
pdf_error: [497](#), [597](#), [599](#), [683](#), [685](#), [686](#), [689](#), [693](#), [705](#), [706](#), [712](#), [720](#), [726](#), [727](#), [747](#), [748](#), [772](#), [784](#), [823](#), [1005](#), [1537](#), [1545](#), [1548](#), [1552](#), [1556](#), [1558](#), [1560](#), [1561](#), [1565](#), [1566](#), [1573](#), [1579](#), [1587](#), [1589](#), [1593](#), [1603](#), [1620](#), [1623](#), [1630](#), [1635](#), [1637](#), [1639](#), [1645](#)
\pdfescapehex primitive: [494](#)
pdf_escape_hex_code: [494](#), [495](#), [497](#)
\pdfescapename primitive: [494](#)
pdf_escape_name_code: [494](#), [495](#), [497](#)
\pdfescapestring primitive: [494](#)
pdf_escape_string_code: [494](#), [495](#), [497](#)
pdf_f: [691](#), [692](#), [693](#)
\pdfffakespace primitive: [1524](#)
pdf_fake_space_node: [1524](#), [1526](#), [1528](#), [1596](#), [1603](#), [1604](#), [1605](#), [1639](#), [1645](#)
pdf_file: [680](#), [684](#), [794](#)
\pdffiledump primitive: [494](#)
pdf_file_dump_code: [494](#), [495](#), [497](#)
\pdffilemoddate primitive: [494](#)
pdf_file_mod_date_code: [494](#), [495](#), [497](#)
\pdffilesize primitive: [494](#)
pdf_file_size_code: [494](#), [495](#), [497](#)
pdf_first_line_height: [265](#), [1064](#), [1065](#)
\pdffirstlineheight primitive: [266](#)
pdf_first_line_height_code: [265](#), [266](#)
pdf_first_outline: [788](#), [789](#), [1563](#), [1628](#), [1629](#)
pdf_fix_dest: [795](#), [796](#)
pdf_fix_struct_dest: [797](#), [798](#)
pdf_fix_thread: [1600](#)

pdf_flush: 680, 685, 794
pdf_font_attr: 704, 801, 1589
\pdffontattr primitive: 1524
pdf_font_attr_code: 1524, 1526, 1528
pdf_font_auto_expand: 692, 693, 705, 712, 720, 821
pdf_font_blink: 192, 693, 705, 720, 821
pdf_font_ef_base: 705, 821
pdf_font_elink: 703, 705, 821
\pdffontexpand primitive: 1524
pdf_font_expand_code: 1524, 1526, 1528
pdf_font_expand_ratio: 192, 692, 705, 706, 712, 720, 821, 823
pdf_font_has_space_char: 693, 821
pdf_font_kn_ac_base: 705, 821
pdf_font_kn_bc_base: 705, 821
pdf_font_kn_bs_base: 705, 821
pdf_font_list: 693, 708, 750, 753, 764, 766, 775, 776, 777
pdf_font_lp_base: 705, 821
pdf_font_map: 693, 708
\pdffontname primitive: 494
pdf_font_name_code: 494, 495, 497, 498
pdf_font_nobuiltin_tounicode: 704, 1593
pdf_font_num: 498, 692, 693, 698, 708, 766, 801
\pdffontobjnum primitive: 494
pdf_font_objnum_code: 494, 495, 497, 498
pdf_font_rp_base: 705, 821
pdf_font_sh_bs_base: 705, 821
pdf_font_shrink: 705, 712, 821, 823
pdf_font_size: 690, 692, 693, 708
\pdffontsize primitive: 494
pdf_font_size_code: 494, 495, 497, 498
pdf_font_st_bs_base: 705, 821
pdf_font_step: 705, 706, 712, 821, 823
pdf_font_stretch: 705, 712, 821, 823
pdf_font_type: 703, 704, 705, 706, 712, 720, 726
pdf_force_pagebox: 254, 1552
\pdfforcepagebox primitive: 256
pdf_force_pagebox_code: 254, 255, 256
pdf_gamma: 254, 672, 683
\pdffgamma primitive: 256
pdf_gamma_code: 254, 255, 256
pdf_gen_tounicode: 254, 801
\pdffgentounicode primitive: 256
pdf_gen_tounicode_code: 254, 255, 256
pdf_get_mem: 678, 705, 1544, 1548, 1552, 1563, 1637
\pdffglyptounicode primitive: 1524
pdf_glyph_to_unicode_code: 1524, 1526, 1528
pdf_gone: 680, 681, 685, 794
pdf_h: 690, 691, 692
pdf_h_origin: 265, 672, 755

\pdforigin primitive: 266
pdf_h_origin_code: 265, 266
pdf_height: 695, 1549, 1554, 1556, 1565, 1601, 1603, 1604, 1606, 1607, 1611, 1612, 1630, 1635, 1636, 1637, 1643, 1644
pdf_hlist_out: 727, 729, 733, 737, 738, 742, 746, 751, 1555
pdf_ignored_dimen: 237, 265, 855, 963, 1064, 1065, 1202, 1234, 1261, 1277, 1345, 1481
\pdfignoreddimen primitive: 266
pdf_ignored_dimen_code: 265, 266
pdf_image_apply_gamma: 254, 672, 683
\pdfimageapplygamma primitive: 256
pdf_image_apply_gamma_code: 254, 255, 256
pdf_image_gamma: 254, 672, 683
\pdfimagegamma primitive: 256
pdf_image_gamma_code: 254, 255, 256
pdf_image_hicolor: 254, 672, 683
\pdfimagehicolor primitive: 256
pdf_image_hicolor_code: 254, 255, 256
pdf_image_procset: 701, 750, 753, 768, 776, 777
pdf_image_resolution: 254, 672, 1552
\pdfimageresolution primitive: 256
pdf_image_resolution_code: 254, 255, 256
pdf_include_chars: 1587, 1588
\pdfincludechars primitive: 1524
pdf_include_chars_code: 1524, 1526, 1528
pdf_inclusion_copy_font: 254, 683
\pdfinclusioncopyfonts primitive: 256
pdf_inclusion_copy_font_code: 254, 255, 256
pdf_inclusion_errorlevel: 254, 1505, 1552
\pdfinclusionerrorlevel primitive: 256
pdf_inclusion_errorlevel_code: 254, 255, 256
pdf_indirect: 702, 1630
pdf_indirect_ln: 702, 756, 769, 770, 788, 789, 803, 804, 806, 814, 815, 1600
\pdfinfo primitive: 1524
pdf_info_code: 1524, 1526, 1528
pdf_info OMIT_DATE: 254, 807
\pdfinfo OMIT_DATE primitive: 256
pdf_info OMIT_DATE_CODE: 254, 255, 256
pdf_info_toks: 807, 1578, 1628, 1629
pdf_init_font: 693, 1587
pdf_init_font_CUR_VAL: 497, 693, 703
pdf_init_map_file: 1517
pdf_insert_fake_space: 693, 1639, 1645
\pdfinsertht primitive: 494
pdf_insert_ht_code: 494, 495, 497, 498
pdf_insert_interword_space: 693
pdf_int_entry: 702, 1630
pdf_int_entry_ln: 702, 788, 789, 803, 814, 815
pdf_int_pars: 254

\pdfinterwordspaceoff primitive: [1524](#)
`pdf_interword_space_off_node`: [1524](#), [1526](#), [1528](#),
[1595](#), [1603](#), [1604](#), [1605](#), [1639](#), [1645](#)
\pdfinterwordspaceon primitive: [1524](#)
`pdf_interword_space_on_node`: [1524](#), [1526](#), [1528](#),
[1594](#), [1603](#), [1604](#), [1605](#), [1639](#), [1645](#)
`pdf_last_annot`: [450](#), [1557](#), [1558](#)
\pdflastannot primitive: [442](#)
`pdf_last_annot_code`: [442](#), [443](#), [450](#)
`pdf_last_byte`: [685](#), [696](#)
`pdf_last_f`: [691](#), [693](#)
`pdf_last_fs`: [691](#), [693](#)
`pdf_last_line_depth`: [265](#), [1064](#), [1065](#)
\pdflastlinedepth primitive: [266](#)
`pdf_last_line_depth_code`: [265](#), [266](#)
`pdf_last_link`: [450](#), [1559](#), [1560](#)
\pdflastlink primitive: [442](#)
`pdf_last_link_code`: [442](#), [443](#), [450](#)
\pdflastmatch primitive: [494](#)
`pdf_last_match_code`: [494](#), [495](#), [497](#)
`pdf_last_obj`: [450](#), [1504](#), [1505](#), [1543](#), [1544](#), [1623](#)
\pdflastobj primitive: [442](#)
`pdf_last_obj_code`: [442](#), [443](#), [450](#)
`pdf_last_outline`: [788](#), [789](#), [1563](#), [1628](#), [1629](#)
`pdf_last_page`: [696](#), [752](#), [769](#), [784](#), [1637](#)
`pdf_last_pages`: [696](#), [770](#), [794](#), [802](#), [803](#), [806](#), [1513](#)
`pdf_last_resources`: [750](#), [752](#), [756](#), [762](#), [769](#)
`pdf_last_stream`: [696](#), [752](#), [769](#)
`pdf_last_thread_id`: [1628](#), [1637](#)
`pdf_last_thread_named_id`: [1628](#), [1637](#)
`pdf_last_x_pos`: [450](#), [1570](#), [1621](#), [1641](#)
\pdflastxpos primitive: [442](#)
`pdf_last_x_pos_code`: [442](#), [443](#), [450](#)
`pdf_last_xform`: [450](#), [1504](#), [1505](#), [1547](#), [1548](#), [1623](#)
\pdflastxf orm primitive: [442](#)
`pdf_last_xform_code`: [442](#), [443](#), [450](#)
`pdf_last_ximage`: [450](#), [1504](#), [1505](#), [1550](#), [1552](#), [1623](#)
\pdflastximage primitive: [442](#)
`pdf_last_ximage_code`: [442](#), [443](#), [450](#)
`pdf_last_ximage_colordepth`: [450](#), [1550](#), [1552](#)
\pdflastximagecolordepth primitive: [442](#)
`pdf_last_ximage_colordepth_code`: [442](#), [443](#), [450](#)
`pdf_last_ximage_pages`: [450](#), [1550](#), [1552](#)
\pdflastximagepages primitive: [442](#)
`pdf_last_ximage_pages_code`: [442](#), [443](#), [450](#)
`pdf_last_y_pos`: [450](#), [1570](#), [1621](#), [1641](#)
\pdflastypos primitive: [442](#)
`pdf_last_y_pos_code`: [442](#), [443](#), [450](#)
`pdf_lateliteral_node`: [727](#), [1524](#), [1538](#), [1603](#), [1604](#),
[1605](#), [1639](#), [1645](#)
`pdf_left`: [695](#), [781](#), [782](#), [784](#), [785](#), [1630](#), [1635](#), [1637](#)
`pdf_link_action`: [695](#), [782](#), [1560](#), [1603](#), [1604](#), [1605](#)

`pdf_link_attr`: [695](#), [782](#), [1556](#), [1603](#), [1604](#), [1605](#)
`pdf_link_list`: [754](#), [765](#), [771](#), [782](#), [783](#), [1628](#),
[1635](#), [1636](#)
`pdf_link_margin`: [265](#), [1635](#), [1636](#)
\pdflinkmargin primitive: [266](#)
`pdf_link_margin_code`: [265](#), [266](#)
`pdf_link_objnum`: [695](#), [1560](#), [1604](#), [1635](#)
`pdf_link_stack`: [729](#), [730](#), [1631](#), [1633](#), [1636](#)
`pdf_link_stack_ptr`: [730](#), [1631](#), [1633](#), [1634](#), [1635](#)
`pdf_link_stack_record`: [1632](#), [1633](#)
`pdf_link_stack_top`: [1631](#), [1635](#)
\pdfliteral primitive: [1524](#)
`pdf_literal_data`: [695](#), [727](#), [1538](#), [1603](#), [1604](#), [1605](#)
`pdf_literal_mode`: [695](#), [727](#), [1538](#), [1603](#)
`pdf_literal_node`: [1524](#), [1526](#), [1528](#), [1538](#), [1603](#),
[1604](#), [1605](#), [1639](#), [1645](#)
`pdf_lookup_list`: [700](#), [1637](#)
`pdf_major_version`: [254](#), [672](#), [673](#), [683](#), [768](#),
[1505](#), [1552](#)
\pdfmajorversion primitive: [256](#)
`pdf_major_version_code`: [254](#), [255](#), [256](#)
\pdfmapfile primitive: [1524](#)
`pdf_map_file_code`: [1524](#), [1526](#), [1528](#)
\pdfmapline primitive: [1524](#)
`pdf_map_line_code`: [1524](#), [1526](#), [1528](#)
`pdf_mark_char`: [686](#), [693](#), [801](#), [1587](#)
\pdfmatch primitive: [494](#)
`pdf_match_code`: [494](#), [495](#), [497](#)
`pdf_max_link_level`: [1631](#), [1633](#), [1635](#)
\pdfmdfivesum primitive: [494](#)
`pdf_mdfive_sum_code`: [494](#), [495](#), [497](#)
`pdf_mem`: [675](#), [676](#), [677](#), [678](#), [695](#), [705](#), [1504](#), [1505](#)
`pdf_mem_ptr`: [676](#), [677](#), [678](#), [1504](#), [1505](#), [1513](#)
`pdf_mem_size`: [676](#), [677](#), [678](#), [1504](#), [1505](#), [1513](#)
`pdf_minor_version`: [254](#), [672](#), [683](#), [1505](#), [1552](#)
\pdfminorversion primitive: [256](#)
\pdfoptionpdfminorversion primitive: [256](#)
`pdf_minor_version_code`: [254](#), [255](#), [256](#)
`pdf_move_chars`: [254](#), [692](#)
\pdfmovechars primitive: [256](#)
`pdf_move_chars_code`: [254](#), [255](#), [256](#)
\pdfnames primitive: [1524](#)
`pdf_names_code`: [1524](#), [1526](#), [1528](#)
`pdf_names_toks`: [804](#), [1580](#), [1628](#), [1629](#)
`pdf_new_dict`: [698](#), [752](#), [788](#), [804](#), [806](#), [807](#),
[814](#), [1600](#)
`pdf_new_line_char`: [685](#), [686](#), [699](#)
`pdf_new_obj`: [698](#), [786](#), [790](#), [1563](#), [1579](#)
`pdf_new_objnum`: [698](#), [752](#), [802](#), [1558](#), [1560](#),
[1630](#), [1635](#), [1637](#)
`pdf_new_Tm_a`: [692](#)
\pdfnobuiltintounicode primitive: [1524](#)

pdf_nobuiltin_tounicode_code: [1524](#), [1526](#), [1528](#)
\pdfobj primitive: [1524](#)
pdf_obj_code: [1524](#), [1526](#), [1528](#), [1623](#)
pdf_obj_count: [1504](#), [1505](#), [1544](#), [1628](#), [1629](#)
pdf_obj_list: [750](#), [753](#), [764](#), [773](#), [775](#), [776](#), [777](#),
[1543](#), [1628](#), [1639](#), [1645](#)
pdf_obj_objnum: [695](#), [1546](#), [1603](#), [1639](#), [1645](#)
pdf_objcompresslevel: [254](#), [672](#), [683](#)
\pdfobjcompresslevel primitive: [256](#)
pdf_objcompresslevel_code: [254](#), [255](#), [256](#)
pdf_objtype_max: [695](#)
pdf_offset: [680](#), [685](#), [698](#), [794](#), [813](#)
pdf_omit_charset: [254](#), [673](#)
\pdfomitcharset primitive: [256](#)
pdf_omit_charset_code: [254](#), [255](#), [256](#)
pdf_omit_info_dict: [254](#), [794](#), [814](#), [815](#)
\pdfomitinfodict primitive: [256](#)
pdf_omit_info_dict_code: [254](#), [255](#), [256](#)
pdf_omit_procset: [254](#), [768](#)
\pdfomitprocset primitive: [256](#)
pdf_omit_procset_code: [254](#), [255](#), [256](#)
pdf_op_buf: [680](#), [681](#), [698](#)
pdf_op_buf_size: [679](#), [680](#), [681](#), [698](#)
pdf_op_ptr: [680](#), [681](#), [698](#)
pdf_option_always_use_pdfpagebox: [254](#), [1552](#)
\pdfoptionalalwaysusepdfpagebox primitive: [256](#)
pdf_option_always_use_pdfpagebox_code: [254](#), [255](#),
[256](#)
pdf_option_pdf_inclusion_errorlevel: [254](#), [1552](#)
\pdfoptionpdfinclusionerrorlevel primitive: [256](#)
pdf_option_pdf_inclusion_errorlevel_code: [254](#),
[255](#), [256](#)
pdf_origin_h: [691](#), [692](#), [752](#), [780](#)
pdf_origin_v: [691](#), [692](#), [752](#), [780](#)
pdf_os: [698](#)
pdf_os_buf: [680](#), [686](#), [698](#), [699](#)
pdf_os_buf_size: [679](#), [680](#), [681](#), [686](#), [698](#)
pdf_os_cntr: [680](#), [681](#), [698](#), [1513](#)
pdf_os_cur_objnum: [680](#), [681](#), [698](#), [699](#)
pdf_os_enable: [680](#), [683](#), [698](#), [794](#), [815](#)
pdf_os_get_os_buf: [680](#), [686](#)
pdf_os_level: [698](#)
pdf_os_max_objs: [679](#), [698](#), [1513](#)
pdf_os_mode: [680](#), [681](#), [685](#), [698](#), [699](#)
pdf_os_objidx: [680](#), [698](#), [699](#), [1513](#)
pdf_os_objnum: [680](#), [698](#), [699](#)
pdf_os_objoff: [680](#), [698](#), [699](#)
pdf_os_prepare_obj: [698](#)
pdf_os_ptr: [680](#), [681](#), [698](#)
pdf_os_switch: [698](#), [794](#)
pdf_os_write_objstream: [698](#), [699](#), [794](#)

pdf_out: [680](#), [683](#), [685](#), [686](#), [690](#), [692](#), [693](#), [699](#),
[702](#), [756](#), [766](#), [767](#), [769](#), [772](#), [784](#), [785](#), [786](#), [790](#),
[795](#), [797](#), [805](#), [808](#), [814](#), [1600](#), [1630](#), [1637](#)
pdf_out_bytes: [702](#), [814](#)
pdf_out_colorstack: [727](#), [1639](#), [1645](#)
pdf_out_colorstack_startpage: [727](#), [757](#)
pdf_out_literal: [727](#), [1639](#), [1645](#)
pdf_out_restore: [727](#), [1639](#), [1645](#)
pdf_out_save: [727](#), [1639](#), [1645](#)
pdf_out_setmatrix: [727](#), [1639](#), [1645](#)
\pdfoutline primitive: [1524](#)
pdf_outline_code: [1524](#), [1526](#), [1528](#)
pdf_output: [254](#), [747](#), [791](#), [1027](#), [1517](#), [1537](#), [1578](#),
[1579](#), [1581](#), [1582](#)
\pdfoutput primitive: [256](#)
pdf_output_code: [254](#), [255](#), [256](#)
pdf_output_option: [691](#), [1517](#)
pdf_output_value: [691](#), [1517](#)
pdf_page_attr: [248](#), [769](#)
\pdfpageattr primitive: [248](#)
pdf_page_attr_loc: [248](#), [249](#)
pdf_page_group_val: [680](#), [752](#), [769](#), [1637](#)
pdf_page_height: [265](#), [644](#), [755](#), [1600](#)
\pdfpageheight primitive: [266](#)
pdf_page_height_code: [265](#), [266](#)
\pdfpageref primitive: [494](#)
pdf_page_ref_code: [494](#), [495](#), [497](#), [498](#)
pdf_page_resources: [248](#), [763](#)
\pdfpageresources primitive: [248](#)
pdf_page_resources_loc: [248](#), [249](#)
pdf_page_width: [265](#), [644](#), [755](#), [1600](#)
\pdfpagewidth primitive: [266](#)
pdf_page_width_code: [265](#), [266](#)
pdf_pagebox: [254](#), [1552](#)
\pdfpagebox primitive: [256](#)
pdf_pagebox_code: [254](#), [255](#), [256](#)
pdf_pages_attr: [248](#), [803](#)
\pdfpagesattr primitive: [248](#)
pdf_pages_attr_loc: [248](#), [249](#)
pdf_parent_outline: [789](#), [1563](#), [1628](#), [1629](#)
pdf_pk_mode: [248](#), [792](#)
\pdfpkmode primitive: [248](#)
pdf_pk_mode_loc: [248](#), [249](#)
pdf_pk_resolution: [254](#), [792](#)
\pdfpkresolution primitive: [256](#)
pdf_pk_resolution_code: [254](#), [255](#), [256](#)
pdf_prepend_kern: [254](#), [705](#), [1212](#)
\pdfprependkern primitive: [256](#)
pdf_prepend_kern_code: [254](#), [255](#), [256](#)
pdf_print: [683](#), [686](#), [692](#), [693](#), [698](#), [699](#), [702](#),
[727](#), [756](#), [758](#), [766](#), [767](#), [768](#), [769](#), [771](#), [772](#),

782, 784, 790, 803, 805, 808, 813, 814, 815,
 1600, 1630, 1637
`pdf_print_bp`: 690, 692, 693, 756, 1600, 1637
`pdf_print_char`: 686, 726
`pdf_print_fw_int`: 702, 813
`pdf_print_info`: 749, 794, 807
`pdf_print_int`: 683, 686, 690, 693, 698, 699, 702,
 766, 767, 769, 771, 784, 790, 795, 797, 803,
 805, 808, 814, 1600, 1630, 1637
`pdf_print_int_ln`: 683, 686, 699, 813, 815
`pdf_print_ln`: 685, 686, 692, 693, 698, 699, 727,
 756, 758, 766, 767, 768, 769, 771, 772, 781, 782,
 784, 786, 788, 790, 795, 797, 803, 805, 806, 807,
 808, 813, 814, 815, 1600, 1630, 1637
`pdf_print_mag_bp`: 690, 693, 769, 784, 785
`pdf_print_nl`: 683, 686, 693, 702, 814
`pdf_print_octal`: 686
`pdf_print_real`: 690, 692, 693, 758, 1637
`pdf_print_rect_spec`: 784, 785, 786
`pdf_print_resname_prefix`: 693, 766, 767, 1637
`pdf_print_str`: 702, 805, 1630
`pdf_print_str_ln`: 702, 1563
`pdf_print_toks`: 727
`pdf_print_toks_ln`: 727, 756, 763, 769, 772, 778,
 781, 782, 789, 803, 804, 806, 814, 815, 1630
`pdf_print_two`: 686
`pdf_protrude_chars`: 254, 1027, 1057, 1063
`\pdfprotrudechars` primitive: 256
`pdf_protrude_chars_code`: 254, 255, 256
`pdf_ptex_use_underscore`: 254, 673
`\pdfptexuseunderscore` primitive: 256
`pdf_ptex_use_underscore_code`: 254, 255, 256
`pdf_ptr`: 680, 681, 685, 686, 698, 699, 772
`pdf_px_dimen`: 265, 481, 672
`\pdfpxdimen` primitive: 266
`pdf_px_dimen_code`: 265, 266
`pdf_quick_out`: 680, 686, 699, 702
`pdf_read_dummy_font`: 693
`pdf_rectangle`: 693, 781, 782
`\pdfrefobj` primitive: 1524
`pdf_refobj_node`: 1524, 1526, 1528, 1546, 1603,
 1604, 1605, 1639, 1645
`pdf_refobj_node_size`: 695, 1546, 1604, 1605
`\pdfrefxform` primitive: 1524
`pdf_refxform_node`: 1005, 1524, 1526, 1528, 1549,
 1603, 1604, 1605, 1606, 1607, 1608, 1609, 1611,
 1612, 1637, 1639, 1645
`pdf_refxform_node_size`: 695, 1549, 1604, 1605
`\pdfrefximage` primitive: 1524
`pdf_refximage_node`: 1005, 1524, 1526, 1528, 1554,
 1603, 1604, 1605, 1606, 1607, 1608, 1609, 1611,
 1612, 1637, 1639, 1645
`pdf_refximage_node_size`: 695, 1554, 1604, 1605
`\pdfrestore` primitive: 1524
`pdf_restore_node`: 1524, 1526, 1528, 1542, 1603,
 1604, 1605, 1639, 1645
`pdf_restore_node_size`: 695, 1542, 1604, 1605
`pdf_retval`: 450, 1544, 1583
`\pdfretval` primitive: 442
`pdf retval_code`: 442, 443, 450
`pdf_right`: 695, 781, 782, 785, 1630, 1635
`pdf_room`: 680, 686, 699, 702
`\pdfrunninglinkoff` primitive: 1524
`pdf_running_link_off_node`: 1524, 1526, 1528, 1597,
 1603, 1604, 1605, 1639, 1645
`\pdfrunninglinkon` primitive: 1524
`pdf_running_link_on_node`: 1524, 1526, 1528, 1598,
 1603, 1604, 1605, 1639, 1645
`\pdfsave` primitive: 1524
`pdf_save_node`: 1524, 1526, 1528, 1541, 1603, 1604,
 1605, 1639, 1645
`pdf_save_node_size`: 695, 1541, 1604, 1605
`pdf_save_offset`: 680, 685, 813, 815
`\pdfsavepos` primitive: 1524
`pdf_save_pos_node`: 1524, 1526, 1528, 1576, 1603,
 1604, 1605, 1620, 1639, 1645
`pdf_scan_ext_toks`: 496
`pdf_seek_write_length`: 681, 685, 696
`pdf_set_font`: 693
`pdf_set_origin`: 692, 693
`pdf_set_origin_temp`: 692, 693
`pdf_set_rule`: 693, 726, 734, 743
`pdf_set_textmatrix`: 692, 693
`\pdfsetmatrix` primitive: 1524
`pdf_setmatrix_data`: 695, 727, 1540, 1603, 1604,
 1605
`pdf_setmatrix_node`: 1524, 1526, 1528, 1540, 1603,
 1604, 1605, 1639, 1645
`pdf_setmatrix_node_size`: 695, 1540, 1604, 1605
`\pdfshellescape` primitive: 442
`pdf_shell_escape_code`: 442, 443, 450
`pdf_ship_out`: 727, 749, 750, 775, 791, 1555,
 1623, 1640
`\pdfsnaprefpoint` primitive: 1524
`pdf_snap_ref_point_node`: 1524, 1526, 1528, 1572,
 1603, 1604, 1605, 1639, 1645
`pdf_snapx_refpos`: 1570, 1642
`\pdfsnappy` primitive: 1524
`\pdfsnappycomp` primitive: 1524
`pdf_snappy_comp_node`: 1145, 1177, 1524, 1526,
 1528, 1575, 1603, 1604, 1605, 1637, 1639, 1645
`pdf_snappy_node`: 1145, 1177, 1524, 1526, 1528,
 1574, 1603, 1604, 1605, 1637, 1639, 1645

pdf_snapy_refpos: [1570](#), [1637](#), [1642](#)
\pdfspacefont primitive: [1524](#)
pdf_space_font_code: [1524](#), [1526](#), [1528](#)
pdf_space_font_name: [693](#), [1599](#), [1628](#), [1629](#)
pdf_special: [727](#), [1639](#), [1645](#)
\pdfstartlink primitive: [1524](#)
pdf_start_link_node: [783](#), [1524](#), [1526](#), [1528](#), [1556](#),
[1560](#), [1603](#), [1604](#), [1605](#), [1631](#), [1632](#), [1635](#),
[1639](#), [1645](#)
\pdfstartthread primitive: [1524](#)
pdf_start_thread_node: [786](#), [1524](#), [1526](#), [1528](#), [1556](#),
[1568](#), [1603](#), [1604](#), [1605](#), [1637](#), [1639](#), [1645](#)
pdf_str_entry: [702](#), [1630](#)
pdf_str_entry_ln: [702](#), [807](#)
\pdfstrcmp primitive: [494](#)
pdf_strcmp_code: [494](#), [495](#), [497](#), [498](#)
pdf_stream_length: [685](#), [696](#)
pdf_stream_length_offset: [685](#), [696](#)
pdf_suppress_ptex_info: [254](#), [673](#), [807](#)
\pdfsuppressptexinfo primitive: [256](#)
pdf_suppress_ptex_info_code: [254](#), [255](#), [256](#)
pdf_suppress_warning_dup_dest: [254](#), [1564](#)
\pdfsuppresswarningdupdest primitive: [256](#)
pdf_suppress_warning_dup_dest_code: [254](#), [255](#), [256](#)
pdf_suppress_warning_dup_map: [254](#), [673](#)
\pdfsuppresswarningdupmap primitive: [256](#)
pdf_suppress_warning_dup_map_code: [254](#), [255](#), [256](#)
pdf_suppress_warning_page_group: [254](#), [673](#)
\pdfsuppresswarningpagegroup primitive: [256](#)
pdf_suppress_warning_page_group_code: [254](#), [255](#),
[256](#)
pdf_text_procset: [701](#), [750](#), [753](#), [766](#), [768](#), [776](#), [777](#)
\pdfthread primitive: [1524](#)
pdf_thread_attr: [695](#), [1556](#), [1603](#), [1604](#), [1605](#), [1637](#)
pdf_thread_dp: [739](#), [1628](#), [1637](#)
pdf_thread_ht: [1628](#), [1637](#)
pdf_thread_id: [695](#), [1566](#), [1603](#), [1604](#), [1605](#), [1637](#)
pdf_thread_level: [739](#), [1628](#), [1637](#)
pdf_thread_margin: [265](#), [1637](#)
\pdfthreadmargin primitive: [266](#)
pdf_thread_margin_code: [265](#), [266](#)
pdf_thread_named_id: [695](#), [1566](#), [1603](#), [1604](#),
[1605](#), [1637](#)
pdf_thread_node: [1524](#), [1526](#), [1528](#), [1556](#), [1567](#),
[1603](#), [1604](#), [1605](#), [1639](#), [1645](#)
pdf_thread_node_size: [695](#), [1567](#), [1568](#), [1604](#),
[1605](#), [1637](#)
pdf_thread_wd: [1628](#), [1637](#)
pdf_tj_start_h: [691](#), [692](#), [693](#)
pdf_toks: [248](#)
pdf_top: [695](#), [781](#), [782](#), [784](#), [785](#), [1630](#), [1635](#), [1637](#)
pdf_tracing_fonts: [192](#), [254](#)

\pdftracingfonts primitive: [256](#)
pdf_tracing_fonts_code: [254](#), [255](#), [256](#)
\pdftrailer primitive: [1524](#)
pdf_trailer_code: [1524](#), [1526](#), [1528](#)
\pdftrailerid primitive: [1524](#)
pdf_trailer_id_code: [1524](#), [1526](#), [1528](#)
pdf_trailer_id_toks: [814](#), [815](#), [1582](#), [1628](#), [1629](#)
pdf_trailer_toks: [814](#), [815](#), [1581](#), [1628](#), [1629](#)
\pdfunescapehex primitive: [494](#)
pdf_unescape_hex_code: [494](#), [495](#), [497](#)
pdf_unique_resname: [254](#), [792](#)
\pdfuniqueeresname primitive: [256](#)
pdf_unique_resname_code: [254](#), [255](#), [256](#)
pdf_use_font: [692](#), [693](#)
pdf_v: [691](#), [692](#), [693](#)
pdf_v_origin: [265](#), [672](#), [755](#)
\pdfvorigin primitive: [266](#)
pdf_v_origin_code: [265](#), [266](#)
pdf_version_written: [680](#), [681](#), [683](#)
pdf_vlist_node: [738](#)
pdf_vlist_out: [727](#), [728](#), [733](#), [737](#), [738](#), [742](#), [746](#),
[751](#), [1555](#)
pdf_warning: [683](#), [686](#), [692](#), [705](#), [706](#), [794](#), [795](#), [797](#),
[799](#), [801](#), [1537](#), [1544](#), [1552](#), [1564](#), [1600](#), [1635](#)
pdf_width: [695](#), [730](#), [1549](#), [1554](#), [1556](#), [1565](#), [1601](#),
[1603](#), [1604](#), [1606](#), [1607](#), [1608](#), [1609](#), [1630](#), [1635](#),
[1636](#), [1637](#), [1646](#), [1647](#)
pdf_write_image: [778](#), [779](#), [1623](#)
pdf_write_obj: [772](#), [773](#), [1623](#)
pdf_x: [691](#), [693](#), [784](#), [785](#), [1637](#)
\pdfxform primitive: [1524](#)
pdf_xform_code: [1524](#), [1526](#), [1528](#), [1623](#)
pdf_xform_count: [1504](#), [1505](#), [1548](#), [1628](#), [1629](#)
pdf_xform_depth: [752](#), [756](#), [1628](#), [1641](#)
pdf_xform_height: [752](#), [756](#), [1628](#), [1641](#)
pdf_xform_list: [750](#), [753](#), [764](#), [767](#), [775](#), [776](#),
[777](#), [1628](#), [1637](#)
\pdfxformname primitive: [494](#)
pdf_xform_name_code: [494](#), [495](#), [497](#), [498](#)
pdf_xform_objnum: [695](#), [1549](#), [1603](#), [1637](#)
pdf_xform_width: [752](#), [756](#), [1628](#)
\pdfximage primitive: [1524](#)
\pdfximagebbox primitive: [494](#)
pdf_ximage_bbox_code: [494](#), [495](#), [497](#), [498](#)
pdf_ximage_code: [1524](#), [1526](#), [1528](#), [1623](#)
pdf_ximage_count: [1504](#), [1505](#), [1552](#), [1628](#), [1629](#)
pdf_ximage_list: [750](#), [753](#), [764](#), [767](#), [775](#), [776](#),
[777](#), [779](#), [1628](#), [1637](#)
pdf_ximage_objnum: [695](#), [1554](#), [1603](#), [1637](#)
pdf_y: [691](#), [693](#), [784](#), [785](#), [1637](#)
pdfassert: [692](#), [693](#), [705](#), [712](#), [725](#), [730](#), [801](#),
[825](#), [1635](#), [1636](#)

pdfdraftmode: 748
pdfmapfile: 1590
pdfmapline: 1591
pdfmaplinesp: 693
pdfmem_bead_size: 695, 1637
pdfmem_obj_size: 695, 1544
pdfmem_outline_size: 695, 1563
pdfmem_xform_size: 695, 1548
pdfmem_ximage_size: 695, 1552
pdfoutput: 747
pdfpagegroupval: 761
\pdfprimitive primitive: 287
\pdfprimitive primitive (internalized): 394
pdfsetmatrix: 727
pdfshipoutbegin: 757
pdfshipoutend: 760
PDFTEX: 2
pdfTeX_banner: 2
pdftex_banner: 498, 807, 811
\pdftexbanner primitive: 494
pdftex_banner_code: 494, 495, 497, 498
pdftex_convert_codes: 494
pdftex_first_dimen_code: 265
pdftex_first_expand_code: 494
pdftex_first_extension_code: 1524, 1620
pdftex_first_integer_code: 254
pdftex_first_loc: 248
pdftex_first_rint_code: 442
pdftex_last_dimen_code: 265
pdftex_last_extension_code: 1524, 1620
pdftex_last_item_codes: 442
pdftex_revision: 2, 498, 808
\pdftexrevision primitive: 494
pdftex_revision_code: 494, 495, 497, 498
pdftex_version: 2, 450, 808
\pdftexversion primitive: 442
pdftex_version_code: 442, 443, 450
pdftex_version_string: 2
pen: 902, 937, 943, 1053, 1067
penalties: 1280
penalties: 902, 943
penalty: 175, 176, 212, 251, 450, 992, 1042, 1067, 1150, 1173, 1177, 1187, 1188, 1190, 1866
\penalty primitive: 287
penalty_node: 175, 176, 201, 220, 224, 450, 674, 906, 937, 943, 992, 993, 1005, 1013, 1032, 1042, 1055, 1076, 1145, 1150, 1173, 1177, 1187, 1188, 1190, 1285
pg_field: 230, 231, 236, 237, 448, 1422
pi: 1005, 1007, 1027, 1032, 1035, 1147, 1149, 1150, 1151, 1171, 1177, 1182, 1183
pk_dpi: 792, 1628
pk_scale_factor: 691, 792
plain: 547, 550, 1511
Plass, Michael Frederick: 2, 989
Please type...: 382, 556
Please use \mathaccent...: 1344
PLtoTF: 587
plus: 488
point_token: 464, 466, 474, 478
pointer: 133, 134, 136, 138, 141, 142, 143, 148, 149, 154, 157, 162, 163, 165, 169, 170, 171, 172, 174, 176, 183, 185, 190, 216, 218, 219, 220, 222, 230, 231, 236, 270, 274, 275, 278, 281, 285, 297, 298, 299, 300, 301, 303, 306, 317, 319, 321, 327, 328, 330, 345, 347, 355, 358, 388, 408, 414, 415, 433, 439, 476, 487, 489, 490, 491, 496, 499, 508, 515, 523, 524, 575, 586, 609, 619, 632, 634, 642, 647, 657, 666, 673, 680, 686, 693, 699, 700, 705, 706, 708, 727, 729, 738, 750, 772, 785, 791, 819, 821, 823, 829, 844, 855, 862, 864, 865, 867, 868, 880, 881, 882, 885, 887, 891, 892, 893, 895, 896, 898, 902, 910, 911, 912, 913, 914, 919, 925, 928, 932, 938, 946, 948, 950, 963, 967, 975, 976, 990, 997, 999, 1002, 1004, 1005, 1006, 1009, 1038, 1048, 1053, 1069, 1077, 1078, 1083, 1084, 1089, 1103, 1111, 1145, 1147, 1154, 1157, 1159, 1170, 1171, 1189, 1207, 1209, 1221, 1242, 1246, 1252, 1253, 1257, 1264, 1271, 1279, 1283, 1288, 1291, 1295, 1297, 1301, 1316, 1329, 1333, 1338, 1352, 1354, 1362, 1369, 1372, 1376, 1389, 1414, 1425, 1435, 1466, 1471, 1480, 1481, 1525, 1528, 1529, 1537, 1545, 1550, 1556, 1562, 1573, 1577, 1600, 1602, 1615, 1617, 1620, 1628, 1630, 1632, 1635, 1636, 1637, 1683, 1705, 1719, 1723, 1733, 1738, 1741, 1744, 1750, 1753, 1756, 1757, 1773, 1777, 1782, 1815, 1816, 1819, 1821, 1822, 1823, 1825, 1835, 1837, 1838, 1839, 1840, 1841, 1842, 1859
pointer_node_size: 1820, 1821, 1837, 1841
Poirot, Hercule: 1461
pool_file: 47, 50, 51, 52, 53
pool_name: 11, 51
pool_pointer: 38, 39, 45, 46, 59, 60, 69, 70, 286, 433, 490, 491, 496, 539, 545, 629, 666, 686, 693, 702, 706, 749, 750, 793, 1106, 1111, 1537, 1587, 1615, 1753
pool_ptr: 38, 39, 41, 42, 43, 44, 47, 52, 58, 70, 216, 279, 490, 491, 496, 497, 542, 551, 645, 686, 727, 1487, 1488, 1512, 1514, 1519, 1615, 1688, 1754
pool_size: 11, 38, 42, 52, 58, 216, 551, 686, 727, 1488, 1514, 1519, 1615
pop: 611, 612, 613, 617, 628, 635, 670, 719, 726
pop_alignment: 948, 976
pop_input: 344, 346, 351

pop_lig_stack: 1087, 1088
pop_link_level: 1635
pop_LR: 1705, 1708, 1711, 1712, 1717, 1718, 1728, 1735, 1737, 1739
pop_nest: 235, 972, 975, 988, 992, 1203, 1264, 1274, 1278, 1297, 1346, 1362, 1384, 1732
pop_node: 1005
pop_packet_state: 725
positive: 107, 689
post: 610, 612, 613, 617, 618, 670, 712
post_break: 163, 193, 213, 220, 224, 674, 823, 1005, 1016, 1034, 1058, 1060, 1093, 1217, 1297
post_disc_break: 1053, 1057, 1060
post_display_penalty: 254, 1383, 1384
\postdisplaypenalty primitive: 256
post_display_penalty_code: 254, 255, 256
post_line_break: 1052, 1053, 1705
post_post: 612, 613, 617, 618, 670
pre: 610, 612, 613, 645, 714
pre_adjust_head: 180, 1065, 1066, 1254, 1263, 1377, 1383
pre_adjust_tail: 823, 825, 829, 830, 831, 972, 1065, 1066, 1254, 1263, 1377
pre_break: 163, 193, 213, 220, 224, 674, 823, 1005, 1034, 1045, 1057, 1058, 1061, 1092, 1217, 1295, 1297
pre_display_direction: 254, 1316, 1377, 1744
\predisplaydirection primitive: 1657
pre_display_direction_code: 254, 1323, 1657, 1659
pre_display_penalty: 254, 1381, 1384
\predisplaypenalty primitive: 256
pre_display_penalty_code: 254, 255, 256
pre_display_size: 265, 1316, 1323, 1326, 1381, 1733
\predisplaysize primitive: 266
pre_display_size_code: 265, 266, 1323
pre_kern: 1295
pre_t: 1376, 1377, 1383
preamble: 944, 950
preamble: 946, 947, 948, 953, 962, 977, 980
preamble of DVI file: 645
precedes_break: 166, 1044, 1150, 1177
prefix: 227, 1386, 1387, 1388, 1389, 1770
prefixed_command: 1388, 1389, 1448
prepare_mag: 310, 483, 645, 670, 690, 693, 752, 758, 792, 1513
prepend_nl: 686
pretolerance: 254, 1004, 1039
\pretolerance primitive: 256
pretolerance_code: 254, 255, 256
prev_active_width: 999
prev_auto_breaking: 999
prev_break: 997, 1021, 1022, 1053, 1054
prev_char_p: 823, 825, 828, 999, 1003, 1017, 1018, 1039, 1042, 1043, 1046, 1047
prev_depth: 230, 231, 233, 444, 855, 951, 962, 963, 1202, 1234, 1261, 1277, 1345, 1384, 1420, 1421, 1481
\prevdepth primitive: 442
prev_dp: 1147, 1149, 1150, 1151, 1153, 1612
prev_graf: 230, 231, 233, 234, 448, 990, 992, 1040, 1053, 1065, 1067, 1269, 1327, 1378, 1420
\prevgraf primitive: 287
prev_legal: 999, 1039
prev_p: 647, 648, 650, 729, 731, 732, 999, 1005, 1038, 1039, 1042, 1043, 1044, 1045, 1145, 1146, 1147, 1150, 1189, 1191, 1194, 1199, 1721, 1722
prev_prev_legal: 999
prev_prev_r: 1006, 1008, 1019, 1020, 1036
prev_r: 1005, 1006, 1008, 1019, 1020, 1021, 1027, 1030, 1036
prev_rightmost: 498, 1005, 1057, 1545
prev_s: 1038, 1071, 1073
prev_tail: 231, 232, 1217
prim: 275, 276, 282, 1496, 1497
prim_base: 275, 281
prim_eq_level: 275, 286
prim_eq_level_field: 275
prim_eq_type: 275, 286, 394, 395, 527, 1223
prim_eq_type_field: 275
prim_eqtb: 286
prim_eqtb_base: 240, 275, 284, 285, 395, 1223
prim_equiv: 275, 286, 394, 395, 527, 1223
prim_equiv_field: 275
prim_is_full: 275, 282
prim_lookup: 281, 286, 394, 395, 527, 1223
prim_next: 275, 276, 281, 282
prim_prime: 275, 281, 283
prim_size: 240, 275, 276, 277, 282, 283, 1496, 1497
prim_text: 275, 276, 281, 282, 284, 285
prim_used: 275, 277, 282
prim_val: 286
primitive: 244, 248, 256, 266, 286, 287, 288, 320, 356, 402, 410, 437, 442, 494, 513, 517, 579, 956, 1160, 1230, 1236, 1249, 1266, 1285, 1292, 1319, 1334, 1347, 1356, 1366, 1386, 1397, 1400, 1408, 1428, 1432, 1440, 1450, 1455, 1464, 1469, 1511, 1512, 1524, 1649, 1657, 1663, 1666, 1669, 1672, 1675, 1684, 1686, 1689, 1692, 1697, 1701, 1747, 1759, 1762, 1770, 1778, 1801, 1805, 1809, 1861, 1864
primitive_size: 275
print: 54, 59, 60, 62, 63, 68, 70, 71, 73, 84, 85, 86, 89, 91, 94, 95, 121, 192, 193, 195, 196, 200, 201, 202, 203, 204, 205, 206, 208, 209, 210, 211,

213, 215, 229, 236, 237, 243, 251, 252, 255, 265, 269, 284, 285, 306, 310, 316, 320, 321, 328, 339, 340, 345, 358, 360, 361, 385, 399, 421, 422, 424, 426, 454, 480, 482, 485, 491, 498, 528, 535, 556, 560, 562, 587, 593, 606, 608, 645, 666, 667, 670, 674, 686, 693, 705, 706, 712, 714, 717, 727, 750, 772, 794, 795, 797, 799, 801, 836, 839, 842, 850, 851, 853, 868, 870, 873, 899, 952, 1005, 1022, 1032, 1113, 1145, 1155, 1162, 1163, 1164, 1177, 1183, 1188, 1192, 1201, 1227, 1242, 1273, 1310, 1344, 1391, 1410, 1415, 1435, 1437, 1439, 1473, 1474, 1476, 1487, 1489, 1496, 1498, 1500, 1502, 1504, 1508, 1514, 1515, 1518, 1519, 1526, 1564, 1600, 1601, 1603, 1661, 1662, 1679, 1680, 1681, 1691, 1704, 1713, 1755, 1765, 1774, 1776, 1777, 1823, 1852
print_ASCII: 68, 192, 194, 320, 608, 674, 867, 899
print_char: 58, 59, 60, 64, 65, 66, 67, 69, 70, 82, 91, 94, 95, 103, 132, 189, 190, 192, 193, 194, 195, 196, 202, 204, 205, 206, 207, 208, 209, 210, 211, 214, 236, 237, 241, 247, 251, 252, 253, 260, 269, 270, 273, 284, 288, 306, 307, 316, 318, 321, 328, 335, 339, 384, 411, 498, 535, 562, 563, 587, 608, 645, 666, 667, 670, 674, 750, 794, 867, 899, 1022, 1032, 1110, 1183, 1188, 1243, 1247, 1390, 1391, 1458, 1472, 1473, 1474, 1489, 1498, 1500, 1502, 1508, 1513, 1515, 1520, 1601, 1602, 1603, 1661, 1662, 1679, 1680, 1681, 1730, 1755, 1765, 1823
print_cmd_chr: 241, 251, 288, 318, 320, 321, 345, 358, 444, 454, 529, 536, 1227, 1244, 1306, 1390, 1391, 1415, 1515, 1519, 1656, 1679, 1681, 1691, 1765, 1769, 1776, 1777, 1823
print_creation_date: 809
print_cs: 284, 315, 336, 427
print_current_string: 70, 200, 868
print_delimiter: 867, 872, 873
print_err: 72, 73, 93, 94, 95, 98, 121, 310, 358, 360, 368, 396, 399, 421, 422, 424, 429, 434, 441, 444, 454, 459, 460, 461, 462, 463, 468, 471, 472, 480, 482, 485, 486, 497, 501, 502, 505, 512, 526, 529, 536, 556, 587, 604, 606, 669, 683, 686, 899, 952, 959, 960, 968, 1002, 1113, 1114, 1137, 1138, 1139, 1140, 1153, 1155, 1170, 1181, 1186, 1192, 1201, 1204, 1205, 1225, 1227, 1242, 1244, 1246, 1247, 1256, 1260, 1262, 1273, 1277, 1288, 1298, 1299, 1305, 1306, 1307, 1310, 1313, 1337, 1339, 1344, 1355, 1361, 1370, 1373, 1375, 1385, 1390, 1391, 1393, 1403, 1410, 1414, 1415, 1419, 1421, 1422, 1430, 1436, 1437, 1461, 1476, 1482, 1513, 1539, 1619, 1656, 1696, 1765, 1769, 1782, 1784, 1811
print_esc: 63, 86, 192, 194, 201, 202, 205, 206, 207, 208, 209, 210, 212, 213, 214, 215, 243, 245, 247, 249, 251, 252, 253, 255, 257, 260, 265, 267, 269, 284, 285, 288, 314, 315, 316, 345, 357, 399, 403, 411, 443, 454, 454, 495, 512, 514, 518, 526, 606, 867, 870, 871, 872, 873, 875, 952, 957, 968, 1032, 1113, 1137, 1138, 1155, 1161, 1163, 1186, 1192, 1205, 1231, 1237, 1243, 1247, 1250, 1267, 1273, 1277, 1286, 1293, 1298, 1307, 1310, 1313, 1321, 1335, 1344, 1357, 1367, 1370, 1387, 1391, 1398, 1401, 1409, 1419, 1422, 1429, 1433, 1441, 1451, 1456, 1465, 1470, 1473, 1500, 1515, 1526, 1602, 1603, 1650, 1658, 1659, 1664, 1667, 1670, 1673, 1676, 1679, 1681, 1685, 1687, 1690, 1691, 1693, 1698, 1700, 1702, 1748, 1760, 1763, 1764, 1765, 1771, 1777, 1779, 1802, 1806, 1823, 1832, 1833, 1862, 1865
print_fam_and_char: 867, 868, 872
print_file_name: 544, 556, 587, 794, 1500, 1603
print_font_and_char: 194, 201, 211
print_font_identifier: 192, 194, 674, 801
print_glue: 195, 196, 203, 204
print_group: 1661, 1662, 1679, 1774, 1777
print_hex: 67, 867, 1401
print_ID: 814, 815
print_ID_alt: 814, 815
print_if_line: 321, 1691, 1776, 1777
print_int: 65, 84, 91, 94, 103, 132, 186, 187, 188, 189, 190, 192, 203, 206, 212, 213, 214, 236, 237, 245, 247, 249, 251, 252, 253, 257, 260, 267, 269, 273, 307, 310, 321, 335, 358, 426, 491, 498, 535, 562, 587, 606, 645, 666, 667, 670, 705, 706, 727, 750, 794, 795, 797, 799, 836, 839, 843, 850, 851, 854, 867, 899, 1022, 1032, 1110, 1163, 1183, 1186, 1188, 1201, 1205, 1277, 1410, 1474, 1487, 1489, 1496, 1498, 1502, 1504, 1508, 1515, 1519, 1539, 1564, 1600, 1602, 1603, 1637, 1661, 1679, 1681, 1691, 1713, 1822, 1823
print_length_param: 265, 267, 269
print_ln: 57, 58, 59, 61, 62, 71, 86, 89, 90, 132, 200, 216, 236, 254, 263, 318, 328, 336, 339, 352, 382, 385, 427, 510, 560, 563, 666, 667, 683, 686, 693, 750, 795, 797, 799, 801, 836, 839, 842, 843, 850, 851, 853, 854, 868, 1005, 1027, 1057, 1145, 1163, 1443, 1458, 1487, 1489, 1496, 1498, 1502, 1504, 1520, 1564, 1600, 1617, 1679, 1691, 1713, 1730, 1755, 1774, 1776, 1777
print_locs: 185
print_mark: 194, 214, 1564, 1603
print_meaning: 318, 498, 1472
print_mod_date: 810
print_mode: 229, 236, 321, 1227

print_nl: 62, 73, 82, 84, 85, 90, 186, 187, 188, 189, 190, 236, 237, 263, 273, 307, 310, 321, 328, 333, 335, 336, 345, 382, 426, 556, 560, 608, 666, 667, 669, 670, 693, 712, 714, 717, 727, 750, 772, 794, 836, 842, 843, 850, 853, 854, 1022, 1032, 1033, 1039, 1110, 1163, 1164, 1169, 1183, 1188, 1299, 1472, 1474, 1475, 1500, 1502, 1508, 1513, 1515, 1518, 1617, 1637, 1679, 1691, 1713, 1774, 1776, 1777
print_param: 255, 257, 260
print_plus: 1162
print_plus_end: 1162
print_roman_int: 69, 498
print_rule_dimen: 194, 205, 1601, 1603
print_sa_num: 1822, 1823, 1832, 1833
print_scaled: 103, 121, 132, 192, 194, 195, 196, 201, 202, 206, 209, 210, 237, 269, 491, 498, 587, 842, 853, 873, 1162, 1163, 1164, 1183, 1188, 1437, 1439, 1500, 1603, 1637, 1680, 1681, 1823, 1852
print_size: 875, 899, 1409
print_skip_param: 207, 243, 245, 247
print_spec: 196, 206, 207, 208, 247, 491, 1603, 1637, 1823
print_style: 866, 870, 1348
print_subsidary_data: 868, 872, 873
print_the_digs: 64, 65, 67
print_totals: 236, 1162, 1163, 1183
print_two: 66, 562, 645
print_word: 132, 1519
print_write_whatsit: 1602, 1603
printed_node: 997, 1032, 1033, 1034, 1040
privileged: 1229, 1232, 1308, 1318
Producer: 807
producer_given: 807
prompt_file_name: 556, 558, 561, 563, 684, 1508, 1622
prompt_input: 71, 83, 87, 382, 385, 510, 556
protected: 1770
\protected primitive: 1770
protected_token: 311, 415, 504, 1391, 1473, 1772
prune_movements: 642, 647, 657
prune_page_top: 1145, 1154, 1198
pseudo: 54, 57, 58, 59, 338
pseudo_close: 351, 1757, 1758
pseudo_files: 1750, 1751, 1754, 1756, 1757, 1758
pseudo_input: 384, 1756
pseudo_start: 1749, 1752, 1753
pstack: 414, 416, 422, 426
pt: 479
ptmp: 1053, 1057
punct_noad: 858, 866, 872, 874, 904, 928, 937, 1334, 1335
push: 611, 612, 613, 617, 619, 628, 635, 643, 647, 657, 719, 721, 726
push_alignment: 948, 950
push_input: 343, 345, 347, 350
push_link_level: 1635
push_LR: 1705, 1708, 1711, 1717, 1728, 1737, 1739
push_math: 1314, 1317, 1323, 1331, 1350, 1352, 1369
push_nest: 234, 950, 962, 963, 1202, 1261, 1269, 1277, 1295, 1297, 1314, 1345, 1378
push_node: 1005
push_packet_state: 725
put: 26, 29, 1483
put_LR: 1705, 1710
put_rule: 612, 613, 661, 719, 726
put_sa_ptr: 1819, 1831
put1: 612, 710, 719, 726
put2: 612
put3: 612
put4: 612
px: 481
q: 112, 114, 122, 141, 143, 148, 149, 162, 169, 170, 171, 185, 190, 220, 222, 236, 297, 314, 337, 358, 388, 415, 433, 439, 476, 487, 489, 490, 491, 499, 508, 523, 524, 634, 689, 700, 727, 823, 881, 882, 885, 888, 896, 902, 910, 911, 912, 913, 914, 919, 925, 928, 932, 938, 967, 976, 1002, 1006, 1038, 1053, 1078, 1083, 1111, 1125, 1130, 1134, 1136, 1137, 1145, 1147, 1171, 1189, 1221, 1246, 1257, 1271, 1283, 1297, 1301, 1316, 1362, 1369, 1376, 1389, 1414, 1480, 1481, 1528, 1615, 1617, 1637, 1683, 1723, 1738, 1744, 1753, 1757, 1782, 1815, 1819, 1821, 1822, 1825, 1837
qi: 130, 498, 571, 575, 590, 596, 600, 603, 609, 643, 648, 823, 929, 1084, 1085, 1088, 1090, 1100, 1135, 1136, 1158, 1185, 1186, 1211, 1212, 1215, 1216, 1218, 1278, 1329, 1333, 1338, 1343, 1487, 1503, 1671, 1754, 1769, 1856, 1858
qo: 130, 177, 192, 194, 203, 206, 580, 596, 603, 629, 643, 648, 674, 823, 867, 884, 898, 899, 917, 928, 931, 1073, 1074, 1075, 1080, 1086, 1100, 1122, 1158, 1163, 1185, 1195, 1198, 1216, 1488, 1502, 1503, 1661, 1858
qqqq: 128, 131, 132, 576, 580, 595, 600, 601, 710, 823, 859, 889, 917, 928, 1086, 1216, 1359, 1483, 1484, 1754, 1756
quad: 573, 584
quad_code: 573, 584
quarterword: 128, 131, 162, 271, 286, 293, 298, 299, 301, 303, 320, 322, 345, 619, 857, 882, 885, 887, 888, 900, 914, 925, 1053, 1098, 1120, 1121, 1124, 1137, 1239, 1257, 1283, 1656, 1679,

1777, 1815, 1835, 1837
`\quitvmode` primitive: 1266
`quotient`: 1796, 1797
`qw`: 586, 590, 596, 600, 603
`r`: 108, 122, 141, 143, 149, 222, 236, 388, 415, 439, 491, 508, 524, 727, 823, 844, 882, 896, 902, 928, 967, 976, 1005, 1038, 1053, 1078, 1130, 1143, 1145, 1147, 1171, 1189, 1257, 1283, 1301, 1338, 1376, 1414, 1528, 1555, 1615, 1617, 1733, 1744, 1753, 1756, 1782, 1799
`R_code`: 165, 210, 1717, 1731
`r_count`: 1089, 1091, 1095
`r_hyf`: 1068, 1069, 1071, 1076, 1079, 1100, 1609, 1610
`r_type`: 902, 903, 904, 905, 936, 942, 943
`radical`: 226, 287, 288, 1224, 1340
`\radical` primitive: 287
`radical_noad`: 859, 866, 872, 874, 909, 937, 1341
`radical_noad_size`: 859, 874, 937, 1341
`radix`: 388, 464, 465, 466, 470, 471, 474
`radix_backup`: 388
`\raise` primitive: 1249
 Ramshaw, Lyle Harold: 565
`random_seed`: 110, 450, 1517, 1585
`\pdfrandomseed` primitive: 442
`random_seed_code`: 442, 443, 450
`randoms`: 110, 124, 125, 126, 127
`rbrace_ptr`: 415, 425, 426
`rc`: 823
`read`: 52, 53, 1518, 1519
`\read` primitive: 287
`read_expand_font`: 705, 1535
`read_file`: 506, 511, 512, 1453
`read_font_info`: 586, 590, 693, 705, 706, 712, 1218, 1435
`read_image`: 1552
`\readline` primitive: 1759
`read_ln`: 52
`read_open`: 506, 507, 509, 511, 512, 527, 1453
`read_sixteen`: 590, 591, 594
`read_to_cs`: 227, 287, 288, 1388, 1403, 1759
`read_toks`: 325, 508, 1403
`ready_already`: 1511, 1512
`real`: 3, 109, 128, 200, 204, 647, 657, 729, 738, 1301, 1303, 1637, 1723
`real addition`: 1303
`real division`: 834, 840, 849, 852, 986, 987, 1301, 1303
`real multiplication`: 132, 204, 653, 662, 735, 744, 985, 1303, 1638
`real_font_type`: 703, 712
`rebox`: 891, 920, 926

`reconstitute`: 1082, 1083, 1090, 1092, 1093, 1094, 1209
`recursion`: 76, 78, 191, 198, 216, 220, 221, 388, 428, 433, 524, 553, 619, 646, 868, 895, 896, 901, 930, 1126, 1134, 1136, 1513, 1623, 1682
`ref_count`: 415, 416, 427
`ref_link_node`: 1632, 1635, 1636
`reference counts`: 168, 218, 219, 221, 297, 313, 329, 1820, 1821
`reflected`: 643, 1722, 1738
`register`: 227, 437, 438, 439, 1388, 1399, 1402, 1413, 1414, 1415, 1823, 1832, 1834
`rejected_cur_p`: 999, 1039
`rel_noad`: 858, 866, 872, 874, 904, 937, 943, 1334, 1335
`rel_penalty`: 254, 858, 937
`\relpenalty` primitive: 256
`rel_penalty_code`: 254, 255, 256
`relax`: 225, 287, 288, 380, 395, 398, 430, 504, 532, 1223, 1402, 1784
`\relax` primitive: 287
`rem_byte`: 571, 580, 583, 596, 884, 889, 916, 925, 929, 1088, 1218
`remainder`: 104, 106, 107, 483, 484, 569, 570, 571, 892, 893
`remove_item`: 226, 1282, 1285, 1286
`remove_last_space`: 686, 790, 803, 805
`remove_pdffile`: 1513
`rep`: 572
`replace_count`: 163, 193, 213, 674, 1005, 1016, 1034, 1045, 1058, 1059, 1095, 1217, 1258, 1298
`report_illegal_case`: 1223, 1228, 1229, 1421, 1625
`reset`: 26, 27, 33
`reset_disc_width`: 1015, 1045
`reset_OK`: 27
`\pdfresettimer` primitive: 1524
`reset_timer_code`: 1524, 1526, 1528
`restart`: 15, 143, 144, 286, 363, 368, 379, 381, 382, 384, 395, 406, 439, 466, 928, 929, 958, 961, 965, 1329, 1393, 1782, 1783, 1788
`restore_active_width`: 999
`restore_cur_string`: 496, 497
`restore_old_value`: 290, 298, 304
`restore_sa`: 290, 304, 1837
`restore_trace`: 299, 305, 306, 1823
`restore_zero`: 290, 298, 300
`restrictedshell`: 450
`result`: 45, 46
`resume_after_display`: 976, 1377, 1378, 1384
`reswitch`: 15, 363, 365, 374, 388, 394, 489, 647, 648, 729, 731, 823, 825, 826, 902, 904, 1111,

1112, 1206, 1207, 1213, 1223, 1316, 1325, 1329,
 1552, 1722, 1723, 1724, 1727, 1765
return: 15, 16
return_sign: 122, 123
reverse: 3, 1721, 1722, 1723
reversed: 643, 1714, 1721
rewrite: 26, 27, 33
rewrite_OK: 27
rh: 128, 131, 132, 136, 231, 237, 239, 252, 274,
 275, 290, 861, 1098, 1135, 1817
right: 693
\right primitive: 1366
right_brace: 225, 311, 316, 320, 369, 379, 415, 468,
 500, 503, 961, 1112, 1138, 1245, 1430, 1683
right_brace_limit: 311, 347, 348, 418, 425, 426,
 500, 503, 1683
right_brace_token: 311, 361, 1243, 1305, 1404, 1618
right_delimiter: 859, 873, 924, 1359, 1360
right_hyphen_min: 254, 1269, 1378, 1624, 1625
\righthyphenmin primitive: 256
right_hyphen_min_code: 254, 255, 256
\rightmarginkern primitive: 494
right_margin_kern_code: 494, 495, 497, 498
right_noad: 863, 866, 872, 874, 901, 903, 904, 936,
 937, 938, 1362, 1366, 1369
right_ptr: 632, 633, 634, 642
right_pw: 823, 1005, 1057
right_side: 173, 498, 823, 1057
right_skip: 242, 1003, 1056, 1057, 1740, 1843
\rightskip primitive: 244
right_skip_code: 242, 243, 244, 498, 1057, 1062,
 1740, 1746
right_to_left: 643, 651, 654, 656, 660, 661, 665,
 733, 736, 737, 742, 743, 746, 1714, 1715, 1734
rightskip: 1057
right1: 612, 613, 634, 637, 643, 706, 719, 726
right2: 612, 637
right3: 612, 637
right4: 612, 637
rlink: 142, 143, 144, 145, 147, 148, 149, 150, 163,
 167, 182, 187, 948, 995, 997, 1489, 1490
\romannumeral primitive: 494
roman_numeral_code: 494, 495, 497, 498
root: 806, 814, 815, 1513
round: 3, 132, 204, 653, 662, 735, 744, 985,
 1303, 1638
round_decimals: 102, 103, 478
round_glue: 653, 1726
round_xn_over_d: 689, 690, 693, 705, 706, 823,
 1637
rover: 142, 143, 144, 145, 146, 147, 148, 149,
 150, 182, 187, 1489, 1490
rp: 822, 1005
\rpcode primitive: 1432
rp_code_base: 173, 452, 1431, 1432, 1433
rt_hit: 1083, 1084, 1087, 1088, 1210, 1212,
 1217, 1218
rule: 1550, 1552, 1556, 1565
rule_dp: 619, 650, 652, 654, 659, 661, 663, 732,
 734, 736, 741, 743, 745
rule_ht: 619, 650, 652, 654, 659, 661, 662, 663,
 664, 726, 732, 734, 736, 741, 743, 744, 745,
 1637, 1638
rule_node: 156, 157, 166, 193, 201, 220, 224, 650,
 654, 659, 663, 732, 736, 741, 745, 825, 827,
 845, 846, 906, 937, 981, 1017, 1018, 1042, 1046,
 1047, 1145, 1150, 1177, 1252, 1265, 1299, 1325,
 1550, 1637, 1725, 1733
rule_node_size: 156, 157, 220, 224, 1733
rule_save: 976, 980
rule_wd: 619, 650, 652, 653, 654, 655, 659, 661,
 663, 726, 732, 734, 735, 736, 741, 743, 745,
 1699, 1722, 1725, 1728, 1729
rules aligning with characters: 616
run: 1005
runaway: 138, 328, 360, 422, 512
Runaway...: 328
s: 45, 46, 58, 59, 60, 62, 63, 93, 94, 95, 103,
 108, 143, 148, 165, 195, 196, 281, 286, 306,
 415, 433, 496, 499, 508, 555, 556, 586, 666,
 693, 705, 727, 772, 807, 817, 823, 844, 864,
 875, 882, 896, 902, 914, 967, 976, 1006, 1038,
 1053, 1078, 1111, 1143, 1145, 1164, 1189, 1238,
 1239, 1301, 1316, 1376, 1414, 1435, 1457, 1529,
 1555, 1587, 1602, 1630, 1679, 1683, 1719, 1744,
 1753, 1782, 1821, 1823
s_out: 690, 693
sa_bot_mark: 1825, 1828, 1830
sa_chain: 290, 304, 1835, 1836, 1837, 1841
sa_def: 1839, 1840
sa_def_box: 1255, 1839
sa_define: 1404, 1405, 1414, 1839
sa_destroy: 1838, 1839, 1840, 1841
sa_dim: 1820, 1823
sa_first_mark: 1825, 1828, 1829, 1830
sa_index: 1815, 1820, 1821, 1822, 1837, 1838, 1841
sa_int: 453, 1415, 1820, 1821, 1823, 1837, 1839,
 1840, 1841
sa_lev: 1820, 1837, 1839, 1840, 1841
sa_level: 290, 304, 1835, 1836, 1837
sa_loc: 1837, 1841
sa_mark: 1154, 1189, 1515, 1816, 1817
sa_null: 1815, 1816, 1817, 1820
sa_num: 1820, 1822

sa_ptr: 441, 453, 1405, 1415, 1820, 1821, 1823, 1837, 1838, 1839, 1840, 1841
sa_ref: 1820, 1821, 1837
sa_restore: 304, 1841
sa_root: 1489, 1490, 1816, 1818, 1819, 1821
sa_save: 1837, 1839
sa_split_bot_mark: 1825, 1826, 1827
sa_split_first_mark: 1825, 1826, 1827
sa_top_mark: 1825, 1828, 1829
sa_type: 453, 1415, 1820, 1823, 1832
sa_used: 1815, 1819, 1820, 1821, 1825
sa_w_def: 1839, 1840
sa_word_define: 1414, 1839
save_active_width: 999
save_cond_ptr: 524, 526, 535
save_cs_ptr: 950, 953
save_cur_cs: 433, 1537
save_cur_h: 725
save_cur_string: 496, 497
save_cur_v: 725
save_cur_val: 476, 481
save_def_ref: 496, 497
save_font_list: 750, 776, 777
save_for_after: 302, 1449
save_h: 647, 651, 655, 656, 657, 660, 665, 729, 737, 1721, 1722
save_image_procset: 750, 776, 777
save_index: 290, 296, 298, 302, 304, 1679, 1774, 1777, 1837
save_level: 290, 291, 296, 298, 302, 304, 1679, 1777, 1837
save_link: 1006, 1033
save_loc: 647, 657
save_obj_list: 750, 776, 777
save_pdf_delta_h: 693
save_pointer: 1678, 1679, 1773
save_ptr: 290, 293, 294, 295, 296, 298, 302, 304, 305, 307, 817, 980, 1264, 1277, 1278, 1295, 1298, 1320, 1331, 1346, 1350, 1352, 1364, 1372, 1482, 1679, 1774, 1777, 1837
save_scanner_status: 388, 393, 394, 415, 496, 497, 520, 524, 527, 533, 1766
save_size: 11, 129, 293, 295, 1514, 1678
save_split_top_skip: 1189, 1191
save_stack: 221, 290, 292, 293, 295, 296, 297, 298, 299, 303, 304, 305, 307, 322, 398, 515, 817, 944, 1240, 1249, 1309, 1318, 1328, 1331, 1519, 1678
save_style: 896, 902, 930
save_tail: 231, 233, 705, 1211, 1217
save_text_procset: 750, 776, 777
save_type: 290, 296, 298, 302, 304, 1837
save_v: 647, 651, 656, 657, 660, 664, 665, 738, 742, 746, 1643, 1644
save_vbadness: 1189, 1194
save_vf_nf: 712, 715
save_vfuzz: 1189, 1194
save_warning_index: 415, 496, 497
save_xform_list: 750, 776, 777
save_ximage_list: 750, 776, 777
saved: 296, 817, 980, 1261, 1264, 1277, 1278, 1295, 1297, 1320, 1331, 1346, 1350, 1352, 1364, 1372, 1661, 1662, 1679, 1680, 1681
saved_pdf_cur_form: 774, 775
saved_pdf_gone: 685
saving_hyph_codes: 254, 1137
\savinghyphcodes primitive: 1657
saving_hyph_codes_code: 254, 1657, 1659
saving_vdiscards: 254, 1154, 1176, 1859
\savingvdiscards primitive: 1657
saving_vdiscards_code: 254, 1657, 1659
sc: 128, 131, 132, 153, 168, 177, 182, 231, 237, 265, 268, 269, 439, 446, 451, 576, 578, 580, 583, 584, 598, 600, 602, 607, 695, 705, 706, 876, 877, 951, 998, 999, 1008, 1019, 1020, 1024, 1026, 1036, 1037, 1066, 1220, 1327, 1384, 1425, 1426, 1431, 1674, 1820, 1842
scale_image: 1552
scaled: 101, 102, 103, 104, 105, 106, 107, 108, 110, 126, 128, 131, 165, 168, 174, 194, 195, 473, 474, 476, 479, 574, 575, 586, 597, 611, 619, 634, 643, 647, 657, 673, 687, 689, 690, 691, 692, 693, 705, 706, 708, 712, 722, 725, 729, 738, 818, 823, 844, 855, 880, 881, 882, 888, 891, 892, 893, 895, 902, 911, 912, 913, 914, 919, 925, 932, 938, 967, 976, 999, 1005, 1006, 1015, 1023, 1053, 1083, 1147, 1148, 1154, 1157, 1159, 1171, 1189, 1246, 1264, 1301, 1316, 1376, 1435, 1552, 1628, 1630, 1635, 1636, 1637, 1719, 1723, 1744, 1842, 1844
scaled: 1436
scaled_base: 265, 267, 269, 1402, 1415
scaled_out: 687, 689, 690, 692, 693
scan_action: 1556, 1560, 1563, 1579
scan_alt_rule: 1552, 1556, 1565
scan_box: 1251, 1262, 1419
scan_char_num: 440, 452, 460, 1112, 1207, 1215, 1301, 1302, 1329, 1332, 1402, 1410, 1431, 1671, 1769
scan_delimiter: 1338, 1341, 1360, 1361, 1369, 1370
scan_dimen: 436, 466, 473, 474, 487, 488, 1239
scan_eight_bit_int: 459, 1277
scan_expr: 1780, 1781, 1782
scan_fifteen_bit_int: 462, 1329, 1332, 1343, 1402

scan_file_name: 287, 356, 552, 553, 563, 1435, 1453, 1531
scan_font_ident: 441, 452, 497, 604, 605, 705, 706, 1412, 1431, 1587, 1589, 1593, 1671, 1769
scan_four_bit_int: 461, 527, 604, 1412, 1453, 1530
scan_general_text: 1682, 1683, 1688, 1753
scan_glue: 436, 487, 958, 1238, 1406, 1416, 1573, 1787
scan_image: 1552, 1553
scan_int: 435, 436, 458, 459, 460, 461, 462, 463, 464, 466, 473, 474, 487, 497, 529, 530, 535, 605, 705, 706, 1281, 1403, 1406, 1410, 1416, 1418, 1421, 1422, 1424, 1426, 1431, 1436, 1530, 1539, 1544, 1546, 1549, 1552, 1554, 1556, 1558, 1563, 1565, 1566, 1575, 1585, 1625, 1674, 1769, 1785, 1811, 1866
scan_keyword: 180, 433, 479, 480, 481, 482, 484, 488, 489, 497, 705, 706, 817, 1260, 1277, 1403, 1414, 1436, 1534, 1538, 1539, 1544, 1548, 1552, 1556, 1558, 1563, 1565, 1566, 1579
scan_left_brace: 429, 499, 817, 961, 1111, 1137, 1202, 1277, 1295, 1297, 1331, 1350, 1352, 1683
scan_math: 1328, 1329, 1336, 1341, 1343, 1354
scan_mu_glue: 1785, 1786, 1787, 1807
scan_normal_dimen: 474, 489, 529, 817, 1251, 1260, 1360, 1361, 1406, 1416, 1421, 1423, 1425, 1426, 1431, 1437, 1552, 1769, 1785
scan_normal_glue: 1785, 1786, 1787, 1803, 1804, 1808
scan_optional_equals: 431, 705, 706, 958, 1402, 1404, 1406, 1410, 1412, 1414, 1419, 1421, 1422, 1423, 1424, 1425, 1426, 1431, 1435, 1453, 1531
scan_pdf_box_spec: 1552
scan_pdf_ext_late_toks: 1537, 1538
scan_pdf_ext_toks: 497, 703, 1537, 1538, 1539, 1540, 1544, 1548, 1552, 1556, 1558, 1563, 1565, 1566, 1578, 1579, 1580, 1581, 1582, 1587, 1589, 1590, 1591, 1599
scan_register_num: 412, 441, 446, 453, 497, 531, 1257, 1260, 1279, 1288, 1402, 1404, 1405, 1415, 1419, 1425, 1474, 1548, 1810, 1811
scan_rule_spec: 489, 1234, 1262
scan_something_internal: 435, 436, 439, 458, 466, 475, 477, 481, 487, 491, 1780
scan_spec: 817, 944, 950, 1249, 1261, 1345
scan_special: 693, 695, 726, 727
scan_thread_id: 1566, 1567, 1568
scan_tokens: 1747
\scantokens primitive: 1747
scan_toks: 313, 490, 499, 1137, 1279, 1396, 1404, 1457, 1466, 1532, 1534, 1537, 1618, 1682
scan_twenty_seven_bit_int: 463, 1329, 1332, 1338
scanned_result: 439, 440, 441, 444, 448, 451, 452, 454
scanned_result_end: 439
scanner_status: 327, 328, 353, 358, 361, 388, 393, 394, 415, 417, 496, 497, 499, 508, 520, 524, 527, 533, 953, 965, 1223, 1683, 1766
\scriptfont primitive: 1408
script_mlist: 865, 871, 874, 907, 1352
\scriptscriptfont primitive: 1408
script_script_mlist: 865, 871, 874, 907, 1352
script_script_size: 875, 932, 1373, 1408
script_script_style: 864, 870, 907, 1347
\scriptscriptstyle primitive: 1347
script_size: 875, 932, 1373, 1408
script_space: 265, 933, 934, 935
\scriptspace primitive: 266
script_space_code: 265, 266
script_style: 864, 870, 878, 879, 907, 932, 942, 1347
\scriptstyle primitive: 1347
scripts_allowed: 863, 1354
scroll_mode: 71, 73, 84, 86, 93, 556, 1440, 1441, 1459
\scrollmode primitive: 1440
search_mem: 183, 190, 273, 1519
second_indent: 1023, 1024, 1025, 1066
second_pass: 1004, 1039, 1042
second_width: 1023, 1024, 1025, 1026, 1066
seconds_and_micros: 1555, 1584, 1586
Sedgewick, Robert: 2
see the transcript file...: 1515
seed: 125
selector: 54, 55, 57, 58, 59, 62, 71, 75, 86, 90, 92, 98, 263, 333, 334, 338, 382, 491, 496, 560, 561, 645, 666, 686, 705, 706, 712, 727, 1435, 1443, 1457, 1476, 1508, 1513, 1515, 1615, 1617, 1688, 1753
semi_simple_group: 291, 1241, 1243, 1246, 1247, 1661, 1679
serial: 997, 1021, 1022, 1032
set_aux: 227, 439, 442, 443, 444, 1388, 1420
set_box: 227, 287, 288, 1388, 1419
\setbox primitive: 287
set_box_allowed: 76, 77, 1419, 1448
set_box_dimen: 227, 439, 442, 443, 1388, 1420
set_box_lr: 643, 983, 984, 1372, 1380, 1714, 1721
set_box_lr_end: 643
set_break_width_to_background: 1013
set_char_and_font: 705
set_char_0: 612, 613, 648, 719, 726
set_conversion: 484
set_conversion_end: 484
set_cur_lang: 1111, 1137, 1269, 1378

set_cur_r: 1085, 1087, 1088
set_ef_code: 705, 1431
set_expand_params: 705, 712
set_ff: 498, 693, 696, 698, 766
set_font: 227, 439, 579, 604, 706, 1388, 1395, 1435, 1439
set_glue_ratio_one: 109, 840, 852, 986, 987
set_glue_ratio_zero: 109, 154, 833, 834, 840, 848, 849, 852, 986, 987
set_height_zero: 1147
set_hyph_index: 1068, 1111, 1610, 1858
set_image_group_ref: 1637
set_interaction: 227, 1388, 1440, 1441, 1442
set_job_id: 792
set_kn_ac_code: 705, 1431
set_kn_bc_code: 705, 1431
set_kn_bs_code: 705, 1431
\setlanguage primitive: 1524
set_language_code: 1524, 1526, 1528
set_lc_code: 1073, 1074, 1075, 1114, 1858
set_lp_code: 705, 1431
set_math_char: 1332, 1333
set_no_ligatures: 705, 706, 1431
set_obj_fresh: 695, 698, 812
set_obj_scheduled: 695, 1630, 1635
set_origin: 497, 693, 695, 727, 1538, 1603
set_page_dimen: 227, 439, 1159, 1160, 1161, 1388, 1420
set_page_int: 227, 439, 442, 443, 1388, 1420, 1692
set_page_so_far_zero: 1164
set_prev_graf: 227, 287, 288, 439, 1388, 1420
set_random_seed_code: 1524, 1526, 1528
\pdfsetrandomseed primitive: 1524
set_rect_dimens: 1630, 1635, 1636, 1637
set_rp_code: 705, 1431
set_rule: 610, 612, 613, 652, 719, 726
set_sa_box: 1821
set_sh_bs_code: 705, 1431
set_shape: 227, 251, 287, 288, 439, 1388, 1426, 1864
set_st_bs_code: 705, 1431
set_tag_code: 705, 1431
set_trick_count: 338, 339, 340, 342
set1: 612, 613, 648, 706, 719, 726
set2: 612
set3: 612
set4: 612
sf_code: 248, 250, 1211
\sfcode primitive: 1408
sf_code_base: 248, 253, 1408, 1409, 1411
sh: 705
\shbscode primitive: 1432
sh_bs_code_base: 173, 452, 1431, 1432, 1433

shape_ref: 228, 250, 297, 1248, 1426
shellenabledp: 450
shift_amount: 153, 154, 177, 202, 651, 656, 660, 665, 733, 737, 742, 746, 823, 827, 844, 846, 857, 882, 896, 913, 914, 925, 926, 932, 933, 935, 975, 982, 983, 984, 1066, 1254, 1259, 1303, 1734, 1740, 1744, 1745, 1746
shift_case: 1463, 1466
shift_down: 919, 920, 921, 922, 923, 925, 927, 932, 933, 935
shift_up: 919, 920, 921, 922, 923, 925, 927, 932, 934, 935
ship_out: 619, 727, 750, 761, 791, 816, 983, 984, 1200, 1253, 1648, 1700, 1705
\shipout primitive: 1249
ship_out_flag: 1249, 1253, 1681
shipping_page: 750, 751, 752, 757, 759, 760, 763, 1640
short_display: 191, 192, 193, 211, 674, 839, 1033, 1519
short_display_n: 674, 823, 1005, 1027, 1057
short_real: 109, 128
shortcut: 473, 474
shortfall: 1006, 1027, 1028, 1029, 1842, 1847, 1849, 1850
shorthand_def: 227, 1388, 1400, 1401, 1402
\show primitive: 1469
show_activities: 236, 1471
show_box: 198, 200, 216, 236, 237, 254, 666, 669, 750, 839, 851, 1163, 1169, 1299, 1474, 1519
\showbox primitive: 1469
show_box_breadth: 254, 1519
\showboxbreadth primitive: 256
show_box_breadth_code: 254, 255, 256
show_box_code: 1469, 1470, 1471
show_box_depth: 254, 1519
\showboxdepth primitive: 256
show_box_depth_code: 254, 255, 256
show_code: 1469, 1471
show_context: 54, 78, 82, 88, 332, 333, 340, 556, 561, 563, 1564, 1774, 1776, 1777
show_cur_cmd_chr: 321, 391, 520, 524, 536, 1208, 1389
show_eqtb: 270, 306, 1823
show_groups: 1675, 1676, 1677
\showgroups primitive: 1675
show_ifs: 1689, 1690, 1691
\showifs primitive: 1689
show_info: 868, 869
show_lists_code: 1469, 1470, 1471
\showlists primitive: 1469

show_node_list: 191, 194, 198, 199, 200, 213, 216,
 251, 866, 868, 869, 871, 1005, 1145, 1519, 1823
show_sa: 1823, 1839, 1840, 1841
show_save_groups: 1515, 1677, 1679
\showthe primitive: 1469
show_the_code: 1469, 1470
show_token_list: 194, 241, 251, 286, 314, 317, 328,
 341, 342, 426, 686, 727, 1519, 1615, 1823
show_tokens: 1684, 1685, 1686
\showtokens primitive: 1684
show_whatever: 1468, 1471
shown_mode: 231, 233, 321
shrink: 168, 169, 182, 196, 457, 488, 653, 662,
 705, 735, 744, 832, 847, 892, 985, 1001, 1003,
 1014, 1044, 1153, 1181, 1186, 1220, 1222, 1326,
 1407, 1417, 1418, 1637, 1638, 1699, 1746, 1790,
 1791, 1794, 1795, 1796, 1798, 1804
shrink_amount: 1637
shrink_limit: 705
shrink_order: 168, 182, 196, 488, 653, 662, 735,
 744, 832, 847, 892, 985, 1001, 1002, 1153,
 1181, 1186, 1326, 1417, 1637, 1638, 1699,
 1746, 1791, 1794, 1803
shrinking: 153, 204, 647, 657, 729, 738, 840, 852,
 985, 986, 987, 1326, 1637, 1699, 1723
si: 38, 42, 69, 1128, 1141, 1488, 1754, 1856
side: 823
sign: 689
simple_group: 291, 1241, 1246, 1661, 1679
Single-character primitives: 289
 \:-: 1292
 \/: 287
 \u: 287
single_base: 240, 284, 285, 286, 376, 394, 395, 400,
 468, 527, 706, 1223, 1435, 1467, 1768
skew_char: 173, 452, 575, 578, 603, 705, 706,
 917, 1431, 1500, 1501
\skewchar primitive: 1432
skip: 242, 453, 1186
\skip primitive: 437
skip_base: 242, 245, 247, 1402, 1415
skip_blanks: 325, 366, 367, 369, 371, 376
skip_byte: 571, 583, 823, 917, 928, 929, 1086, 1216
skip_code: 1236, 1237, 1238
\skipdef primitive: 1400
skip_def_code: 1400, 1401, 1402
skip_line: 358, 519, 520
skipping: 327, 328, 358, 520
slant: 573, 584, 602, 673, 1301, 1303
slant_code: 573, 584
slow_print: 60, 61, 63, 84, 544, 562, 563, 608, 670,
 1439, 1458, 1461, 1508, 1513, 1519
slow_print_substr: 693
small_char: 859, 867, 873, 882, 1338
small_fam: 859, 867, 873, 882, 1338
small_node_size: 159, 162, 163, 165, 170, 171, 174,
 176, 220, 224, 831, 897, 1080, 1087, 1091, 1214,
 1278, 1279, 1561, 1569, 1572, 1575, 1576, 1594,
 1595, 1596, 1597, 1598, 1604, 1605, 1624, 1625,
 1722, 1725, 1727, 1729, 1733, 1738
small_number: 101, 102, 165, 170, 172, 286, 388,
 415, 439, 464, 466, 476, 487, 491, 496, 508,
 515, 520, 523, 524, 549, 604, 634, 706, 729,
 772, 823, 844, 864, 882, 895, 896, 902, 932,
 938, 1005, 1069, 1070, 1082, 1083, 1098, 1111,
 1121, 1137, 1147, 1164, 1238, 1253, 1264, 1269,
 1354, 1359, 1369, 1376, 1389, 1414, 1424, 1425,
 1435, 1471, 1503, 1515, 1529, 1530, 1556, 1564,
 1573, 1617, 1620, 1633, 1636, 1705, 1719, 1782,
 1815, 1819, 1821, 1823, 1825, 1842
snap_glue_ptr: 695, 1573, 1603, 1604, 1605, 1637
snap_node_size: 695, 1573, 1604, 1605
snap_unit: 1637
snappy_comp_ratio: 695, 1575, 1603, 1637
so: 38, 45, 59, 60, 69, 70, 286, 433, 490, 545, 630,
 645, 693, 942, 1108, 1130, 1132, 1133, 1136,
 1140, 1487, 1615, 1754, 1855
Sorry, I can't find...: 550
sort_avail: 149, 1489
sort_dest_names: 793, 804
sp: 104, 614
sp: 484
sp: 1637
space: 573, 584, 693, 928, 931, 1220
space_code: 573, 584, 605, 1220
space_factor: 230, 231, 444, 962, 963, 975, 1207,
 1211, 1221, 1222, 1234, 1254, 1261, 1269, 1271,
 1295, 1297, 1301, 1374, 1378, 1420, 1421
\spacefactor primitive: 442
space_shrink: 573, 584, 693, 1220
space_shrink_code: 573, 584, 605
space_skip: 242, 1219, 1221
\spaceskip primitive: 244
space_skip_code: 242, 243, 244, 1219
space_stretch: 573, 584, 1220
space_stretch_code: 573, 584
space_token: 311, 419, 490, 1393, 1761
spacer: 225, 226, 250, 311, 313, 316, 320, 325,
 359, 367, 369, 370, 371, 376, 430, 432, 433,
 469, 470, 478, 490, 959, 961, 967, 1112, 1138,
 1207, 1223, 1399
\span primitive: 956
span_code: 956, 957, 958, 965, 967
span_count: 177, 203, 972, 977, 984

span_node_size: 973, 974, 979
spec: 1550, 1552, 1556, 1565
spec_code: 817
spec_log: 117, 118, 120
\special primitive: 1524
special_node: 1521, 1524, 1526, 1528, 1534, 1603, 1604, 1605, 1620, 1639, 1645
special_out: 1615, 1620
split: 1188
split_bot_mark: 408, 409, 1154, 1156, 1809, 1826, 1827
\splitbotmark primitive: 410
split_bot_mark_code: 408, 410, 411, 1515, 1809, 1831
\splitbotmarks primitive: 1809
split_disc: 1145, 1154, 1859, 1860
\splitdiscards primitive: 1861
split_first_mark: 408, 409, 1154, 1156, 1809, 1827
\splitfirstmark primitive: 410
split_first_mark_code: 408, 410, 411, 1809
\splitfirstmarks primitive: 1809
split_fist_mark: 1826
split_max_depth: 158, 265, 1154, 1246, 1278
\splitmaxdepth primitive: 266
split_max_depth_code: 265, 266
split_top_ptr: 158, 206, 220, 224, 1198, 1199, 1278
split_top_skip: 158, 242, 1145, 1154, 1189, 1191, 1198, 1278
\splittopskip primitive: 244
split_top_skip_code: 242, 243, 244, 1146
split_up: 1158, 1163, 1185, 1187, 1197, 1198
spotless: 76, 77, 263, 686, 1512, 1515, 1774, 1776, 1777
spread: 817
sprint_cs: 241, 285, 360, 421, 422, 424, 498, 505, 510, 587, 1472
sq: 597
square roots: 913
ss_code: 1236, 1237, 1238
ss_glue: 180, 182, 891, 1238
st: 705
\stbscode primitive: 1432
st_bs_code_base: 173, 452, 1431, 1432, 1433
stack conventions: 322
stack_h: 722, 726
stack_into_box: 887, 889
stack_level: 712, 717, 719
stack_no: 727
stack_size: 11, 323, 332, 343, 1514
stack_v: 722, 726
stack_w: 722, 726
stack_x: 722, 726

stack_y: 722, 726
stack_z: 722, 726
start: 322, 324, 325, 329, 340, 341, 345, 346, 347, 348, 350, 351, 353, 382, 384, 385, 393, 394, 509, 564, 1755
start_cs: 363, 376, 377
start_eq_no: 1318, 1320
start_field: 322, 324
start_font_error_message: 587, 593
start_here: 5, 1512
start_input: 388, 402, 404, 563, 1517
start_of_TEX: 6, 1512
start_packet: 725
start_par: 226, 1266, 1267, 1268, 1270
start_status: 727
stat: 7, 135, 138, 139, 140, 141, 143, 148, 270, 279, 296, 299, 304, 305, 306, 667, 1002, 1005, 1021, 1031, 1039, 1164, 1182, 1187, 1513, 1662, 1823, 1839, 1840, 1841
state: 87, 322, 324, 325, 329, 333, 334, 345, 347, 350, 352, 353, 359, 363, 365, 366, 368, 369, 371, 374, 375, 376, 416, 509, 563, 1515
state_field: 322, 324, 1309, 1775
step: 705
stomach: 428
stop: 225, 1223, 1224, 1230, 1231, 1232, 1272
stop_flag: 571, 583, 823, 917, 928, 929, 1086, 1216
store_background: 1040
store_break_width: 1019
store_fmt_file: 1480, 1515
store_four_quarters: 590, 594, 595, 600, 601
store_new_token: 397, 398, 419, 423, 425, 433, 490, 492, 499, 500, 502, 503, 508, 509, 1683, 1761, 1767
store_scaled: 597, 598, 600, 602
store_scaled_f: 597, 712, 717, 725
storepacket: 706, 718
str_eq_buf: 45, 278
str_eq_str: 46, 281, 693, 705, 801, 1438
str_in_str: 686, 693
str_less_str: 793
str_number: 38, 39, 43, 45, 46, 47, 62, 63, 79, 93, 94, 95, 195, 196, 281, 286, 306, 433, 496, 538, 545, 551, 553, 555, 556, 558, 575, 586, 686, 693, 698, 702, 704, 705, 706, 708, 712, 725, 727, 749, 772, 793, 807, 811, 1103, 1106, 1111, 1435, 1457, 1477, 1537, 1552, 1555, 1564, 1587, 1602, 1627, 1628, 1630, 1679, 1753, 1814, 1823
str_pool: 38, 39, 42, 43, 45, 46, 47, 59, 60, 69, 70, 274, 279, 283, 286, 325, 433, 490, 545, 629, 630, 645, 666, 686, 693, 702, 706, 727, 749, 750, 793,

940, 942, 1106, 1108, 1111, 1118, 1487, 1488,
 1514, 1537, 1587, 1615, 1630, 1753, 1754
str_ptr: 38, 39, 41, 43, 44, 47, 59, 60, 70, 279,
 281, 284, 496, 543, 551, 563, 645, 673, 693,
 727, 1438, 1487, 1488, 1501, 1503, 1507,
 1512, 1514, 1615
str_room: 42, 198, 279, 490, 542, 551, 706, 712,
 717, 726, 727, 1116, 1435, 1457, 1508, 1513,
 1615, 1753
str_start: 38, 39, 40, 41, 43, 44, 45, 46, 47, 59, 60,
 69, 70, 274, 279, 281, 286, 433, 496, 497, 543,
 545, 630, 645, 686, 693, 702, 706, 727, 749,
 793, 941, 1106, 1108, 1111, 1118, 1487, 1488,
 1537, 1587, 1615, 1630, 1754
str Toks: 490, 491, 496, 497, 1688
stretch: 168, 169, 182, 196, 457, 488, 653, 662,
 705, 735, 744, 832, 847, 892, 985, 1003, 1014,
 1044, 1153, 1181, 1186, 1220, 1222, 1326, 1407,
 1417, 1418, 1637, 1638, 1699, 1746, 1790, 1791,
 1794, 1795, 1796, 1798, 1804, 1843, 1853
stretch_amount: 1637
stretch_limit: 705
stretch_order: 168, 182, 196, 488, 653, 662, 735,
 744, 832, 847, 892, 985, 1003, 1014, 1044, 1153,
 1181, 1186, 1326, 1417, 1637, 1638, 1699, 1746,
 1791, 1794, 1803, 1843
stretching: 153, 653, 662, 735, 744, 834, 849, 985,
 986, 987, 1326, 1638, 1699
string pool: 47, 1486
\string primitive: 494
string_code: 494, 495, 497, 498
string_vacancies: 11, 52
style: 902, 903, 936, 937, 938
style_node: 178, 864, 866, 874, 906, 907, 937,
 1347, 1719
style_node_size: 864, 865, 874, 939, 1719
sub_box: 857, 863, 868, 874, 896, 910, 911, 913,
 914, 925, 930, 1254, 1271, 1346
sub_char_shrink: 1015, 1017
sub_char_shrink_end: 1015
sub_char_stretch: 1015, 1017
sub_char_stretch_end: 1015
sub_disc_width_from_active_width: 1015, 1045
sub_drop: 876, 932
sub_kern_shrink: 1015, 1017
sub_kern_shrink_end: 1015
sub_kern_stretch: 1015, 1017
sub_kern_stretch_end: 1015
sub_mark: 225, 316, 320, 369, 1224, 1353
sub_mlist: 857, 859, 868, 896, 918, 930, 1359,
 1363, 1364, 1369
sub_style: 878, 926, 933, 935
sub_{sup}: 1353, 1354
subscr: 857, 859, 862, 863, 866, 872, 874, 914, 918,
 925, 926, 927, 928, 929, 930, 931, 932, 933, 935,
 1329, 1341, 1343, 1353, 1354, 1355, 1364
subscripts: 930, 1353
subst_ex_font: 823, 825, 828
subst_font_type: 703
substituted: 823
substr_of_str: 749, 769, 807
subtype: 151, 152, 153, 154, 157, 158, 160, 161,
 162, 163, 164, 165, 167, 168, 170, 171, 172, 173,
 174, 176, 177, 193, 201, 206, 207, 208, 209, 210,
 211, 450, 498, 515, 521, 522, 643, 653, 655, 662,
 664, 674, 695, 705, 727, 735, 744, 772, 823, 825,
 832, 844, 847, 857, 858, 862, 863, 864, 865, 866,
 872, 893, 906, 907, 908, 909, 925, 939, 942,
 944, 962, 969, 971, 985, 995, 996, 998, 1005,
 1013, 1017, 1018, 1019, 1020, 1042, 1044, 1046,
 1047, 1055, 1057, 1073, 1074, 1075, 1076, 1080,
 1087, 1145, 1158, 1163, 1165, 1177, 1185, 1186,
 1195, 1197, 1198, 1212, 1238, 1239, 1256, 1258,
 1278, 1279, 1291, 1303, 1326, 1337, 1341, 1343,
 1349, 1359, 1369, 1515, 1521, 1529, 1573, 1603,
 1604, 1605, 1606, 1607, 1608, 1609, 1610, 1611,
 1612, 1615, 1620, 1622, 1635, 1637, 1639, 1645,
 1691, 1699, 1711, 1717, 1719, 1720, 1728, 1736,
 1737, 1739, 1777, 1788, 1789, 1815
sub1: 876, 933
sub2: 876, 935
succumb: 93, 94, 95, 686, 1482
sup_dest_names_size: 695, 698, 1513
sup_drop: 876, 932
sup_mark: 225, 316, 320, 366, 377, 1224, 1353,
 1354, 1355
sup_obj_tab_size: 695, 698, 1513
sup_pdf_mem_size: 675, 678, 1513
sup_pdf_os_buf_size: 679, 686
sup_pk_dpi: 695
sup_style: 878, 926, 934
superscripts: 930, 1353
supscr: 857, 859, 862, 863, 866, 872, 874, 914,
 918, 925, 927, 928, 929, 930, 932, 934, 1329,
 1341, 1343, 1353, 1354, 1355, 1364
sup1: 876, 934
sup2: 876, 934
sup3: 876, 934
sw: 586, 597, 602
switch: 363, 365, 366, 368, 372
synch_h: 643, 648, 652, 656, 661, 665, 1615
synch_v: 643, 648, 652, 656, 660, 661, 665, 1615
sys_day: 259, 264, 562
sys_month: 259, 264, 562

sys_obj_ptr: 696, 697, 698, 802, 812, 814, 815, 1504, 1505
sys_time: 259, 264, 562
sys_year: 259, 264, 562
 system dependencies: 2, 3, 4, 9, 10, 11, 12, 19, 21, 23, 26, 27, 28, 32, 33, 34, 35, 37, 38, 49, 56, 59, 61, 72, 81, 84, 96, 109, 112, 128, 130, 131, 179, 204, 259, 326, 335, 350, 511, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 549, 551, 563, 564, 583, 590, 618, 622, 624, 974, 1484, 1511, 1512, 1513, 1518, 1520, 1831, 1867
sz: 1753, 1754, 1756
s1: 82, 88, 793, 1537, 1564, 1587
s2: 82, 88, 793, 1537, 1564, 1587
s3: 82, 88
s4: 82, 88
t: 46, 107, 108, 143, 236, 299, 301, 302, 303, 345, 363, 388, 415, 490, 496, 499, 689, 706, 880, 881, 902, 932, 976, 1005, 1006, 1053, 1083, 1111, 1143, 1147, 1207, 1301, 1354, 1369, 1376, 1435, 1466, 1471, 1723, 1733, 1738, 1744, 1782, 1799, 1819, 1823
t_open_in: 33, 37
t_open_out: 33, 1512
tab_mark: 225, 311, 316, 364, 369, 956, 957, 958, 959, 960, 964, 1304
tab_skip: 242
\tabskip primitive: 244
tab_skip_code: 242, 243, 244, 954, 958, 962, 969, 971, 985
tab_token: 311, 1306
tag: 569, 570, 580
tag_code: 173, 452, 1431, 1432, 1433
\tagcode primitive: 1432
tail: 160, 230, 231, 232, 233, 234, 450, 855, 894, 952, 962, 971, 972, 975, 988, 992, 1067, 1172, 1194, 1200, 1203, 1211, 1212, 1213, 1214, 1217, 1218, 1219, 1221, 1232, 1238, 1239, 1254, 1256, 1258, 1269, 1274, 1278, 1279, 1283, 1288, 1291, 1295, 1297, 1298, 1301, 1303, 1323, 1328, 1333, 1336, 1337, 1341, 1343, 1346, 1349, 1352, 1354, 1355, 1359, 1362, 1364, 1365, 1369, 1374, 1383, 1384, 1529, 1530, 1531, 1532, 1533, 1534, 1538, 1539, 1540, 1546, 1549, 1554, 1556, 1558, 1560, 1565, 1566, 1575, 1623, 1624, 1625, 1863
tail_append: 231, 232, 962, 971, 992, 1212, 1214, 1217, 1218, 1232, 1234, 1238, 1239, 1269, 1271, 1278, 1281, 1290, 1291, 1295, 1328, 1336, 1341, 1343, 1346, 1349, 1350, 1355, 1369, 1374, 1381, 1383, 1384, 1574, 1703
tail_field: 230, 231, 1172
tail_page_disc: 1176, 1859

take_frac: 114, 126, 127
take_fraction: 1799
tally: 54, 55, 57, 58, 314, 334, 337, 338, 339
tats: 7
temp_head: 180, 328, 417, 422, 426, 490, 492, 493, 496, 497, 504, 895, 896, 930, 936, 992, 1038, 1039, 1040, 1053, 1055, 1056, 1057, 1063, 1145, 1242, 1243, 1372, 1374, 1377, 1475, 1683, 1688, 1707, 1709, 1734, 1735, 1737, 1738, 1753
temp_ptr: 133, 172, 646, 647, 651, 656, 657, 660, 665, 668, 729, 733, 737, 738, 742, 746, 752, 855, 868, 869, 1146, 1178, 1198, 1214, 1219, 1515, 1705, 1707, 1709, 1712, 1721, 1722, 1723, 1727, 1746
ten_pow: 687, 688, 689, 690, 792, 1637
term_and_log: 54, 57, 58, 71, 75, 92, 263, 560, 1476, 1508, 1515, 1617
term_in: 32, 33, 34, 36, 37, 71, 1518, 1519
term_input: 71, 78
term_offset: 54, 55, 57, 58, 61, 62, 71, 563, 666, 750, 1458, 1755
term_only: 54, 55, 57, 58, 71, 75, 92, 561, 1476, 1513, 1515
term_out: 32, 33, 34, 35, 36, 37, 51, 56
terminal_input: 326, 335, 350, 352, 382
test_char: 1083, 1086
test_no_ligatures: 452, 604
TEX: 2, 4
 TeX capacity exceeded ...: 94
 buffer size: 35, 286, 350, 400, 1768
 exception dictionary: 1117
 font memory: 607
 grouping levels: 296
 hash size: 279
 input stack size: 343
 main memory size: 138, 143
 number of strings: 43, 543
 parameter stack size: 416
 pattern memory: 1131, 1141
 pool size: 42
 primitive size: 282
 save size: 295
 semantic nest size: 234
 text input levels: 350
TEX.POOL check sum...: 53
TEX.POOL doesn't match: 53
TEX.POOL has no check sum: 52
TEX.POOL line doesn't...: 52
TEX_area: 540, 563
tex_b_openin: 772
TeX_banner: 2
TEX_font_area: 540, 589

TEX_format_default: 546, 547, 549
tex_int_pars: 254
tex_toks: 248
The *TeXbook*: 1, 23, 49, 108, 225, 441, 472, 482, 485, 859, 864, 940, 1393, 1511
TeXfonts: 540
TeXformats: 11, 547
TeXinputs: 540
texput: 35, 560, 1435
text: 274, 276, 277, 278, 279, 284, 285, 286, 287, 394, 395, 517, 527, 579, 706, 956, 1223, 1366, 1394, 1435, 1496, 1616
Text line contains...: 368
text_char: 19, 20, 25, 47
\textfont primitive: 1408
text_mlist: 865, 871, 874, 907, 1352
text_size: 875, 879, 908, 1373, 1377
text_style: 864, 870, 879, 907, 913, 920, 921, 922, 924, 925, 934, 1347, 1372, 1374
\textstyle primitive: 1347
TeXeT: 1700
TeXeT_code: 2, 1700, 1701
TeXeT_en: 823, 825, 1055, 1056, 1057, 1700, 1703, 1734, 1735, 1736
TeXeT_state: 1700
\TeXeT_state primitive: 1701
TEX82: 1, 99
TFM files: 565
tfm_file: 565, 586, 589, 590, 602
tfm_lookup: 705, 706, 712
tfm_width: 712, 717
TFtoPL: 587
That makes 100 errors...: 82
the: 228, 287, 288, 388, 391, 504, 1686
The following...deleted: 669, 1169, 1299
\the primitive: 287
the_toks: 491, 492, 493, 504, 1475, 1688
thick_mu_skip: 242
\thickmuskip primitive: 244
thick_mu_skip_code: 242, 243, 244, 942
thickness: 859, 873, 901, 919, 920, 922, 923, 1360
thin_mu_skip: 242
\thinmuskip primitive: 244
thin_mu_skip_code: 242, 243, 244, 247, 942
This can't happen: 95
/: 112
align: 976
copying: 224
curlevel: 303
disc1: 1017
disc2: 1018
disc3: 1046

disc4: 1047
display: 1378
endv: 967
ext1: 1528
ext2: 1604
ext3: 1605
ext4: 1620
flushing: 220
if: 523
line breaking: 1053
LR1: 1712
LR2: 1725
LR3: 1730
mlist1: 904
mlist2: 930
mlist3: 937
mlist4: 942
page: 1177
paragraph: 1042
pdfvlistout: 740
prefix: 1389
pruning: 1145
right: 1363
rightbrace: 1246
tail1: 1258
vcenter: 912
vertbreak: 1150
vlistout: 658
vpack: 845
256 spans: 974
this_box: 647, 652, 653, 657, 661, 662, 729, 730, 734, 735, 738, 739, 743, 744, 1637, 1638, 1639, 1645, 1714, 1715, 1721, 1722, 1723
this_if: 524, 527, 529, 531, 532, 1769
thread: 1600
thread_title: 1600
threads: 790, 806, 1513
three_codes: 817
threshold: 1004, 1027, 1030, 1039
Tight \hbox...: 843
Tight \vbox...: 854
tight_fit: 993, 995, 1006, 1009, 1010, 1012, 1029, 1842, 1848
time: 254, 259, 645, 792
\time primitive: 256
time_code: 254, 255, 256
tini: 8
tmp_b0: 706, 710, 712, 714
tmp_b1: 706, 710, 712, 714
tmp_b2: 706, 710, 714
tmp_b3: 706, 710, 714
tmp_int: 710, 726

tmp_k1: 1207, 1211, 1216, 1217
tmp_k2: 1207, 1212
tmp_v: 1637
tmp_w: 705, 710
to: 817, 1260, 1403
tok_val: 436, 441, 444, 454, 491, 1402, 1404, 1405, 1489, 1490, 1815, 1818, 1823
tok_val_limit: 1815, 1837
token: 311
token_list: 329, 333, 334, 345, 347, 352, 359, 363, 368, 416, 1309, 1515, 1775
token_ref_count: 218, 221, 313, 499, 508, 1156, 1683
token_show: 317, 318, 345, 427, 1457, 1462, 1475, 1617, 1688, 1753
token_type: 329, 333, 334, 336, 341, 345, 346, 347, 349, 405, 416, 1203, 1273
tokens_to_string: 497, 686, 708, 727, 769, 772, 792, 807, 1537, 1552, 1555, 1563, 1565, 1587, 1589, 1599, 1630, 1637
toks: 248
\toks primitive: 287
toks_base: 248, 249, 250, 251, 329, 441, 1402, 1404, 1405
\toksdef primitive: 1400
toks_def_code: 1400, 1402
toks_register: 227, 287, 288, 439, 441, 1388, 1399, 1402, 1404, 1405, 1823, 1833, 1834
tolerance: 254, 258, 1004, 1039
\tolerance primitive: 256
tolerance_code: 254, 255, 256
Too many }'s: 1246
Too many color stacks: 497
too_big: 1799
too_small: 1481, 1484
top: 572, 693
top_bot_mark: 228, 318, 388, 391, 410, 411, 412, 1809
top_edge: 657, 664, 738, 739, 1639
top_mark: 408, 409, 1189, 1809, 1828
\topmark primitive: 410
top_mark_code: 408, 410, 412, 1515, 1809, 1831
\topmarks primitive: 1809
top_skip: 242
\topskip primitive: 244
top_skip_code: 242, 243, 244, 1178
total_demerits: 995, 1021, 1022, 1031, 1040, 1050, 1051
total_font_shrink: 999, 1027
total_font_stretch: 999, 1027
total height: 1163
total_mathex_params: 877, 1373

total_mathsy_params: 876, 1373
total_pages: 619, 620, 645, 668, 670, 751, 752, 770, 794
total_pw: 1005, 1027
total_shrink: 818, 824, 832, 840, 841, 842, 843, 847, 852, 853, 854, 972, 1379
total_stretch: 818, 824, 832, 834, 835, 836, 847, 849, 850, 972
Trabb Pardo, Luis Isidoro: 2
tracing_assigns: 254, 299, 1839, 1840
\tracingassigns primitive: 1657
tracing_assigns_code: 254, 1657, 1659
tracing_commands: 254, 391, 524, 535, 536, 1208, 1389
\tracingcommands primitive: 256
tracing_commands_code: 254, 255, 256
tracing_groups: 254, 296, 304
\tracinggroups primitive: 1657
tracing_groups_code: 254, 1657, 1659
tracing_ifs: 254, 321, 520, 524, 536
\tracingifs primitive: 1657
tracing_ifs_code: 254, 1657, 1659
tracing_lost_chars: 254, 608
\tracinglostchars primitive: 256
tracing_lost_chars_code: 254, 255, 256
tracing_macros: 254, 345, 415, 426
\tracingmacros primitive: 256
tracing_macros_code: 254, 255, 256
tracing_nesting: 254, 384, 1774, 1775, 1776, 1777
\tracingnesting primitive: 1657
tracing_nesting_code: 254, 1657, 1659
tracing_online: 254, 263, 608, 1471, 1476
\tracingonline primitive: 256
tracing_online_code: 254, 255, 256
tracing_output: 254, 666, 669, 750
\tracingoutput primitive: 256
tracing_output_code: 254, 255, 256
tracing_pages: 254, 1164, 1182, 1187
\tracingpages primitive: 256
tracing_pages_code: 254, 255, 256
tracing_paragraphs: 254, 1002, 1021, 1031, 1039
\tracingparagraphs primitive: 256
tracing_paragraphs_code: 254, 255, 256
tracing_restores: 254, 305, 1841
\tracingrestores primitive: 256
tracing_restores_code: 254, 255, 256
tracing_scan_tokens: 254, 1755
\tracingscantokens primitive: 1657
tracing_scan_tokens_code: 254, 1657, 1659
tracing_stats: 135, 254, 667, 1506, 1513
\tracingstats primitive: 256
tracing_stats_code: 254, 255, 256

Transcript written...: 1513
trap_zero_glue: 1406, 1407, 1414
trapped_given: 807
trick_buf: 54, 58, 337, 339
trick_count: 54, 58, 337, 338, 339
 Trickey, Howard Wellington: 2
trie: 1097, 1098, 1099, 1127, 1129, 1130, 1131, 1135, 1136, 1143, 1502, 1503
trie_back: 1127, 1131, 1133
trie_c: 1124, 1125, 1128, 1130, 1132, 1133, 1136, 1140, 1141, 1855, 1856
trie_char: 1097, 1098, 1100, 1135, 1136, 1858
trie_fix: 1135, 1136
trie_hash: 1124, 1125, 1126, 1127, 1129
trie_l: 1124, 1125, 1126, 1134, 1136, 1137, 1140, 1141, 1856
trie_link: 1097, 1098, 1100, 1127, 1129, 1130, 1131, 1132, 1133, 1135, 1136, 1858
trie_max: 1127, 1129, 1131, 1135, 1502, 1503
trie_min: 1127, 1129, 1130, 1133, 1857
trie_node: 1125, 1126
trie_not_ready: 1068, 1111, 1127, 1128, 1137, 1143, 1502, 1503
trie_o: 1124, 1125, 1136, 1140, 1141, 1856
trie_op: 1097, 1098, 1100, 1101, 1120, 1135, 1136, 1854, 1858
trie_op_hash: 1120, 1121, 1122, 1123, 1125, 1129
trie_op_lang: 1120, 1121, 1122, 1129
trie_op_ptr: 1120, 1121, 1122, 1123, 1502, 1503
trie_op_size: 11, 1098, 1120, 1121, 1123, 1502, 1503
trie_op_val: 1120, 1121, 1122, 1129
trie_pack: 1134, 1143, 1857
trie_pointer: 1097, 1098, 1099, 1124, 1125, 1126, 1127, 1130, 1134, 1136, 1137, 1143, 1858
trie_ptr: 1124, 1128, 1129, 1141
trie_r: 1124, 1125, 1126, 1132, 1133, 1134, 1136, 1140, 1141, 1854, 1855, 1856
trie_ref: 1127, 1129, 1130, 1133, 1134, 1136, 1857
trie_root: 1124, 1126, 1128, 1129, 1135, 1143, 1854, 1857
trie_size: 11, 1097, 1125, 1127, 1129, 1131, 1141, 1503
trie_taken: 1127, 1129, 1130, 1131, 1133
trie_used: 1120, 1121, 1122, 1123, 1502, 1503
true: 4, 16, 31, 34, 37, 45, 46, 49, 51, 53, 71, 77, 88, 97, 98, 104, 105, 106, 107, 112, 114, 115, 186, 187, 274, 276, 278, 304, 333, 349, 350, 358, 368, 383, 384, 387, 398, 400, 404, 433, 439, 456, 466, 470, 473, 479, 487, 488, 512, 527, 534, 538, 542, 550, 552, 560, 589, 605, 619, 649, 656, 665, 666, 669, 683, 685, 686, 689, 692, 693, 698, 699, 702, 705, 706, 720, 726, 727, 737, 746, 747, 748, 749, 750, 766, 791, 793, 794, 795, 799, 801, 802, 804, 823, 839, 851, 882, 895, 967, 1002, 1003, 1004, 1005, 1027, 1030, 1039, 1056, 1057, 1058, 1060, 1080, 1082, 1087, 1088, 1128, 1133, 1139, 1140, 1169, 1197, 1198, 1202, 1207, 1212, 1214, 1218, 1229, 1232, 1258, 1261, 1268, 1279, 1299, 1323, 1341, 1372, 1373, 1396, 1402, 1404, 1414, 1415, 1431, 1436, 1448, 1457, 1461, 1476, 1481, 1516, 1522, 1534, 1537, 1538, 1539, 1540, 1541, 1542, 1544, 1546, 1548, 1549, 1551, 1552, 1553, 1554, 1555, 1558, 1560, 1561, 1563, 1564, 1565, 1567, 1568, 1569, 1572, 1574, 1575, 1580, 1588, 1589, 1590, 1591, 1593, 1594, 1595, 1596, 1597, 1598, 1599, 1600, 1618, 1622, 1629, 1635, 1639, 1645, 1648, 1656, 1662, 1679, 1756, 1767, 1768, 1774, 1775, 1777, 1790, 1793, 1797, 1799, 1819, 1825, 1827, 1830, 1839, 1843, 1856
true: 479
try_break: 1004, 1005, 1015, 1027, 1034, 1038, 1042, 1044, 1045, 1049, 1055
try_prev_break: 999, 1039
two: 101, 102
two_choices: 131
two_halves: 131, 136, 142, 190, 239, 274, 275, 860, 1098, 1143
two_to_the: 117, 118, 120
tx: 439, 450, 1257, 1258, 1259, 1283
type: 4, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 170, 171, 173, 174, 175, 176, 177, 178, 193, 201, 202, 220, 224, 450, 497, 498, 515, 521, 522, 523, 531, 650, 651, 654, 656, 659, 660, 663, 665, 668, 674, 695, 705, 732, 733, 736, 737, 741, 742, 745, 746, 751, 772, 823, 825, 827, 831, 844, 845, 846, 856, 857, 858, 859, 862, 863, 864, 865, 872, 874, 889, 891, 896, 897, 902, 903, 904, 905, 907, 908, 912, 923, 926, 928, 936, 937, 938, 943, 944, 972, 975, 977, 981, 983, 985, 986, 987, 992, 995, 996, 998, 1005, 1006, 1008, 1013, 1017, 1018, 1019, 1020, 1021, 1032, 1034, 1035, 1036, 1037, 1038, 1040, 1041, 1042, 1044, 1046, 1047, 1050, 1051, 1055, 1057, 1073, 1074, 1076, 1080, 1091, 1145, 1147, 1149, 1150, 1153, 1155, 1156, 1158, 1163, 1165, 1170, 1173, 1174, 1177, 1181, 1185, 1186, 1187, 1188, 1190, 1191, 1198, 1217, 1252, 1258, 1265, 1278, 1279, 1283, 1288, 1291, 1299, 1325, 1333, 1336, 1337, 1341, 1343, 1346, 1359, 1363, 1364, 1369, 1380, 1381, 1521, 1529, 1573, 1635, 1636, 1637, 1679, 1691, 1699, 1704, 1707, 1711, 1717, 1719, 1725, 1728, 1733, 1738, 1739, 1746, 1777, 1788, 1789, 1815

Type <return> to proceed...: 85

u: 69, 107, 127, 415, 496, 586, 689, 706, 882, 967, 976, 1106, 1111, 1121, 1435, 1744
u_part: 944, 945, 955, 964, 970, 977
u_template: 329, 336, 346, 964
uc_code: 248, 250, 433
\uccode primitive: 1408
uc_code_base: 248, 253, 1408, 1409, 1464, 1466
uc_hyph: 254, 1068, 1073
\uchyph primitive: 256
uc_hyph_code: 254, 255, 256
un_hbox: 226, 1268, 1285, 1286, 1287
\unhbox primitive: 1285
\unhcopy primitive: 1285
\unkern primitive: 1285
\unpenalty primitive: 1285
\unskip primitive: 1285
un_vbox: 226, 1224, 1272, 1285, 1286, 1287, 1861
\unvbox primitive: 1285
\unvcopy primitive: 1285
unbalance: 415, 417, 422, 425, 499, 503, 1683
Unbalanced output routine: 1204
Unbalanced write...: 1619
Undefined control sequence: 396
undefined_control_sequence: 240, 250, 274, 276, 278, 284, 290, 304, 312, 1496, 1497
undefined.cs: 228, 240, 388, 398, 527, 1404, 1405, 1473, 1766, 1767
undefined_primitive: 275, 281, 394, 395, 527, 1223
under_noad: 863, 866, 872, 874, 909, 937, 1334, 1335
Underfull \hbox...: 836
Underfull \vbox...: 850
\underline primitive: 1334
undump: 1484, 1488, 1490, 1492, 1497, 1501, 1503, 1507, 1655
undump_end: 1484
undump_end_end: 1484
undump_four_ASCII: 1488
undump_hh: 1484, 1497, 1503
undump_int: 1484, 1486, 1490, 1495, 1497, 1501, 1505, 1507
undump_qqq: 1484, 1488, 1501
undump_size: 1484, 1488, 1499, 1503
undump_size_end: 1484
undump_size_end_end: 1484
undump_wd: 1484, 1490, 1495, 1499
undumpimagemeta: 1505
undumptounicode: 1505
unescapehex: 497
\unexpanded primitive: 1686
unfloat: 109, 834, 840, 849, 852, 986, 987
unhyphenated: 995, 1005, 1013, 1040, 1042, 1044
unif_rand: 126, 498
\pdfuniformdeviate primitive: 494
uniform_deviate_code: 494, 495, 497, 498
unity: 101, 103, 119, 132, 182, 204, 479, 594, 1437
Unknown color stack: 1539
\unless primitive: 1762
unless_code: 513, 514, 524, 1668, 1765
unpackage: 1287, 1288
unsave: 303, 305, 967, 976, 1203, 1241, 1246, 1264, 1278, 1297, 1311, 1346, 1352, 1364, 1369, 1372, 1374, 1378
unset_node: 177, 193, 201, 202, 220, 224, 450, 825, 845, 858, 864, 865, 944, 972, 975, 977, 981
update_active: 1037
update_adjust_list: 831
update_heights: 1147, 1149, 1150, 1171, 1174, 1177
update_image_procset: 767
update_terminal: 34, 37, 61, 71, 86, 384, 550, 563, 666, 674, 714, 750, 1458, 1518, 1755
update_width: 1008, 1036
\uppercase primitive: 1464
Use of x doesn't match...: 424
use_err_help: 79, 80, 89, 90, 1461
v: 69, 107, 415, 476, 689, 882, 891, 912, 919, 925, 976, 1006, 1099, 1111, 1121, 1137, 1154, 1316, 1679
v_offset: 265, 644, 668, 669, 755
\voffset primitive: 266
v_offset_code: 265, 266
v_out: 692, 693
v_part: 944, 945, 955, 965, 970, 977
v_template: 329, 336, 347, 416, 965, 1309
vacuous: 466, 470, 471
vadjust: 226, 287, 288, 1275, 1276, 1277, 1278
\valign primitive: 287
val: 682
valign: 226, 287, 288, 1224, 1268, 1308, 1700, 1701
\valign primitive: 287
var_code: 250, 1329, 1333, 1343
var_delimiter: 882, 913, 924, 938
var_used: 135, 143, 148, 182, 667, 1489, 1490
vbadness: 254, 850, 853, 854, 1189, 1194
\vbadieness primitive: 256
vbadness_code: 254, 255, 256
\ vbox primitive: 1249
vbox_group: 291, 1261, 1263, 1661, 1679
vcenter: 226, 287, 288, 1224, 1345
\vccenter primitive: 287
vcenter_group: 291, 1345, 1346, 1661, 1679
vcenter_noad: 863, 866, 872, 874, 909, 937, 1346
vert_break: 1147, 1148, 1153, 1154, 1157, 1159, 1187

very_loose_fit: 993, 995, 1006, 1009, 1010, 1012, 1028, 1842, 1847
vet_glue: 653, 662, 735, 744, 1638
vf_alpha: 706
vf_b_open_in: 713
vf_beta: 706
vf_byte: 712, 714, 715, 717
vf_cur_s: 721, 723, 724, 725
vf_def_font: 712, 715
vf_default_font: 705, 706, 710, 715, 719, 720, 725, 726
vf_e_fnts: 706, 710, 715, 719, 720, 726
vf_error: 710, 712
vf_expand_local_fonts: 705
vf_f: 725, 726
vf_file: 710, 712, 713
vf_i_fnts: 705, 706, 710, 715, 720, 725, 726
vf_id: 710, 714
vf_local_font_num: 705, 706, 710, 715, 719, 720, 726
vf_local_font_warning: 712
vf_max_packet_length: 710, 717, 719
vf_max_recursion: 721, 723, 725
vf_nf: 706, 710, 711, 715, 720
vf_packet_base: 706, 710, 716, 720
vf_packet_length: 710, 725
vf_read_signed: 712, 714, 717
vf_read_unsigned: 712, 715, 717, 719
vf_replace_z: 706
vf_stack: 723, 726
vf_stack_index: 712, 722, 723
vf_stack_ptr: 723, 724, 726
vf_stack_record: 722, 723
vf_stack_size: 719, 721, 722
vf_z: 706
\vfil primitive: 1236
\vfilneg primitive: 1236
\vfill primitive: 1236
vfuzz: 265, 853, 1189, 1194
\vfuzz primitive: 266
vfuzz_code: 265, 266
VIRTEX: 1511
virtual memory: 144
virtual_font_type: 703, 705, 706, 712, 720, 726
Vitter, Jeffrey Scott: 280
vlist_node: 155, 166, 177, 193, 201, 202, 220, 224, 531, 646, 650, 651, 656, 657, 659, 660, 665, 668, 732, 733, 737, 741, 742, 746, 751, 816, 825, 844, 845, 857, 889, 891, 896, 912, 923, 926, 983, 985, 987, 1017, 1018, 1042, 1046, 1047, 1145, 1150, 1155, 1177, 1252, 1258, 1265, 1288, 1325, 1637, 1725, 1733
vlist_out: 619, 642, 643, 646, 647, 651, 656, 657, 660, 665, 666, 668, 727, 728, 869, 1620
vmode: 229, 233, 442, 443, 444, 448, 450, 527, 951, 961, 962, 980, 983, 984, 985, 988, 1202, 1206, 1223, 1224, 1226, 1234, 1235, 1249, 1250, 1251, 1254, 1256, 1257, 1258, 1261, 1268, 1269, 1272, 1276, 1277, 1281, 1283, 1287, 1288, 1289, 1308, 1345, 1421, 1422, 1560, 1561, 1679, 1681
vmove: 226, 1226, 1249, 1250, 1251, 1681
vpack: 254, 816, 817, 818, 844, 881, 911, 914, 935, 975, 980, 1154, 1198, 1278, 1346
vpackage: 844, 972, 1154, 1194, 1264
vrule: 226, 287, 288, 489, 1234, 1262, 1268
\vrule primitive: 287
vsizer: 265, 1157, 1164
\vsizer primitive: 266
vsizer_code: 265, 266
vskip: 226, 1224, 1235, 1236, 1237, 1256, 1272
\vskip primitive: 1236
vsplit: 1144, 1154, 1155, 1157, 1260, 1809, 1825, 1826
\vsplit needs a *\vbox*: 1155
\vsplit primitive: 1249
vsplit_code: 1249, 1250, 1257, 1515, 1859, 1861, 1862
vsplit_init: 1154, 1825, 1826
\vss primitive: 1236
\vtop primitive: 1249
vtop_code: 1249, 1250, 1261, 1263, 1264
vtop_group: 291, 1261, 1263, 1661, 1679
w: 132, 165, 174, 297, 300, 301, 634, 690, 823, 844, 882, 891, 914, 967, 976, 1083, 1171, 1301, 1316, 1376, 1414, 1480, 1481, 1529, 1530, 1683, 1719, 1753, 1756, 1774, 1776, 1819, 1839, 1840
w_close: 28, 1509, 1517
w_make_name_string: 551, 1508
w_open_in: 27, 550
w_open_out: 27, 1508
wait: 1189, 1197, 1198, 1199
wake_up_terminal: 34, 37, 51, 71, 73, 385, 510, 550, 556, 686, 1472, 1475, 1481, 1513, 1518
warn: 693
warn_dest_dup: 1564, 1565, 1637
warn_pdfpagebox: 1550, 1551, 1552
Warning: end of file when...: 1777
Warning: end of...: 1774, 1776
warning_index: 327, 353, 360, 415, 416, 421, 422, 424, 427, 497, 499, 505, 508, 950, 953, 1683
warning_issued: 76, 263, 686, 1515, 1774, 1776, 1777
was_free: 183, 185, 189
was_hi_min: 183, 184, 185, 189

was_lo_max: 183, 184, 185, 189
was_mem_end: 183, 184, 185, 189
\wd primitive: 442
WEB: 1, 4, 38, 40, 50, 1486
what_lang: 1521, 1603, 1609, 1610, 1624, 1625
what_lhm: 1521, 1603, 1609, 1610, 1624, 1625
what_rhm: 1521, 1603, 1609, 1610, 1624, 1625
whatsit_node: 164, 166, 193, 201, 220, 224, 650, 659, 732, 741, 772, 825, 845, 906, 937, 1005, 1042, 1073, 1076, 1145, 1150, 1177, 1325, 1521, 1529, 1573, 1635, 1637, 1700, 1733
\widowpenalties primitive: 1864
widow_penalties_loc: 248, 1864, 1865
widow_penalties_ptr: 1067, 1864
widow_penalty: 254, 990, 1067
\widowpenalty primitive: 256
widow_penalty_code: 254, 255, 256
width: 489
width: 153, 154, 156, 157, 165, 168, 169, 173, 174, 196, 201, 202, 205, 209, 210, 450, 455, 457, 477, 488, 489, 498, 580, 632, 634, 638, 644, 650, 651, 653, 654, 659, 661, 662, 663, 669, 705, 732, 733, 735, 736, 741, 743, 744, 745, 752, 755, 823, 825, 827, 832, 833, 842, 844, 845, 846, 847, 855, 859, 864, 882, 885, 890, 891, 892, 893, 907, 914, 920, 923, 925, 926, 933, 934, 935, 944, 955, 969, 972, 973, 974, 977, 978, 979, 980, 982, 983, 984, 985, 986, 987, 1003, 1005, 1013, 1014, 1017, 1018, 1042, 1044, 1046, 1047, 1057, 1146, 1153, 1173, 1178, 1181, 1186, 1220, 1222, 1232, 1269, 1271, 1325, 1326, 1377, 1379, 1383, 1407, 1417, 1418, 1548, 1552, 1556, 1565, 1573, 1630, 1637, 1638, 1699, 1714, 1716, 1719, 1720, 1721, 1722, 1725, 1728, 1729, 1734, 1736, 1738, 1740, 1745, 1746, 1780, 1790, 1794, 1795, 1796, 1798, 1853
width_base: 576, 578, 580, 592, 595, 598, 603, 705, 706, 1500, 1501
width_index: 569, 576
width_offset: 153, 442, 443, 1425
Wirth, Niklaus: 10
wlog: 56, 58, 562, 1513, 1514
wlog_cr: 56, 57, 58, 562, 1513
wlog_ln: 56, 1513, 1514
word_define: 1392, 1406, 1410, 1839
word_file: 25, 27, 28, 131, 551, 1483
word_node_size: 1820, 1821, 1837, 1841
words: 222, 223, 224, 1604, 1733
wrap_lig: 1087, 1088
wrapup: 1212, 1217, 1218
write: 37, 56, 58, 624
\write primitive: 1524
write_action: 782, 1563, 1579, 1630
write_dvi: 624, 625, 626
write_file: 57, 58, 1522, 1622, 1626
write_fontstuff: 801
write_image: 778
write_ln: 35, 37, 51, 56, 57
write_loc: 1491, 1492, 1524, 1525, 1618
write_node: 1521, 1524, 1526, 1528, 1603, 1604, 1605, 1620, 1622
write_node_size: 1521, 1530, 1532, 1533, 1534, 1538, 1604, 1605
write_open: 1522, 1523, 1617, 1622, 1626
write_out: 1617, 1622
write_pdf: 685
write_stream: 1521, 1530, 1534, 1602, 1617, 1622
write_stream_length: 685
write_text: 329, 336, 345, 1520, 1618
write_tokens: 727, 1521, 1532, 1533, 1534, 1603, 1604, 1605, 1615, 1618
write_zip: 685
writing: 605
wterm: 56, 58, 61
wterm_cr: 56, 57, 58
wterm_ln: 56, 61, 550, 1481, 1512, 1517
Wyatt, Douglas Kirk: 2
w0: 612, 613, 631, 636, 719, 726
w1: 612, 613, 634, 719, 726
w2: 612
w3: 612
w4: 612
x: 100, 105, 106, 107, 119, 124, 126, 127, 614, 627, 689, 823, 844, 882, 896, 902, 911, 913, 914, 919, 925, 932, 1301, 1316, 1480, 1481, 1552, 1744, 1793, 1799
x_height: 573, 584, 585, 673, 914, 1301
x_height_code: 573, 584
x_leaders: 167, 208, 655, 1249, 1250
\xleaders primitive: 1249
x_over_n: 106, 879, 892, 893, 1163, 1185, 1186, 1187, 1418
x_token: 386, 407, 504, 1215, 1330
xchr: 20, 21, 23, 24, 38, 49, 58, 545
xclause: 16
\xdef primitive: 1386
xeq_level: 271, 272, 290, 300, 301, 305, 1482
xn_over_d: 107, 481, 483, 484, 594, 892, 1222, 1438
xord: 20, 24, 31, 52, 53, 549, 551
xpand: 499, 503, 505
xr: 1552
xray: 226, 1468, 1469, 1470, 1675, 1684, 1689
xrealloc_array: 678, 686, 698, 1505
xref_offset_width: 814, 1513
xspace_skip: 242, 1221

\xspaceskip primitive: 244
xspace_skip_code: 242, 243, 244, 1221
xxx1: 612, 613, 719, 726, 1615
xxx2: 612
xxx3: 612
xxx4: 612, 613, 1615
x0: 612, 613, 631, 636, 719, 726
x1: 612, 613, 634, 719, 726
x2: 612
x3: 612
x4: 612
y: 105, 119, 126, 882, 902, 911, 913, 914, 919,
925, 932, 1793
y_here: 635, 636, 638, 639, 640
y_OK: 635, 636, 639
y_seen: 638, 639
year: 254, 259, 645, 792, 1508
\year primitive: 256
year_code: 254, 255, 256
You already have nine...: 502
You can't \insert255: 1277
You can't dump...: 1482
You can't use \hrule...: 1273
You can't use \long...: 1391
You can't use \unless...: 1765
You can't use a prefix with x: 1390
You can't use x after ...: 454, 1415
You can't use x in y mode: 1227
You have to increase POOLSIZE: 52
You want to edit file x: 84
you.cant: 1227, 1228, 1258, 1284
yr: 1552
yz_OK: 635, 636, 637, 639
y0: 612, 613, 621, 631, 636, 719, 726
y1: 612, 613, 634, 640, 719, 726
y2: 612, 621
y3: 612
y4: 612
z: 119, 586, 882, 902, 919, 925, 932, 1099, 1104,
1130, 1136, 1376, 1744
z_here: 635, 636, 638, 639, 641
z_OK: 635, 636, 639
z_seen: 638, 639
Zabala Salellas, Ignacio Andrés: 2
zero_glue: 180, 193, 242, 246, 450, 453, 488, 908,
978, 1005, 1063, 1219, 1220, 1221, 1349, 1407,
1740, 1782, 1790, 1809, 1820, 1821
zero_token: 471, 478, 499, 502, 505
zip_finish: 680, 685
zip_write_state: 679, 680, 681, 685
zip_writing: 680, 685
z0: 612, 613, 631, 636, 719, 726
z1: 612, 613, 634, 641, 719, 726
z2: 612
z3: 612
z4: 612

⟨ (pdfTeX) Move down or output leaders 744 ⟩ Used in section 741.
 ⟨ (pdfTeX) Move right or output leaders 735 ⟩ Used in section 732.
 ⟨ (pdfTeX) Output a box in a vlist 742 ⟩ Used in section 741.
 ⟨ (pdfTeX) Output a box in an hlist 733 ⟩ Used in section 732.
 ⟨ (pdfTeX) Output a leader box at *cur_h*, then advance *cur_h* by *leader_wd + lx* 737 ⟩ Used in section 736.
 ⟨ (pdfTeX) Output a leader box at *cur_v*, then advance *cur_v* by *leader_ht + lx* 746 ⟩ Used in section 745.
 ⟨ (pdfTeX) Output a rule in a vlist, **goto** *next_p* 743 ⟩ Used in section 741.
 ⟨ (pdfTeX) Output a rule in an hlist 734 ⟩ Used in section 732.
 ⟨ (pdfTeX) Output leaders in a vlist, **goto** *fin_rule* if a rule or to *next_p* if done 745 ⟩ Used in section 744.
 ⟨ (pdfTeX) Output leaders in an hlist, **goto** *fin_rule* if a rule or to *next_p* if done 736 ⟩ Used in section 735.
 ⟨ (pdfTeX) Ship box *p* out 751 ⟩ Used in section 750.
 ⟨ Accumulate the constant until *cur_tok* is not a suitable digit 471 ⟩ Used in section 470.
 ⟨ Add the width of node *s* to *act_width* 1047 ⟩ Used in section 1045.
 ⟨ Add the width of node *s* to *break_width* 1018 ⟩ Used in section 1016.
 ⟨ Add the width of node *s* to *disc_width* 1046 ⟩ Used in section 1045.
 ⟨ Adjust for the magnification ratio 483 ⟩ Used in section 479.
 ⟨ Adjust for the setting of \globaldefs 1392 ⟩ Used in section 1389.
 ⟨ Adjust *shift_up* and *shift_down* for the case of a fraction line 922 ⟩ Used in section 919.
 ⟨ Adjust *shift_up* and *shift_down* for the case of no fraction line 921 ⟩ Used in section 919.
 ⟨ Adjust the LR stack for the *hlist_out* routine; if necessary reverse an hlist segment and **goto** *reswitch* 1717 ⟩
 Used in section 1716.
 ⟨ Adjust the LR stack for the *hpack* routine 1711 ⟩ Used in section 825.
 ⟨ Adjust the LR stack for the *init_math* routine 1737 ⟩ Used in section 1736.
 ⟨ Adjust the LR stack for the *just_reverse* routine 1739 ⟩ Used in section 1738.
 ⟨ Adjust the LR stack for the *post_line_break* routine 1708 ⟩ Used in sections 1055, 1057, and 1707.
 ⟨ Adjust the additional data for last line 1849 ⟩ Used in section 1027.
 ⟨ Adjust the final line of the paragraph 1853 ⟩ Used in section 1039.
 ⟨ Adjust transformation matrix for the magnification ratio 758 ⟩ Used in section 757.
 ⟨ Advance *cur_p* to the node following the present string of characters 1043 ⟩ Used in section 1042.
 ⟨ Advance past a whatsit node in the *line_break* loop 1609 ⟩ Used in section 1042.
 ⟨ Advance past a whatsit node in the pre-hyphenation loop 1610 ⟩ Used in section 1073.
 ⟨ Advance *r*; **goto** *found* if the parameter delimiter has been fully matched, otherwise **goto** *continue* 420 ⟩
 Used in section 418.
 ⟨ Allocate entire node *p* and **goto** *found* 147 ⟩ Used in section 145.
 ⟨ Allocate from the top of node *p* and **goto** *found* 146 ⟩ Used in section 145.
 ⟨ Allocate memory for the new virtual font 716 ⟩ Used in section 712.
 ⟨ Apologize for inability to do the operation now, unless \unskip follows non-glue 1284 ⟩ Used in section 1283.
 ⟨ Apologize for not loading the font, **goto** *done* 593 ⟩ Used in section 592.
 ⟨ Append a ligature and/or kern to the translation; **goto** *continue* if the stack of inserted ligatures is
 nonempty 1087 ⟩ Used in section 1083.
 ⟨ Append a new leader node that uses *cur_box* 1256 ⟩ Used in section 1253.
 ⟨ Append a new letter or a hyphen level 1139 ⟩ Used in section 1138.
 ⟨ Append a new letter or hyphen 1114 ⟩ Used in section 1112.
 ⟨ Append a normal inter-word space to the current list, then **goto** *big_switch* 1219 ⟩ Used in section 1207.
 ⟨ Append a penalty node, if a nonzero penalty is appropriate 1067 ⟩ Used in section 1056.
 ⟨ Append an insertion to the current page and **goto** *contribute* 1185 ⟩ Used in section 1177.
 ⟨ Append any *new_hlist* entries for *q*, and any appropriate penalties 943 ⟩ Used in section 936.
 ⟨ Append box *cur_box* to the current list, shifted by *box_context* 1254 ⟩ Used in section 1253.
 ⟨ Append character *cur_chr* and the following characters (if any) to the current hlist in the current font;
 goto *reswitch* when a non-character has been fetched 1211 ⟩ Used in section 1207.
 ⟨ Append characters of *hu*[*j* ..] to *major_tail*, advancing *j* 1094 ⟩ Used in section 1093.
 ⟨ Append inter-element spacing based on *r_type* and *t* 942 ⟩ Used in section 936.

⟨ Append tabskip glue and an empty box to list u , and update s and t as the prototype nodes are passed 985 ⟩
 Used in section 984.

⟨ Append the accent with appropriate kerns, then set $p \leftarrow q$ 1303 ⟩ Used in section 1301.

⟨ Append the current tabskip glue to the preamble list 954 ⟩ Used in section 953.

⟨ Append the display and perhaps also the equation number 1382 ⟩ Used in section 1377.

⟨ Append the glue or equation number following the display 1383 ⟩ Used in section 1377.

⟨ Append the glue or equation number preceding the display 1381 ⟩ Used in section 1377.

⟨ Append the new box to the current vertical list, followed by the list of special nodes taken out of the box by the packager 1065 ⟩ Used in section 1056.

⟨ Append the value n to list p 1115 ⟩ Used in section 1114.

⟨ Assign the values $depth_threshold \leftarrow show_box_depth$ and $breadth_max \leftarrow show_box_breadth$ 254 ⟩
 Used in section 216.

⟨ Assignments 1395, 1396, 1399, 1402, 1403, 1404, 1406, 1410, 1412, 1413, 1419, 1420, 1426, 1430, 1431, 1434, 1442 ⟩
 Used in section 1389.

⟨ Attach list p to the current list, and record its length; then finish up and return 1298 ⟩ Used in section 1297.

⟨ Attach the limits to y and adjust $height(v)$, $depth(v)$ to account for their presence 927 ⟩ Used in section 926.

⟨ Back up an outer control sequence so that it can be reread 359 ⟩ Used in section 358.

⟨ Basic printing procedures 57, 58, 59, 60, 62, 63, 64, 65, 284, 285, 544, 875, 1602, 1822 ⟩ Used in section 4.

⟨ Break the current page at node p , put it in box 255, and put the remaining nodes on the contribution list 1194 ⟩ Used in section 1191.

⟨ Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list 1052 ⟩ Used in section 991.

⟨ Build a character packet 717 ⟩ Used in section 712.

⟨ Build a linked list of free objects 812 ⟩ Used in sections 813 and 814.

⟨ Calculate DVI page dimensions and margins 644 ⟩ Used in section 645.

⟨ Calculate page dimensions and margins 755 ⟩ Used in section 752.

⟨ Calculate the length, l , and the shift amount, s , of the display lines 1327 ⟩ Used in section 1323.

⟨ 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 1324 ⟩ Used in section 1323.

⟨ Calculate variations of marginal kerns 822 ⟩ Used in section 1027.

⟨ Call the packaging subroutine, setting $just_box$ to the justified box 1066 ⟩ Used in section 1056.

⟨ 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 1042 ⟩ Used in section 1039.

⟨ Carry out a ligature replacement, updating the cursor structure and possibly advancing j ; goto $continue$ if the cursor doesn't advance, otherwise goto $done$ 1088 ⟩ Used in section 1086.

⟨ Case statement to copy different types and set $words$ to the number of initial words not yet copied 224 ⟩
 Used in section 223.

⟨ Cases for 'Fetch the $dead_cycles$ or the $insert_penalties'$ 1694 ⟩ Used in section 445.

⟨ Cases for evaluation of the current term 1791, 1795, 1796, 1798 ⟩ Used in section 1783.

⟨ Cases for fetching a dimension value 1671, 1674, 1804 ⟩ Used in section 450.

⟨ Cases for fetching a glue value 1807 ⟩ Used in section 1780.

⟨ Cases for fetching a mu value 1808 ⟩ Used in section 1780.

⟨ Cases for fetching an integer value 1651, 1665, 1668, 1803 ⟩ Used in section 450.

⟨ Cases for noads that can follow a bin_noad 909 ⟩ Used in section 904.

⟨ Cases for nodes that can appear in an mlist, after which we goto $done_with_node$ 906 ⟩ Used in section 904.

⟨ Cases for $alter_integer$ 1696 ⟩ Used in section 1424.

⟨ Cases for $conditional$ 1766, 1767, 1769 ⟩ Used in section 527.

⟨ Cases for do_marks 1826, 1828, 1829, 1831 ⟩ Used in section 1825.

⟨ Cases for $eq_destroy$ 1834 ⟩ Used in section 297.

⟨ Cases for $input$ 1749 ⟩ Used in section 404.

⟨ Cases for *print_param* 1659, 1700 ⟩ Used in section 255.
 ⟨ Cases for *show_whatever* 1677, 1691 ⟩ Used in section 1471.
 ⟨ Cases of DVI commands that can appear in character packet 719 ⟩ Used in section 717.
 ⟨ Cases of ‘Let *d* be the natural width’ that need special treatment 1736 ⟩ Used in section 1325.
 ⟨ Cases of *assign_toks* for *print_cmd_chr* 1658 ⟩ Used in section 249.
 ⟨ Cases of *expandafter* for *print_cmd_chr* 1763 ⟩ Used in section 288.
 ⟨ Cases of *flush_node_list* that arise in mlists only 874 ⟩ Used in section 220.
 ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1263, 1278, 1296, 1310, 1311, 1346, 1351, 1364 ⟩ Used in section 1246.
 ⟨ Cases of *hlist_out* that arise in mixed direction text only 1720 ⟩ Used in sections 650 and 732.
 ⟨ Cases of *if_test* for *print_cmd_chr* 1764 ⟩ Used in section 514.
 ⟨ Cases of *input* for *print_cmd_chr* 1748 ⟩ Used in section 403.
 ⟨ Cases of *last_item* for *print_cmd_chr* 1650, 1664, 1667, 1670, 1673, 1779, 1802, 1806 ⟩ Used in section 443.
 ⟨ Cases of *left_right* for *print_cmd_chr* 1698 ⟩ Used in section 1367.
 ⟨ Cases of *main_control* for *hmode + valign* 1703 ⟩ Used in section 1308.
 ⟨ Cases of *main_control* that are for extensions to TeX 1527 ⟩ Used in section 1223.
 ⟨ Cases of *main_control* that are not part of the inner loop 1223 ⟩ Used in section 1207.
 ⟨ Cases of *main_control* that build boxes and lists 1234, 1235, 1241, 1245, 1251, 1268, 1270, 1272, 1275, 1280, 1282, 1287, 1290, 1294, 1300, 1304, 1308, 1312, 1315, 1318, 1328, 1332, 1336, 1340, 1342, 1345, 1349, 1353, 1358, 1368, 1371 ⟩ Used in section 1223.
 ⟨ Cases of *main_control* that don’t depend on *mode* 1388, 1446, 1449, 1452, 1454, 1463, 1468 ⟩ Used in section 1223.
 ⟨ Cases of *prefix* for *print_cmd_chr* 1771 ⟩ Used in section 1387.
 ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 245, 249, 257, 267, 288, 357, 403, 411, 438, 443, 495, 514, 518, 957, 1161, 1231, 1237, 1250, 1267, 1286, 1293, 1321, 1335, 1348, 1357, 1367, 1387, 1398, 1401, 1409, 1429, 1433, 1439, 1441, 1451, 1456, 1465, 1470, 1473, 1526 ⟩ Used in section 320.
 ⟨ Cases of *read* for *print_cmd_chr* 1760 ⟩ Used in section 288.
 ⟨ Cases of *register* for *print_cmd_chr* 1832 ⟩ Used in section 438.
 ⟨ Cases of *reverse* that need special treatment 1726, 1727, 1728 ⟩ Used in section 1725.
 ⟨ Cases of *set_page_int* for *print_cmd_chr* 1693 ⟩ Used in section 443.
 ⟨ Cases of *set_shape* for *print_cmd_chr* 1865 ⟩ Used in section 288.
 ⟨ Cases of *show_node_list* that arise in mlists only 866 ⟩ Used in section 201.
 ⟨ Cases of *the* for *print_cmd_chr* 1687 ⟩ Used in section 288.
 ⟨ Cases of *toks_register* for *print_cmd_chr* 1833 ⟩ Used in section 288.
 ⟨ Cases of *un_vbox* for *print_cmd_chr* 1862 ⟩ Used in section 1286.
 ⟨ Cases of *valign* for *print_cmd_chr* 1702 ⟩ Used in section 288.
 ⟨ Cases of *xray* for *print_cmd_chr* 1676, 1685, 1690 ⟩ Used in section 1470.
 ⟨ Cases where character is ignored 367 ⟩ Used in section 366.
 ⟨ Change buffered instruction to *y* or *w* and **goto found** 640 ⟩ Used in section 639.
 ⟨ Change buffered instruction to *z* or *x* and **goto found** 641 ⟩ Used in section 639.
 ⟨ Change current mode to *-vmode* for *\halign*, *-hmode* for *\valign* 951 ⟩ Used in section 950.
 ⟨ Change discretionary to compulsory and set *disc_break ← true* 1058 ⟩ Used in section 1057.
 ⟨ Change font *dvi_f* to *f* 649 ⟩ Used in section 648.
 ⟨ Change state if necessary, and **goto switch** if the current character should be ignored, or **goto reswitch** if the current character changes to another 366 ⟩ Used in section 365.
 ⟨ Change the case of the token in *p*, if a change is appropriate 1467 ⟩ Used in section 1466.
 ⟨ Change the current style and **goto delete_q** 939 ⟩ Used in section 937.
 ⟨ Change the interaction level and **return** 86 ⟩ Used in section 84.
 ⟨ Change this node to a style node followed by the correct choice, then **goto done_with_node** 907 ⟩ Used in section 906.
 ⟨ Character *k* cannot be printed 49 ⟩ Used in section 48.
 ⟨ Character *s* is the current new-line character 262 ⟩ Used in sections 58 and 59.
 ⟨ Check flags of unavailable nodes 188 ⟩ Used in section 185.

⟨ Check for LR anomalies at the end of *hlist_out* 1718 ⟩ Used in section 1715.
 ⟨ Check for LR anomalies at the end of *hpack* 1712 ⟩ Used in section 823.
 ⟨ Check for LR anomalies at the end of *ship_out* 1730 ⟩ Used in sections 666 and 750.
 ⟨ Check for charlist cycle 596 ⟩ Used in section 595.
 ⟨ Check for improper alignment in displayed math 952 ⟩ Used in section 950.
 ⟨ Check for non-existing destinations 796 ⟩ Used in section 794.
 ⟨ Check for non-existing pages 799 ⟩ Used in section 794.
 ⟨ Check for non-existing structure destinations 798 ⟩ Used in section 794.
 ⟨ Check for special treatment of last line of paragraph 1843 ⟩ Used in section 1003.
 ⟨ Check if node *p* is a new champion breakpoint; then **goto** *done* if *p* is a forced break or if the page-so-far
 is already too full 1151 ⟩ Used in section 1149.
 ⟨ Check if node *p* is a new champion breakpoint; then if it is time for a page break, prepare for output, and
 either fire up the user's output routine and **return** or ship out the page and **goto** *done* 1182 ⟩
 Used in section 1174.
 ⟨ Check single-word *avail* list 186 ⟩ Used in section 185.
 ⟨ Check that another \$ follows 1375 ⟩ Used in sections 1372, 1372, and 1384.
 ⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set
 danger \leftarrow true 1373 ⟩ Used in sections 1372 and 1372.
 ⟨ Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been
 found, otherwise **goto** *done1* 1076 ⟩ Used in section 1071.
 ⟨ Check the “constant” values for consistency 14, 129, 312, 548, 1427 ⟩ Used in section 1512.
 ⟨ Check the pool check sum 53 ⟩ Used in section 52.
 ⟨ Check variable-size *avail* list 187 ⟩ Used in section 185.
 ⟨ Clean up the memory by removing the break nodes 1041 ⟩ Used in sections 991 and 1039.
 ⟨ Clear dimensions to zero 824 ⟩ Used in sections 823 and 844.
 ⟨ Clear off top level from *save_stack* 304 ⟩ Used in section 303.
 ⟨ Close the format file 1509 ⟩ Used in section 1480.
 ⟨ Coerce glue to a dimension 477 ⟩ Used in sections 475 and 481.
 ⟨ Compiler directives 9 ⟩ Used in section 4.
 ⟨ Complain about an undefined family and set *cur_i* null 899 ⟩ Used in section 898.
 ⟨ Complain about an undefined macro 396 ⟩ Used in section 391.
 ⟨ Complain about missing \endcsname 399 ⟩ Used in sections 398 and 1767.
 ⟨ Complain about unknown unit and **goto** *done2* 485 ⟩ Used in section 484.
 ⟨ Complain that \the can't do this; give zero result 454 ⟩ Used in section 439.
 ⟨ Complain that the user should have said \mathaccent 1344 ⟩ Used in section 1343.
 ⟨ Compleat the incompleat noad 1363 ⟩ Used in section 1362.
 ⟨ Complete a potentially long \show command 1476 ⟩ Used in section 1471.
 ⟨ Compute $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$ 113 ⟩ Used in section 112.
 ⟨ Compute $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$ 116 ⟩ Used in section 114.
 ⟨ Compute $f = \lfloor xn/d + \frac{1}{2} \rfloor$ 1800 ⟩ Used in section 1799.
 ⟨ Compute result of *multiply* or *divide*, put it in *cur_val* 1418 ⟩ Used in section 1414.
 ⟨ Compute result of *register* or *advance*, put it in *cur_val* 1416 ⟩ Used in section 1414.
 ⟨ Compute the amount of skew 917 ⟩ Used in section 914.
 ⟨ Compute the badness, *b*, of the current page, using *awful_bad* if the box is too full 1184 ⟩
 Used in section 1182.
 ⟨ Compute the badness, *b*, using *awful_bad* if the box is too full 1152 ⟩ Used in section 1151.
 ⟨ Compute the demerits, *d*, from *r* to *cur_p* 1035 ⟩ Used in section 1031.
 ⟨ Compute the discretionary *break_width* values 1016 ⟩ Used in section 1013.
 ⟨ Compute the hash code *h* 280 ⟩ Used in section 278.
 ⟨ Compute the magic offset 941 ⟩ Used in section 1517.
 ⟨ Compute the mark pointer for mark type *t* and class *cur_val* 1824 ⟩ Used in section 412.

⟨ Compute the minimum suitable height, w , and the corresponding number of extension steps, n ; also set $width(b)$ 890 ⟩ Used in section 889.
 ⟨ Compute the new line width 1026 ⟩ Used in section 1011.
 ⟨ Compute the primitive code h 283 ⟩ Used in section 281.
 ⟨ Compute the register location l and its type p ; but return if invalid 1415 ⟩ Used in section 1414.
 ⟨ Compute the sum of two glue specs 1417 ⟩ Used in section 1416.
 ⟨ Compute the sum or difference of two glue specs 1794 ⟩ Used in section 1792.
 ⟨ Compute the trie op code, v , and set $l \leftarrow 0$ 1142 ⟩ Used in section 1140.
 ⟨ Compute the values of $break_width$ 1013 ⟩ Used in section 1012.
 ⟨ Consider a node with matching width; goto *found* if it's a hit 639 ⟩ Used in section 638.
 ⟨ 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 1027 ⟩
 Used in section 1005.
 ⟨ Constants in the outer block 11, 675, 679, 695, 721, 1631 ⟩ Used in section 4.
 ⟨ Construct a box with limits above and below it, skewed by δ 926 ⟩ Used in section 925.
 ⟨ Construct a sub/superscript combination box x , with the superscript offset by δ 935 ⟩
 Used in section 932.
 ⟨ Construct a subscript box x when there is no superscript 933 ⟩ Used in section 932.
 ⟨ Construct a superscript box x 934 ⟩ Used in section 932.
 ⟨ Construct a vlist box for the fraction, according to $shift_up$ and $shift_down$ 923 ⟩ Used in section 919.
 ⟨ Construct an extensible character in a new box b , using recipe $rem_byte(q)$ and font f 889 ⟩
 Used in section 886.
 ⟨ Contribute an entire group to the current parameter 425 ⟩ Used in section 418.
 ⟨ Contribute the recently matched tokens to the current parameter, and goto *continue* if a partial match is
 still in effect; but abort if $s = null$ 423 ⟩ Used in section 418.
 ⟨ Convert a final bin_noad to an ord_noad 905 ⟩ Used in sections 902 and 904.
 ⟨ Convert cur_val to a lower level 455 ⟩ Used in section 439.
 ⟨ Convert math glue to ordinary glue 908 ⟩ Used in section 906.
 ⟨ Convert $nucleus(q)$ to an hlist and attach the sub/superscripts 930 ⟩ Used in section 904.
 ⟨ Convert string s into a new pseudo file 1754 ⟩ Used in section 1753.
 ⟨ Copy the tabskip glue between columns 971 ⟩ Used in section 967.
 ⟨ Copy the templates from node cur_loop into node p 970 ⟩ Used in section 969.
 ⟨ Copy the token list 492 ⟩ Used in section 491.
 ⟨ Create a character node p for $nucleus(q)$, possibly followed by a kern node for the italic correction, and set
 δ to the italic correction if a subscript is present 931 ⟩ Used in section 930.
 ⟨ Create a character node q for the next character, but set $q \leftarrow null$ if problems arise 1302 ⟩
 Used in section 1301.
 ⟨ Create a new array element of type t with index i 1820 ⟩ Used in section 1819.
 ⟨ Create a new glue specification whose width is cur_val ; scan for its stretch and shrink components 488 ⟩
 Used in section 487.
 ⟨ Create a page insertion node with $subtype(r) = qi(n)$, and include the glue correction for box n in the
 current page state 1186 ⟩ Used in section 1185.
 ⟨ Create an active breakpoint representing the beginning of the paragraph 1040 ⟩ Used in section 1039.
 ⟨ Create and append a discretionary node as an alternative to the unhyphenated word, and continue to
 develop both branches until they become equivalent 1091 ⟩ Used in section 1090.
 ⟨ Create equal-width boxes x and z for the numerator and denominator, and compute the default amounts
 $shift_up$ and $shift_down$ by which they are displaced from the baseline 920 ⟩ Used in section 919.
 ⟨ Create link annotations for the current hbox if needed 730 ⟩ Used in section 729.
 ⟨ Create new active nodes for the best feasible breaks just found 1012 ⟩ Used in section 1011.
 ⟨ Create the *format.ident*, open the format file, and inform the user that dumping has begun 1508 ⟩
 Used in section 1480.
 ⟨ Create thread for the current vbox if needed 739 ⟩ Used in section 738.

⟨ Current *mem* equivalent of glue parameter number *n* 242 ⟩ Used in sections 170 and 172.
 ⟨ Deactivate node *r* 1036 ⟩ Used in section 1027.
 ⟨ Declare ε-TEX procedures for expanding 1752, 1810, 1815, 1819 ⟩ Used in section 388.
 ⟨ Declare ε-TEX procedures for scanning 1682, 1772, 1781, 1786 ⟩ Used in section 435.
 ⟨ Declare ε-TEX procedures for token lists 1683, 1753 ⟩ Used in section 490.
 ⟨ Declare ε-TEX procedures for tracing and input 306, 1661, 1662, 1756, 1757, 1774, 1776, 1777, 1821, 1823, 1837, 1838, 1839, 1840, 1841 ⟩ Used in section 290.
 ⟨ Declare ε-TEX procedures for use by *main_control* 1656, 1679, 1695 ⟩ Used in section 991.
 ⟨ Declare action procedures for use by *main_control* 1221, 1225, 1227, 1228, 1229, 1232, 1238, 1239, 1242, 1247, 1248, 1253, 1257, 1262, 1264, 1269, 1271, 1273, 1274, 1277, 1279, 1281, 1283, 1288, 1291, 1295, 1297, 1301, 1305, 1307, 1309, 1313, 1314, 1316, 1320, 1329, 1333, 1337, 1338, 1341, 1343, 1350, 1352, 1354, 1359, 1369, 1372, 1378, 1389, 1448, 1453, 1457, 1466, 1471, 1480, 1528, 1624 ⟩ Used in section 1207.
 ⟨ Declare math construction procedures 910, 911, 912, 913, 914, 919, 925, 928, 932, 938 ⟩ Used in section 902.
 ⟨ Declare procedures for preprocessing hyphenation patterns 1121, 1125, 1126, 1130, 1134, 1136, 1137, 1143 ⟩ Used in section 1119.
 ⟨ Declare procedures needed for displaying the elements of mlists 867, 868, 870 ⟩ Used in section 197.
 ⟨ Declare procedures needed for expressions 1782, 1787 ⟩ Used in section 487.
 ⟨ Declare procedures needed in *do_extension* 1529, 1530, 1537, 1552, 1556, 1562, 1566, 1573, 1577, 1587, 1600 ⟩ Used in section 1528.
 ⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1615, 1617, 1620, 1719, 1723 ⟩ Used in section 647.
 ⟨ Declare procedures needed in *pdf_hlist_out*, *pdf_vlist_out* 727, 772, 778, 785, 1564, 1630, 1635, 1636, 1637 ⟩ Used in section 729.
 ⟨ Declare procedures that need to be declared forward for pdfTeX 686, 689, 698, 699, 700, 703, 1545, 1555 ⟩ Used in section 190.
 ⟨ Declare procedures that scan font-related stuff 604, 605 ⟩ Used in section 435.
 ⟨ Declare procedures that scan restricted classes of integers 459, 460, 461, 462, 463, 1811 ⟩ Used in section 435.
 ⟨ Declare subprocedures for *after_math* 1744 ⟩ Used in section 1372.
 ⟨ Declare subprocedures for *init_math* 1733, 1738 ⟩ Used in section 1316.
 ⟨ Declare subprocedures for *line_break* 1002, 1005, 1053, 1072, 1119 ⟩ Used in section 991.
 ⟨ Declare subprocedures for *prefixed_command* 1393, 1407, 1414, 1421, 1422, 1423, 1424, 1425, 1435, 1443 ⟩ Used in section 1389.
 ⟨ Declare subprocedures for *scan_expr* 1793, 1797, 1799 ⟩ Used in section 1782.
 ⟨ Declare subprocedures for *var_delimiter* 885, 887, 888 ⟩ Used in section 882.
 ⟨ Declare the function called *do_marks* 1825 ⟩ Used in section 1154.
 ⟨ Declare the function called *fin_mlist* 1362 ⟩ Used in section 1352.
 ⟨ Declare the function called *open_fmt_file* 550 ⟩ Used in section 1481.
 ⟨ Declare the function called *reconstitute* 1083 ⟩ Used in section 1072.
 ⟨ Declare the procedure called *align_peek* 961 ⟩ Used in section 976.
 ⟨ Declare the procedure called *fire_up* 1189 ⟩ Used in section 1171.
 ⟨ Declare the procedure called *get_preamble_token* 958 ⟩ Used in section 950.
 ⟨ Declare the procedure called *handle_right_brace* 1246 ⟩ Used in section 1207.
 ⟨ Declare the procedure called *init_span* 963 ⟩ Used in section 962.
 ⟨ Declare the procedure called *insert_relax* 405 ⟩ Used in section 388.
 ⟨ Declare the procedure called *macro_call* 415 ⟩ Used in section 388.
 ⟨ Declare the procedure called *print_cmd_chr* 320 ⟩ Used in section 270.
 ⟨ Declare the procedure called *print_skip_param* 243 ⟩ Used in section 197.
 ⟨ Declare the procedure called *runaway* 328 ⟩ Used in section 137.
 ⟨ Declare the procedure called *show_token_list* 314 ⟩ Used in section 137.
 ⟨ Decry the invalid character and *goto restart* 368 ⟩ Used in section 366.
 ⟨ Delete *c* – "0" tokens and *goto continue* 88 ⟩ Used in section 84.
 ⟨ Delete the page-insertion nodes 1196 ⟩ Used in section 1191.
 ⟨ Destroy the *t* nodes following *q*, and make *r* point to the following node 1059 ⟩ Used in section 1058.

⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common-ending* 840 ⟩ Used in section 833.
 ⟨ Determine horizontal glue stretch setting, then **return** or **goto** *common-ending* 834 ⟩ Used in section 833.
 ⟨ Determine the displacement, *d*, of the left edge of the equation, with respect to the line size *z*, assuming that *l = false* 1380 ⟩ Used in section 1377.
 ⟨ Determine the shrink order 841 ⟩ Used in sections 840, 852, and 972.
 ⟨ Determine the stretch order 835 ⟩ Used in sections 834, 849, and 972.
 ⟨ Determine the value of *height(r)* and the appropriate glue setting; then **return** or **goto** *common-ending* 848 ⟩ Used in section 844.
 ⟨ Determine the value of *width(r)* and the appropriate glue setting; then **return** or **goto** *common-ending* 833 ⟩
 Used in section 823.
 ⟨ Determine vertical glue shrink setting, then **return** or **goto** *common-ending* 852 ⟩ Used in section 848.
 ⟨ Determine vertical glue stretch setting, then **return** or **goto** *common-ending* 849 ⟩ Used in section 848.
 ⟨ Discard erroneous prefixes and **return** 1390 ⟩ Used in section 1389.
 ⟨ Discard the prefixes \long and \outer if they are irrelevant 1391 ⟩ Used in section 1389.
 ⟨ Dispense with trivial cases of void or bad boxes 1155 ⟩ Used in section 1154.
 ⟨ Display *rule spec*_{*i*} for whatsit node created by pdfTeX 1601 ⟩ Used in sections 1603, 1603, and 1603.
 ⟨ Display adjustment *p* 215 ⟩ Used in section 201.
 ⟨ Display box *p* 202 ⟩ Used in section 201.
 ⟨ Display choice node *p* 871 ⟩ Used in section 866.
 ⟨ Display discretionary *p* 213 ⟩ Used in section 201.
 ⟨ Display fraction noad *p* 873 ⟩ Used in section 866.
 ⟨ Display glue *p* 207 ⟩ Used in section 201.
 ⟨ Display if this box is never to be reversed 1704 ⟩ Used in section 202.
 ⟨ Display insertion *p* 206 ⟩ Used in section 201.
 ⟨ Display kern *p* 209 ⟩ Used in section 201.
 ⟨ Display leaders *p* 208 ⟩ Used in section 207.
 ⟨ Display ligature *p* 211 ⟩ Used in section 201.
 ⟨ Display mark *p* 214 ⟩ Used in section 201.
 ⟨ Display math node *p* 210 ⟩ Used in section 201.
 ⟨ Display node *p* 201 ⟩ Used in section 200.
 ⟨ Display normal noad *p* 872 ⟩ Used in section 866.
 ⟨ Display penalty *p* 212 ⟩ Used in section 201.
 ⟨ Display rule *p* 205 ⟩ Used in section 201.
 ⟨ Display special fields of the unset node *p* 203 ⟩ Used in section 202.
 ⟨ Display the current context 334 ⟩ Used in section 333.
 ⟨ Display the insertion split cost 1188 ⟩ Used in section 1187.
 ⟨ Display the page break cost 1183 ⟩ Used in section 1182.
 ⟨ Display the token (*m, c*) 316 ⟩ Used in section 315.
 ⟨ Display the value of *b* 528 ⟩ Used in section 524.
 ⟨ Display the value of *glue_set(p)* 204 ⟩ Used in section 202.
 ⟨ Display the whatsit node *p* 1603 ⟩ Used in section 201.
 ⟨ Display token *p*, and **return** if there are problems 315 ⟩ Used in section 314.
 ⟨ 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 904 ⟩ Used in section 903.
 ⟨ Do ligature or kern command, returning to *main_lig_loop* or *main_loop_wrapup* or *main_loop_move* 1218 ⟩
 Used in section 1216.
 ⟨ Do magic computation 342 ⟩ Used in section 314.
 ⟨ Do some work that has been queued up for \write 1622 ⟩ Used in section 1620.
 ⟨ Do typesetting the DVI commands in virtual character packet 726 ⟩ Used in section 725.
 ⟨ Drop current token and complain that it was unmatched 1244 ⟩ Used in section 1242.
 ⟨ Dump a couple more things and the closing check word 1506 ⟩ Used in section 1480.

⟨ Dump constants for consistency check 1485 ⟩ Used in section 1480.
⟨ Dump pdftex data 1504 ⟩ Used in section 1480.
⟨ Dump regions 1 to 4 of *eqtb* 1493 ⟩ Used in section 1491.
⟨ Dump regions 5 and 6 of *eqtb* 1494 ⟩ Used in section 1491.
⟨ Dump the ε -TEX state 1654, 1758 ⟩ Used in section 1485.
⟨ Dump the array info for internal font number *k* 1500 ⟩ Used in section 1498.
⟨ Dump the dynamic memory 1489 ⟩ Used in section 1480.
⟨ Dump the font information 1498 ⟩ Used in section 1480.
⟨ Dump the hash table 1496 ⟩ Used in section 1491.
⟨ Dump the hyphenation tables 1502 ⟩ Used in section 1480.
⟨ Dump the string pool 1487 ⟩ Used in section 1480.
⟨ Dump the table of equivalents 1491 ⟩ Used in section 1480.
⟨ Either append the insertion node *p* after node *q*, and remove it from the current page, or delete node(*p*) 1199 ⟩ Used in section 1197.
⟨ 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 1197 ⟩ Used in section 1191.
⟨ Either process \ifcase or set *b* to the value of a boolean condition 527 ⟩ Used in section 524.
⟨ Empty the last bytes out of *dvi_buf* 626 ⟩ Used in section 670.
⟨ Enable ε -TEX, if requested 1648 ⟩ Used in section 1517.
⟨ Ensure that box 255 is empty after output 1205 ⟩ Used in section 1203.
⟨ Ensure that box 255 is empty before output 1192 ⟩ Used in section 1191.
⟨ Ensure that *trie_max* $\geq h + 256 1131 ⟩ Used in section 1130.
⟨ Enter a hyphenation exception 1116 ⟩ Used in section 1112.
⟨ Enter all of the patterns into a linked trie, until coming to a right brace 1138 ⟩ Used in section 1137.
⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then return 1112 ⟩
 Used in section 1111.
⟨ Enter *skip_blanks* state, emit a space 371 ⟩ Used in section 369.
⟨ Error handling procedures 78, 81, 82, 93, 94, 95 ⟩ Used in section 4.
⟨ Evaluate the current expression 1792 ⟩ Used in section 1783.
⟨ 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 825 ⟩ Used in section 823.
⟨ Examine node *p* in the vlist, taking account of its effect on the dimensions of the new box; then advance *p* to the next node 845 ⟩ Used in section 844.
⟨ Expand a nonmacro 391 ⟩ Used in section 388.
⟨ Expand macros in the token list and make *link(def_ref)* point to the result 1618 ⟩
 Used in sections 727, 727, 1615, and 1617.
⟨ Expand the next part of the input 504 ⟩ Used in section 503.
⟨ Expand the token after the next token 392 ⟩ Used in section 391.
⟨ Explain that too many dead cycles have occurred in a row 1201 ⟩ Used in section 1189.
⟨ Express astonishment that no number was here 472 ⟩ Used in section 470.
⟨ Express consternation over the fact that no alignment is in progress 1306 ⟩ Used in section 1305.
⟨ Express shock at the missing left brace; goto found 501 ⟩ Used in section 500.
⟨ Feed the macro body and its parameters to the scanner 416 ⟩ Used in section 415.
⟨ Fetch a box dimension 446 ⟩ Used in section 439.
⟨ Fetch a character code from some table 440 ⟩ Used in section 439.
⟨ Fetch a font dimension 451 ⟩ Used in section 439.
⟨ Fetch a font integer 452 ⟩ Used in section 439.
⟨ Fetch a penalties array element 1866 ⟩ Used in section 449.
⟨ Fetch a register 453 ⟩ Used in section 439.
⟨ Fetch a token list or font identifier, provided that *level = tok_val* 441 ⟩ Used in section 439.
⟨ Fetch an internal dimension and goto attach_sign, or fetch an internal integer 475 ⟩ Used in section 474.
⟨ Fetch an item in the current node, if appropriate 450 ⟩ Used in section 439.$

⟨Fetch something on the *page_so_far* 447⟩ Used in section 439.
 ⟨Fetch the *dead_cycles* or the *insert_penalties* 445⟩ Used in section 439.
 ⟨Fetch the *par_shape* size 449⟩ Used in section 439.
 ⟨Fetch the *prev_graf* 448⟩ Used in section 439.
 ⟨Fetch the *space_factor* or the *prev_depth* 444⟩ Used in section 439.
 ⟨Find an active node with fewest demerits 1050⟩ Used in section 1049.
 ⟨Find hyphen locations for the word in *hc*, or **return** 1100⟩ Used in section 1072.
 ⟨Find optimal breakpoints 1039⟩ Used in section 991.
 ⟨Find the best active node for the desired looseness 1051⟩ Used in section 1049.
 ⟨Find the best way to split the insertion, and change *type(r)* to *split_up* 1187⟩ Used in section 1185.
 ⟨Find the glue specification, *main_p*, for text spaces in the current font 1220⟩ Used in sections 1219 and 1221.
 ⟨Finish an alignment in a display 1384⟩ Used in section 988.
 ⟨Finish displayed math 1377⟩ Used in section 1372.
 ⟨Finish issuing a diagnostic message for an overfull or underfull hbox 839⟩ Used in section 823.
 ⟨Finish issuing a diagnostic message for an overfull or underfull vbox 851⟩ Used in section 844.
 ⟨Finish line, emit a \par 373⟩ Used in section 369.
 ⟨Finish line, emit a space 370⟩ Used in section 369.
 ⟨Finish line, **goto** *switch* 372⟩ Used in section 369.
 ⟨Finish math in text 1374⟩ Used in section 1372.
 ⟨Finish shipping 759⟩ Used in section 751.
 ⟨Finish stream of page/form contents 760⟩ Used in section 759.
 ⟨Finish the PDF file 794⟩ Used in section 1513.
 ⟨Finish the DVI file 670⟩ Used in section 1513.
 ⟨Finish the extensions 1626⟩ Used in section 1513.
 ⟨Finish the natural width computation 1735⟩ Used in section 1324.
 ⟨Finish the reversed hlist segment and **goto** *done* 1729⟩ Used in section 1728.
 ⟨Finish *hlist_out* for mixed direction typesetting 1715⟩ Used in sections 647 and 729.
 ⟨Fire up the user's output routine and **return** 1202⟩ Used in section 1189.
 ⟨Fix the reference count, if any, and negate *cur_val* if *negative* 456⟩ Used in section 439.
 ⟨Flush PDF mark lists 765⟩ Used in section 759.
 ⟨Flush resource lists 764⟩ Used in section 759.
 ⟨Flush the box from memory, showing statistics if requested 667⟩ Used in sections 666 and 750.
 ⟨Flush the prototype box 1743⟩ Used in section 1377.
 ⟨Flush *pdf_start_link_node*'s created by *append_link* 783⟩ Used in section 782.
 ⟨Forbidden cases detected in *main_control* 1226, 1276, 1289, 1322⟩ Used in section 1223.
 ⟨Generate ProcSet if desired 768⟩ Used in section 762.
 ⟨Generate XObject resources 767⟩ Used in section 762.
 ⟨Generate a *down* or *right* command for *w* and **return** 637⟩ Used in section 634.
 ⟨Generate a *y0* or *z0* command in order to reuse a previous appearance of *w* 636⟩ Used in section 634.
 ⟨Generate all ε-TEx primitives 1649, 1657, 1663, 1666, 1669, 1672, 1675, 1684, 1686, 1689, 1692, 1697, 1701, 1747, 1759,
 1762, 1770, 1778, 1801, 1805, 1809, 1861, 1864⟩ Used in section 1648.
 ⟨Generate array of annotations or beads in page 771⟩ Used in section 769.
 ⟨Generate font resources 766⟩ Used in section 762.
 ⟨Generate parent pages object 770⟩ Used in section 769.
 ⟨Get ready to compress the trie 1129⟩ Used in section 1143.
 ⟨Get ready to start line breaking 992, 1003, 1010, 1024⟩ Used in section 991.
 ⟨Get the first line of input and prepare to start 1517⟩ Used in section 1512.
 ⟨Get the next non-blank non-call token 432⟩
 Used in sections 431, 467, 481, 529, 552, 604, 1223, 1769, 1784, and 1785.
 ⟨Get the next non-blank non-relax non-call token 430⟩
 Used in sections 429, 1256, 1262, 1329, 1338, 1389, 1404, and 1448.
 ⟨Get the next non-blank non-sign token; set *negative* appropriately 467⟩ Used in sections 466, 474, and 487.

⟨ Get the next token, suppressing expansion 380 ⟩ Used in section 379.
 ⟨ Get user's advice and return 83 ⟩ Used in section 82.
 ⟨ Give diagnostic information, if requested 1208 ⟩ Used in section 1207.
 ⟨ Give improper \hyphenation error 1113 ⟩ Used in section 1112.
 ⟨ Global variables 13, 20, 26, 30, 32, 39, 50, 54, 73, 76, 79, 96, 104, 110, 117, 133, 134, 135, 136, 142, 183, 191, 199, 231, 264, 271, 274, 275, 293, 308, 319, 323, 326, 327, 330, 331, 332, 355, 383, 389, 408, 413, 414, 436, 464, 473, 506, 515, 519, 538, 539, 546, 553, 558, 565, 575, 576, 581, 619, 622, 632, 643, 676, 680, 687, 691, 696, 701, 704, 708, 710, 723, 774, 811, 818, 819, 821, 829, 837, 860, 895, 900, 940, 946, 990, 997, 999, 1001, 1004, 1009, 1015, 1023, 1048, 1069, 1077, 1082, 1084, 1098, 1103, 1120, 1124, 1127, 1148, 1157, 1159, 1166, 1209, 1252, 1444, 1459, 1477, 1483, 1511, 1522, 1525, 1543, 1547, 1550, 1557, 1559, 1570, 1583, 1628, 1633, 1640, 1652, 1660, 1705, 1750, 1773, 1814, 1816, 1835, 1842, 1858, 1859 ⟩
 Used in section 4.
 ⟨ Go into display math mode 1323 ⟩ Used in section 1316.
 ⟨ Go into ordinary math mode 1317 ⟩ Used in sections 1316 and 1320.
 ⟨ Go through the preamble list, determining the column widths and changing the alignrecords to dummy unset boxes 977 ⟩ Used in section 976.
 ⟨ Grow more variable-size memory and goto restart 144 ⟩ Used in section 143.
 ⟨ Handle \readline and goto done 1761 ⟩ Used in section 509.
 ⟨ Handle \unexpanded or \detokenize and return 1688 ⟩ Used in section 491.
 ⟨ Handle a glue node for mixed direction typesetting 1699 ⟩ Used in sections 653, 735, and 1726.
 ⟨ Handle a math node in hlist_out 1716 ⟩ Used in sections 650 and 732.
 ⟨ Handle non-positive logarithm 121 ⟩ Used in section 119.
 ⟨ Handle saved items and goto done 1863 ⟩ Used in section 1288.
 ⟨ Handle situations involving spaces, braces, changes of state 369 ⟩ Used in section 366.
 ⟨ If a line number class has ended, create new active nodes for the best feasible breaks in that class; then return if $r = \text{last_active}$, otherwise compute the new line_width 1011 ⟩ Used in section 1005.
 ⟨ If all characters of the family fit relative to h , then goto found, otherwise goto not_found 1132 ⟩
 Used in section 1130.
 ⟨ If an alignment entry has just ended, take appropriate action 364 ⟩ Used in section 363.
 ⟨ If an expanded code is present, reduce it and goto start_cs 377 ⟩ Used in sections 376 and 378.
 ⟨ If dumping is not allowed, abort 1482 ⟩ Used in section 1480.
 ⟨ If instruction cur_i is a kern with cur_c , attach the kern after q ; or if it is a ligature with cur_c , combine noads q and p appropriately; then return if the cursor has moved past a noad, or goto restart 929 ⟩
 Used in section 928.
 ⟨ If no hyphens were found, return 1079 ⟩ Used in section 1072.
 ⟨ If node cur_p is a legal breakpoint, call try_break; then update the active widths by including the glue in $\text{glue_ptr}(\text{cur_p})$ 1044 ⟩ Used in section 1042.
 ⟨ If node p is a legal breakpoint, check if this break is the best known, and goto done if p is null or if the page-so-far is already too full to accept more stuff 1149 ⟩ Used in section 1147.
 ⟨ 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 ($\text{ord_noad} \dots \text{inner_noad}$), and set pen to the associated penalty 937 ⟩ Used in section 936.
 ⟨ If node r is of type delta_node, update cur_active_width, set prev_r and prev_prev_r, then goto continue 1008 ⟩ Used in section 1005.
 ⟨ If the current list ends with a box node, delete it from the list and make cur_box point to it; otherwise set $\text{cur_box} \leftarrow \text{null}$ 1258 ⟩ Used in section 1257.
 ⟨ If the current page is empty and node p is to be deleted, goto done1; otherwise use node p to update the state of the current page; if this node is an insertion, goto contribute; otherwise if this node is not a legal breakpoint, goto contribute or update_heights; otherwise set pi to the penalty associated with this breakpoint 1177 ⟩ Used in section 1174.
 ⟨ If the cursor is immediately followed by the right boundary, goto reswitch; if it's followed by an invalid character, goto big_switch; otherwise move the cursor one step to the right and goto main_lig_loop 1213 ⟩
 Used in section 1211.

⟨ If the next character is a parameter number, make *cur_tok* a *match* token; but if it is a left brace, store ‘*left_brace*, *end_match*’, set *hash_brace*, and **goto** *done* 502 ⟩ Used in section 500.

⟨ If the preamble list has been traversed, check that the row has ended 968 ⟩ Used in section 967.

⟨ If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1405 ⟩ Used in section 1404.

⟨ If the string *hyph_word*[*h*] is less than *hc*[1 .. *hn*], **goto** *not_found*; but if the two strings are equal, set *hyf* to the hyphen positions and **goto** *found* 1108 ⟩ Used in section 1107.

⟨ If the string *hyph_word*[*h*] is less than or equal to *s*, interchange (*hyph_word*[*h*], *hyph_list*[*h*]) with (*s*, *p*) 1118 ⟩ Used in section 1117.

⟨ If there’s a ligature or kern at the cursor position, update the data structures, possibly advancing *j*; continue until the cursor moves 1086 ⟩ Used in section 1083.

⟨ If there’s a ligature/kern command relevant to *cur_l* and *cur_r*, adjust the text appropriately; exit to *main_loop_wrapup* 1216 ⟩ Used in section 1211.

⟨ If this font has already been loaded, set *f* to the internal font number and **goto** *common_ending* 1438 ⟩ Used in section 1435.

⟨ If this *sup_mark* starts an expanded character like $\wedge\wedge A$ or $\wedge\wedge df$, then **goto** *reswitch*, otherwise set *state* \leftarrow *mid_line* 374 ⟩ Used in section 366.

⟨ If *tmp_k1* is not null then append that kern 1217 ⟩ Used in sections 1211 and 1216.

⟨ Ignore the fraction operation and complain about this ambiguous case 1361 ⟩ Used in section 1359.

⟨ Implement \closeout 1533 ⟩ Used in section 1528.

⟨ Implement \immediate 1623 ⟩ Used in section 1528.

⟨ Implement \openout 1531 ⟩ Used in section 1528.

⟨ Implement \pdfannot 1558 ⟩ Used in section 1528.

⟨ Implement \pdfcatalog 1579 ⟩ Used in section 1528.

⟨ Implement \pdfcolorstack 1539 ⟩ Used in section 1528.

⟨ Implement \pdfdest 1565 ⟩ Used in section 1528.

⟨ Implement \pdfendlink 1561 ⟩ Used in section 1528.

⟨ Implement \pdfendthread 1569 ⟩ Used in section 1528.

⟨ Implement \pdffakespace 1596 ⟩ Used in section 1528.

⟨ Implement \pdffontattr 1589 ⟩ Used in section 1528.

⟨ Implement \pdffontexpand 1535 ⟩ Used in section 1528.

⟨ Implement \pdflglyphunicode 1592 ⟩ Used in section 1528.

⟨ Implement \pdfincludechars 1588 ⟩ Used in section 1528.

⟨ Implement \pdfinfo 1578 ⟩ Used in section 1528.

⟨ Implement \pdfinterwordspaceoff 1595 ⟩ Used in section 1528.

⟨ Implement \pdfinterwordspaceon 1594 ⟩ Used in section 1528.

⟨ Implement \pdfliteral 1538 ⟩ Used in section 1528.

⟨ Implement \pdfmapfile 1590 ⟩ Used in section 1528.

⟨ Implement \pdfmapline 1591 ⟩ Used in section 1528.

⟨ Implement \pdfnames 1580 ⟩ Used in section 1528.

⟨ Implement \pdfnobuiltintounicode 1593 ⟩ Used in section 1528.

⟨ Implement \pdfobj 1544 ⟩ Used in section 1528.

⟨ Implement \pdfoutline 1563 ⟩ Used in section 1528.

⟨ Implement \pdfprimitive 394 ⟩ Used in section 391.

⟨ Implement \pdfrefobj 1546 ⟩ Used in section 1528.

⟨ Implement \pdfrefxform 1549 ⟩ Used in section 1528.

⟨ Implement \pdfrefximage 1554 ⟩ Used in section 1528.

⟨ Implement \pdfresettimer 1586 ⟩ Used in section 1528.

⟨ Implement \pdfrestore 1542 ⟩ Used in section 1528.

⟨ Implement \pdfrunninglinkoff 1597 ⟩ Used in section 1528.

⟨ Implement \pdfrunninglinkon 1598 ⟩ Used in section 1528.

⟨ Implement \pdfsavepos 1576 ⟩ Used in section 1528.

⟨ Implement \pdfsave 1541 ⟩ Used in section 1528.
⟨ Implement \pdfsetmatrix 1540 ⟩ Used in section 1528.
⟨ Implement \pdfsetrandomseed 1585 ⟩ Used in section 1528.
⟨ Implement \pdfsnaprefpoint 1572 ⟩ Used in section 1528.
⟨ Implement \pdfsnappycomp 1575 ⟩ Used in section 1528.
⟨ Implement \pdfsnappy 1574 ⟩ Used in section 1528.
⟨ Implement \pdfspacefont 1599 ⟩ Used in section 1528.
⟨ Implement \pdfstartlink 1560 ⟩ Used in section 1528.
⟨ Implement \pdfstartthread 1568 ⟩ Used in section 1528.
⟨ Implement \pdfthread 1567 ⟩ Used in section 1528.
⟨ Implement \pdftrailerid 1582 ⟩ Used in section 1528.
⟨ Implement \pdftrailer 1581 ⟩ Used in section 1528.
⟨ Implement \pdfxform 1548 ⟩ Used in section 1528.
⟨ Implement \pdfximage 1553 ⟩ Used in section 1528.
⟨ Implement \setlanguage 1625 ⟩ Used in section 1528.
⟨ Implement \special 1534 ⟩ Used in section 1528.
⟨ Implement \write 1532 ⟩ Used in section 1528.
⟨ Incorporate a whatsit node into a vbox 1606 ⟩ Used in section 845.
⟨ Incorporate a whatsit node into an hbox 1607 ⟩ Used in section 825.
⟨ Incorporate box dimensions into the dimensions of the hbox that will contain it 827 ⟩ Used in section 825.
⟨ Incorporate box dimensions into the dimensions of the vbox that will contain it 846 ⟩ Used in section 845.
⟨ Incorporate character dimensions into the dimensions of the hbox that will contain it, then move to the next node 828 ⟩ Used in section 825.
⟨ Incorporate glue into the horizontal totals 832 ⟩ Used in section 825.
⟨ Incorporate glue into the vertical totals 847 ⟩ Used in section 845.
⟨ Increase the number of parameters in the last font 607 ⟩ Used in section 605.
⟨ Increase k until x can be multiplied by a factor of 2^{-k} , and adjust y accordingly 120 ⟩ Used in section 119.
⟨ Initialize additional fields of the first active node 1845 ⟩ Used in section 1040.
⟨ Initialize for hyphenating a paragraph 1068 ⟩ Used in section 1039.
⟨ Initialize table entries (done by INITEX only) 182, 240, 246, 250, 258, 268, 277, 578, 672, 1064, 1123, 1128, 1394, 1479, 1616, 1653, 1818, 1854 ⟩ Used in section 8.
⟨ Initialize the LR stack 1710 ⟩ Used in sections 823, 1714, and 1734.
⟨ Initialize the current page, insert the \topskip glue ahead of p , and goto *continue* 1178 ⟩
Used in section 1177.
⟨ Initialize the input routines 353 ⟩ Used in section 1517.
⟨ Initialize the output routines 55, 61, 554, 559 ⟩ Used in section 1512.
⟨ Initialize the print selector based on *interaction* 75 ⟩ Used in sections 1443 and 1517.
⟨ Initialize the special list heads and constant nodes 966, 973, 996, 1158, 1165 ⟩ Used in section 182.
⟨ Initialize variables as *pdf_ship_out* begins 752 ⟩ Used in section 751.
⟨ Initialize variables as *ship_out* begins 645 ⟩ Used in section 668.
⟨ Initialize variables for PDF output 792 ⟩ Used in section 750.
⟨ Initialize variables for ε-TEx compatibility mode 1812 ⟩ Used in sections 1653 and 1655.
⟨ Initialize variables for ε-TEx extended mode 1813 ⟩ Used in sections 1648 and 1655.
⟨ Initialize whatever TeX might access 8 ⟩ Used in section 4.
⟨ Initialize *hlist_out* for mixed direction typesetting 1714 ⟩ Used in sections 647 and 729.
⟨ Initiate input from new pseudo file 1755 ⟩ Used in section 1753.
⟨ Initiate or terminate input from a file 404 ⟩ Used in section 391.
⟨ Initiate the construction of an hbox or vbox, then return 1261 ⟩ Used in section 1257.
⟨ Input and store tokens from the next line of the file 509 ⟩ Used in section 508.
⟨ Input for \read from the terminal 510 ⟩ Used in section 509.
⟨ Input from external file, goto *restart* if no input found 365 ⟩ Used in section 363.

⟨ Input from token list, **goto** *restart* if end of list or if a parameter needs to be expanded 379 ⟩
 Used in section 363.

⟨ Input the first line of *read_file*[*m*] 511 ⟩ Used in section 509.

⟨ Input the next line of *read_file*[*m*] 512 ⟩ Used in section 509.

⟨ Insert LR nodes at the beginning of the current line and adjust the LR stack based on LR nodes in this line 1707 ⟩ Used in section 1056.

⟨ Insert LR nodes at the end of the current line 1709 ⟩ Used in section 1056.

⟨ Insert a delta node to prepare for breaks at *cur_p* 1019 ⟩ Used in section 1012.

⟨ Insert a delta node to prepare for the next active node 1020 ⟩ Used in section 1012.

⟨ Insert a dummy node to be sub/superscripted 1355 ⟩ Used in section 1354.

⟨ Insert a new active node from *best_place*[*fit_class*] to *cur_p* 1021 ⟩ Used in section 1012.

⟨ Insert a new control sequence after *p*, then make *p* point to it 279 ⟩ Used in section 278.

⟨ Insert a new pattern into the linked trie 1140 ⟩ Used in section 1138.

⟨ Insert a new primitive after *p*, then make *p* point to it 282 ⟩ Used in section 281.

⟨ Insert a new trie node between *q* and *p*, and make *p* point to it 1141 ⟩ Used in sections 1140, 1855, and 1856.

⟨ Insert a token containing *frozen_endv* 401 ⟩ Used in section 388.

⟨ Insert a token saved by \afterassignment, if any 1447 ⟩ Used in section 1389.

⟨ Insert glue for *split_top_skip* and set *p* ← *null* 1146 ⟩ Used in section 1145.

⟨ Insert hyphens as specified in *hyp_list*[*h*] 1109 ⟩ Used in section 1108.

⟨ Insert macro parameter and **goto** *restart* 381 ⟩ Used in section 379.

⟨ Insert the appropriate mark text into the scanner 412 ⟩ Used in section 391.

⟨ Insert the current list into its environment 988 ⟩ Used in section 976.

⟨ Insert the pair (*s*, *p*) into the exception table 1117 ⟩ Used in section 1116.

⟨ Insert the ⟨*v_j*⟩ template and **goto** *restart* 965 ⟩ Used in section 364.

⟨ Insert token *p* into TEX’s input 348 ⟩ Used in section 304.

⟨ Interpret code *c* and **return** if done 84 ⟩ Used in section 83.

⟨ Introduce new material from the terminal and **return** 87 ⟩ Used in section 84.

⟨ Issue an error message if *cur_val* = *fmem_ptr* 606 ⟩ Used in section 605.

⟨ Justify the line ending at breakpoint *cur_p*, and append it to the current vertical list, together with associated penalties and other insertions 1056 ⟩ Used in section 1053.

⟨ Labels in the outer block 6 ⟩ Used in section 4.

⟨ Last-minute procedures 1513, 1515, 1516, 1518 ⟩ Used in section 1510.

⟨ Lengthen the preamble periodically 969 ⟩ Used in section 968.

⟨ Let *cur_h* be the position of the first box, and set *leader_wd* + *lx* to the spacing between corresponding parts of boxes 655 ⟩ Used in sections 654 and 736.

⟨ Let *cur_v* be the position of the first box, and set *leader_ht* + *lx* to the spacing between corresponding parts of boxes 664 ⟩ Used in sections 663 and 745.

⟨ 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* ← *max_dimen* 1325 ⟩ Used in section 1324.

⟨ Let *d* be the natural width of this glue; if stretching or shrinking, set *v* ← *max_dimen*; **goto** *found* in the case of leaders 1326 ⟩ Used in section 1325.

⟨ Let *d* be the width of the whatsit *p* 1608 ⟩ Used in section 1325.

⟨ Let *j* be the prototype box for the display 1740 ⟩ Used in section 1734.

⟨ Let *n* be the largest legal code value, based on *cur_chr* 1411 ⟩ Used in section 1410.

⟨ Link node *p* into the current page and **goto** *done* 1175 ⟩ Used in section 1174.

⟨ Local variables for dimension calculations 476 ⟩ Used in section 474.

⟨ Local variables for finishing a displayed formula 1376, 1741 ⟩ Used in section 1372.

⟨ Local variables for formatting calculations 337 ⟩ Used in section 333.

⟨ Local variables for hyphenation 1078, 1089, 1099, 1106 ⟩ Used in section 1072.

⟨ Local variables for initialization 19, 181, 1104 ⟩ Used in section 4.

⟨ Local variables for line breaking 1038, 1070 ⟩ Used in section 991.

⟨ Look ahead for another character, or leave *lig_stack* empty if there’s none there 1215 ⟩ Used in section 1211.

⟨ Look at all the marks in nodes before the break, and set the final link to *null* at the break 1156 ⟩
 Used in section 1154.

⟨ Look at the list of characters starting with *x* in font *g*; set *f* and *c* whenever a better character is found; **goto** *found* as soon as a large enough variant is encountered 884 ⟩ Used in section 883.

⟨ Look at the other stack entries until deciding what sort of DVI command to generate; **goto** *found* if node *p* is a “hit” 638 ⟩ Used in section 634.

⟨ Look at the variants of (z, x) ; set *f* and *c* whenever a better character is found; **goto** *found* as soon as a large enough variant is encountered 883 ⟩ Used in section 882.

⟨ Look for parameter number or ## 505 ⟩ Used in section 503.

⟨ Look for the word *hc*[1 .. *hn*] in the exception table, and **goto** *found* (with *hyf* containing the hyphens) if an entry is found 1107 ⟩ Used in section 1100.

⟨ Look up the characters of list *n* in the hash table, and set *cur_cs* 1768 ⟩ Used in section 1767.

⟨ Look up the characters of list *r* in the hash table, and set *cur_cs* 400 ⟩ Used in section 398.

⟨ Make a copy of node *p* in node *r* 223 ⟩ Used in section 222.

⟨ Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1212 ⟩
 Used in section 1211.

⟨ 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 1604 ⟩ Used in sections 224 and 1733.

⟨ Make a second pass over the mlist, removing all noads and inserting the proper spacing and penalties 936 ⟩
 Used in section 902.

⟨ Make final adjustments and **goto** *done* 603 ⟩ Used in section 588.

⟨ Make node *p* look like a *char_node* and **goto** *reswitch* 826 ⟩ Used in sections 650, 732, 825, and 1325.

⟨ Make sure that *f* is in the proper range 1790 ⟩ Used in section 1783.

⟨ Make sure that *page_max_depth* is not exceeded 1180 ⟩ Used in section 1174.

⟨ Make sure that *pi* is in the proper range 1007 ⟩ Used in section 1005.

⟨ Make the contribution list empty by setting its tail to *contrib_head* 1172 ⟩ Used in section 1171.

⟨ Make the first 256 strings 48 ⟩ Used in section 47.

⟨ Make the height of box *y* equal to *h* 915 ⟩ Used in section 914.

⟨ Make the running dimensions in rule *q* extend to the boundaries of the alignment 982 ⟩ Used in section 981.

⟨ Make the unset node *r* into a *vlist_node* of height *w*, setting the glue as if the height were *t* 987 ⟩
 Used in section 984.

⟨ Make the unset node *r* into an *hlist_node* of width *w*, setting the glue as if the width were *t* 986 ⟩
 Used in section 984.

⟨ Make variable *b* point to a box for (f, c) 886 ⟩ Used in section 882.

⟨ Manufacture a control sequence name 398 ⟩ Used in section 391.

⟨ Math-only cases in non-math modes, or vice versa 1224 ⟩ Used in section 1223.

⟨ Merge the widths in the span nodes of *q* with those of *p*, destroying the span nodes of *q* 979 ⟩
 Used in section 977.

⟨ 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* 1057 ⟩ Used in section 1056.

⟨ Modify the glue specification in *main_p* according to the space factor 1222 ⟩ Used in section 1221.

⟨ Move down or output leaders 662 ⟩ Used in section 659.

⟨ Move down without outputting leaders 1638 ⟩ Used in section 1637.

⟨ Move node *p* to the current page; if it is time for a page break, put the nodes following the break back onto the contribution list, and **return** to the user’s output routine if there is one 1174 ⟩ Used in section 1171.

⟨ 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 1724 ⟩ Used in section 1723.

⟨ Move pointer *s* to the end of the current list, and set *replace_count(r)* appropriately 1095 ⟩
 Used in section 1091.

⟨ Move right or output leaders 653 ⟩ Used in section 650.

⟨ Move the characters of a ligature node to *hu* and *hc*; but **goto** *done3* if they are not all letters 1075 ⟩
 Used in section 1074.

⟨Move the cursor past a pseudo-ligature, then **goto** *main_loop_looakhead* or *main_lig_loop* 1214⟩
 Used in section 1211.

⟨Move the data into *trie* 1135⟩ Used in section 1143.

⟨Move the non-*char_node* *p* to the new list 1725⟩ Used in section 1724.

⟨Move to next line of file, or **goto** *restart* if there is no next line, or **return** if a \read line has finished 382⟩
 Used in section 365.

⟨Negate a boolean conditional and **goto** *reswitch* 1765⟩ Used in section 391.

⟨Negate all three glue components of *cur_val* 457⟩ Used in sections 456 and 1780.

⟨Nullify *width*(*q*) and the tabskip glue following this column 978⟩ Used in section 977.

⟨Numbered cases for *debug_help* 1519⟩ Used in section 1518.

⟨Open *tfm_file* for input 589⟩ Used in section 588.

⟨Open *vf_file*, return if not found 713⟩ Used in section 712.

⟨Other local variables for *try_break* 1006, 1844⟩ Used in section 1005.

⟨Output PDF outline entries 789⟩ Used in section 788.

⟨Output a Form node in a hlist 1647⟩ Used in section 1645.

⟨Output a Form node in a vlist 1644⟩ Used in section 1639.

⟨Output a Image node in a hlist 1646⟩ Used in section 1645.

⟨Output a Image node in a vlist 1643⟩ Used in section 1639.

⟨Output a box in a vlist 660⟩ Used in section 659.

⟨Output a box in an hlist 651⟩ Used in section 650.

⟨Output a leader box at *cur_h*, then advance *cur_h* by *leader_wd* + *lx* 656⟩ Used in section 654.

⟨Output a leader box at *cur_v*, then advance *cur_v* by *leader_ht* + *lx* 665⟩ Used in section 663.

⟨Output a rule in a vlist, **goto** *next_p* 661⟩ Used in section 659.

⟨Output a rule in an hlist 652⟩ Used in section 650.

⟨Output article threads 790⟩ Used in section 794.

⟨Output fonts definition 801⟩ Used in section 794.

⟨Output leaders in a vlist, **goto** *fin_rule* if a rule or to *next_p* if done 663⟩ Used in section 662.

⟨Output leaders in an hlist, **goto** *fin_rule* if a rule or to *next_p* if done 654⟩ Used in section 653.

⟨Output name tree 804⟩ Used in section 794.

⟨Output node *p* for *hlist_out* and move to the next node, maintaining the condition *cur_v* = *base_line* 648⟩
 Used in section 647.

⟨Output node *p* for *pdf_hlist_out* and move to the next node, maintaining the condition *cur_v* = *base_line* 731⟩
 Used in section 729.

⟨Output node *p* for *pdf_vlist_out* and move to the next node, maintaining the condition *cur_h* = *left_edge* 740⟩
 Used in section 738.

⟨Output node *p* for *vlist_out* and move to the next node, maintaining the condition *cur_h* = *left_edge* 658⟩
 Used in section 657.

⟨Output outlines 788⟩ Used in section 794.

⟨Output pages tree 802⟩ Used in section 794.

⟨Output statistics about this job 1514⟩ Used in section 1513.

⟨Output the catalog object 806⟩ Used in section 794.

⟨Output the cross-reference stream dictionary 814⟩ Used in section 794.

⟨Output the current Pages object in this level 803⟩ Used in section 802.

⟨Output the current node in this level 805⟩ Used in section 804.

⟨Output the font definitions for all fonts that were used 671⟩ Used in section 670.

⟨Output the font name whose internal number is *f* 630⟩ Used in section 629.

⟨Output the non-*char_node* *p* for *hlist_out* and move to the next node 650⟩ Used in section 648.

⟨Output the non-*char_node* *p* for *pdf_hlist_out* and move to the next node 732⟩ Used in section 731.

⟨Output the non-*char_node* *p* for *pdf_vlist_out* 741⟩ Used in section 740.

⟨Output the non-*char_node* *p* for *vlist_out* 659⟩ Used in section 658.

⟨Output the trailer 815⟩ Used in section 794.

⟨Output the whatsit node *p* in a vlist 1613⟩ Used in section 659.

⟨ Output the whatsit node p in an hlist 1614 ⟩ Used in section 650.
 ⟨ Output the whatsit node p in pdf_hlist_out 1645 ⟩ Used in section 732.
 ⟨ Output the whatsit node p in pdf_vlist_out 1639 ⟩ Used in section 741.
 ⟨ Output the obj_tab 813 ⟩ Used in section 794.
 ⟨ Pack all stored $hyph_codes$ 1857 ⟩ Used in section 1143.
 ⟨ Pack the family into $trie$ relative to h 1133 ⟩ Used in section 1130.
 ⟨ Package an unset box for the current column and record its width 972 ⟩ Used in section 967.
 ⟨ Package the display line 1746 ⟩ Used in section 1744.
 ⟨ Package the preamble list, to determine the actual tabskip glue amounts, and let p point to this prototype box 980 ⟩ Used in section 976.
 ⟨ Perform computations for last line and **goto found** 1846 ⟩ Used in section 1028.
 ⟨ Perform the default output routine 1200 ⟩ Used in section 1189.
 ⟨ Pontificate about improper alignment in display 1385 ⟩ Used in section 1384.
 ⟨ Pop the condition stack 522 ⟩ Used in sections 524, 526, 535, and 536.
 ⟨ Pop the expression stack and **goto found** 1789 ⟩ Used in section 1783.
 ⟨ Prepare all the boxes involved in insertions to act as queues 1195 ⟩ Used in section 1191.
 ⟨ Prepare for display after a non-empty paragraph 1734 ⟩ Used in section 1324.
 ⟨ Prepare for display after an empty paragraph 1732 ⟩ Used in section 1323.
 ⟨ Prepare to deactivate node r , and **goto deactivate** unless there is a reason to consider lines of text from r to cur_p 1030 ⟩ Used in section 1027.
 ⟨ Prepare to insert a token that matches cur_group , and print what it is 1243 ⟩ Used in section 1242.
 ⟨ Prepare to move a box or rule node to the current page, then **goto contribute** 1179 ⟩ Used in section 1177.
 ⟨ Prepare to move whatsit p to the current page, then **goto contribute** 1611 ⟩ Used in section 1177.
 ⟨ Print a short indication of the contents of node p 193 ⟩ Used in sections 192 and 674.
 ⟨ Print a symbolic description of the new break node 1022 ⟩ Used in section 1021.
 ⟨ Print a symbolic description of this feasible break 1032 ⟩ Used in section 1031.
 ⟨ Print additional data in the new active node 1852 ⟩ Used in section 1022.
 ⟨ Print additional resources 763 ⟩ Used in section 762.
 ⟨ Print either ‘definition’ or ‘use’ or ‘preamble’ or ‘text’, and insert tokens that should lead to recovery 361 ⟩ Used in section 360.
 ⟨ Print location of current line 335 ⟩ Used in section 334.
 ⟨ Print newly busy locations 189 ⟩ Used in section 185.
 ⟨ Print string s as an error message 1461 ⟩ Used in section 1457.
 ⟨ Print string s on the terminal 1458 ⟩ Used in section 1457.
 ⟨ Print the CreationDate key 809 ⟩ Used in section 807.
 ⟨ Print the ModDate key 810 ⟩ Used in section 807.
 ⟨ Print the Producer key 808 ⟩ Used in section 807.
 ⟨ Print the banner line, including the date and time 562 ⟩ Used in section 560.
 ⟨ Print the help information and **goto continue** 89 ⟩ Used in section 84.
 ⟨ Print the list between $printed_node$ and cur_p , then set $printed_node \leftarrow cur_p$ 1033 ⟩ Used in section 1032.
 ⟨ Print the menu of available options 85 ⟩ Used in section 84.
 ⟨ Print the result of command c 498 ⟩ Used in section 496.
 ⟨ Print two lines using the tricky pseudoprinted information 339 ⟩ Used in section 334.
 ⟨ Print type of token list 336 ⟩ Used in section 334.
 ⟨ Process an active-character control sequence and set $state \leftarrow mid_line$ 375 ⟩ Used in section 366.
 ⟨ Process an expression and **return** 1780 ⟩ Used in section 450.
 ⟨ 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 903 ⟩ Used in section 902.
 ⟨ Process the font definitions 715 ⟩ Used in section 712.
 ⟨ Process the preamble 714 ⟩ Used in section 712.
 ⟨ Process whatsit p in $vert_break$ loop, **goto not_found** 1612 ⟩ Used in section 1150.

⟨ Prune the current list, if necessary, until it contains only *char_node*, *kern_node*, *hlist_node*, *vlist_node*, *rule_node*, and *ligature_node* items; set *n* to the length of the list, and set *q* to the list's tail 1299 ⟩ Used in section 1297.

⟨ Prune unwanted nodes at the beginning of the next line 1055 ⟩ Used in section 1053.

⟨ Pseudoprint the line 340 ⟩ Used in section 334.

⟨ Pseudoprint the token list 341 ⟩ Used in section 334.

⟨ Push the condition stack 521 ⟩ Used in section 524.

⟨ Push the expression stack and goto *restart* 1788 ⟩ Used in section 1785.

⟨ Put each of TeX's primitives into the hash table 244, 248, 256, 266, 287, 356, 402, 410, 437, 442, 494, 513, 517, 579, 956, 1160, 1230, 1236, 1249, 1266, 1285, 1292, 1319, 1334, 1347, 1356, 1366, 1386, 1397, 1400, 1408, 1428, 1432, 1440, 1450, 1455, 1464, 1469, 1524 ⟩ Used in section 1516.

⟨ Put help message on the transcript file 90 ⟩ Used in section 82.

⟨ Put the characters *hu*[*i* + 1 ..] into *post_break*(*r*), appending to this list and to *major_tail* until synchronization has been achieved 1093 ⟩ Used in section 1091.

⟨ Put the characters *hu*[*l* .. *i*] and a hyphen into *pre_break*(*r*) 1092 ⟩ Used in section 1091.

⟨ Put the fraction into a box with its delimiters, and make *new_hlist*(*q*) point to it 924 ⟩ Used in section 919.

⟨ Put the \leftskip glue at the left and detach this line 1063 ⟩ Used in section 1056.

⟨ 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 1191 ⟩ Used in section 1189.

⟨ Put the (positive) 'at' size into *s* 1437 ⟩ Used in section 1436.

⟨ Put the \rightskip glue after node *q* 1062 ⟩ Used in section 1057.

⟨ Read and check the font data; abort if the TFM file is malformed; if there's no room for this font, say so and goto *done*; otherwise incr(*font_ptr*) and goto *done* 588 ⟩ Used in section 586.

⟨ Read box dimensions 598 ⟩ Used in section 588.

⟨ Read character data 595 ⟩ Used in section 588.

⟨ Read extensible character recipes 601 ⟩ Used in section 588.

⟨ Read font parameters 602 ⟩ Used in section 588.

⟨ Read ligature/kern program 600 ⟩ Used in section 588.

⟨ Read next line of file into *buffer*, or goto *restart* if the file has ended 384 ⟩ Used in section 382.

⟨ Read one string, but return false if the string memory space is getting too tight for comfort 52 ⟩ Used in section 51.

⟨ Read the first line of the new file 564 ⟩ Used in section 563.

⟨ Read the other strings from the TEX.POOL file and return true, or give an error message and return false 51 ⟩ Used in section 47.

⟨ Read the TFM header 594 ⟩ Used in section 588.

⟨ Read the TFM size fields 591 ⟩ Used in section 588.

⟨ Readjust the height and depth of *cur_box*, for \vtop 1265 ⟩ Used in section 1264.

⟨ Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 1090 ⟩ Used in section 1080.

⟨ Record a new feasible break 1031 ⟩ Used in section 1027.

⟨ Recover from an unbalanced output routine 1204 ⟩ Used in section 1203.

⟨ Recover from an unbalanced write command 1619 ⟩ Used in section 1618.

⟨ Recycle node *p* 1176 ⟩ Used in section 1174.

⟨ Reduce to the case that $a, c \geq 0, b, d > 0$ 123 ⟩ Used in section 122.

⟨ Reduce to the case that $f \geq 0$ and $q > 0$ 115 ⟩ Used in section 114.

⟨ Remove the last box, unless it's part of a discretionary 1259 ⟩ Used in section 1258.

⟨ Replace nodes *ha* .. *hb* by a sequence of nodes that includes the discretionary hyphens 1080 ⟩ Used in section 1072.

⟨ Replace the tail of the list by *p* 1365 ⟩ Used in section 1364.

⟨ Replace *z* by *z'* and compute α, β 599 ⟩ Used in section 598.

⟨ Report LR problems 1713 ⟩ Used in sections 1712 and 1730.

⟨ Report a runaway argument and abort 422 ⟩ Used in sections 418 and 425.

⟨ Report a tight hbox and goto *common_ending*, if this box is sufficiently bad 843 ⟩ Used in section 840.

⟨ Report a tight vbox and **goto** *common-ending*, if this box is sufficiently bad 854 ⟩ Used in section 852.
 ⟨ Report an extra right brace and **goto** *continue* 421 ⟩ Used in section 418.
 ⟨ Report an improper use of the macro and **abort** 424 ⟩ Used in section 423.
 ⟨ Report an overfull hbox and **goto** *common-ending*, if this box is sufficiently bad 842 ⟩ Used in section 840.
 ⟨ Report an overfull vbox and **goto** *common-ending*, if this box is sufficiently bad 853 ⟩ Used in section 852.
 ⟨ Report an underfull hbox and **goto** *common-ending*, if this box is sufficiently bad 836 ⟩ Used in section 834.
 ⟨ Report an underfull vbox and **goto** *common-ending*, if this box is sufficiently bad 850 ⟩ Used in section 849.
 ⟨ Report overflow of the input buffer, and **abort** 35 ⟩ Used in sections 31 and 1756.
 ⟨ Report that an invalid delimiter code is being changed to null; set *cur_val* \leftarrow 0 1339 ⟩ Used in section 1338.
 ⟨ Report that the font won't be loaded 587 ⟩ Used in section 586.
 ⟨ Report that this dimension is out of range 486 ⟩ Used in section 474.
 ⟨ Reset PDF mark lists 754 ⟩ Used in section 752.
 ⟨ Reset resource lists 753 ⟩ Used in sections 752 and 775.
 ⟨ Reset *cur_tok* for unexpandable primitives, **goto** *restart* 395 ⟩ Used in sections 439 and 466.
 ⟨ Restore resource lists 777 ⟩ Used in section 775.
 ⟨ Resume the page builder after an output routine has come to an end 1203 ⟩ Used in section 1278.
 ⟨ Retrieve the prototype box 1742 ⟩ Used in sections 1372 and 1372.
 ⟨ Reverse an hlist segment and **goto** *reswitch* 1722 ⟩ Used in section 1717.
 ⟨ Reverse the complete hlist and set the subtype to *reversed* 1721 ⟩ Used in section 1714.
 ⟨ Reverse the linked list of Page and Pages objects 800 ⟩ Used in section 794.
 ⟨ Reverse the links of the relevant passive nodes, setting *cur_p* to the first breakpoint 1054 ⟩
 Used in section 1053.
 ⟨ Save current position in DVI mode 1621 ⟩ Used in section 1620.
 ⟨ Save current position to *pdf_last_x_pos*, *pdf_last_y_pos* 1641 ⟩ Used in sections 1639 and 1645.
 ⟨ Save current position to *pdf_snapx_refpos*, *pdf_snapy_refpos* 1642 ⟩ Used in sections 1639 and 1645.
 ⟨ Save resource lists 776 ⟩ Used in section 775.
 ⟨ Scan a control sequence and set *state* \leftarrow *skip_blanks* or *mid_line* 376 ⟩ Used in section 366.
 ⟨ Scan a factor *f* of type *o* or start a subexpression 1785 ⟩ Used in section 1783.
 ⟨ Scan a numeric constant 470 ⟩ Used in section 466.
 ⟨ Scan a parameter until its delimiter string has been found; or, if *s* = *null*, simply scan the delimiter
 string 418 ⟩ Used in section 417.
 ⟨ Scan a subformula enclosed in braces and **return** 1331 ⟩ Used in section 1329.
 ⟨ Scan ahead in the buffer until finding a nonletter; if an expanded code is encountered, reduce it and
 goto *start_cs*; otherwise if a multiletter control sequence is found, adjust *cur_cs* and *loc*, and **goto**
 found 378 ⟩ Used in section 376.
 ⟨ Scan an alphabetic character code into *cur_val* 468 ⟩ Used in section 466.
 ⟨ Scan an optional space 469 ⟩ Used in sections 468, 474, 481, 705, 1378, 1544, 1556, 1556, 1558, and 1565.
 ⟨ Scan and build the body of the token list; **goto** *found* when finished 503 ⟩ Used in section 499.
 ⟨ Scan and build the parameter part of the macro definition 500 ⟩ Used in section 499.
 ⟨ Scan and evaluate an expression *e* of type *l* 1783 ⟩ Used in section 1782.
 ⟨ Scan decimal fraction 478 ⟩ Used in section 474.
 ⟨ Scan file name in the buffer 557 ⟩ Used in section 556.
 ⟨ Scan for all other units and adjust *cur_val* and *f* accordingly; **goto** *done* in the case of scaled points 484 ⟩
 Used in section 479.
 ⟨ Scan for *fil* units; **goto** *attach_fraction* if found 480 ⟩ Used in section 479.
 ⟨ Scan for *mu* units and **goto** *attach_fraction* 482 ⟩ Used in section 479.
 ⟨ Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 481 ⟩
 Used in section 479.
 ⟨ Scan preamble text until *cur_cmd* is *tab_mark* or *car_ret*, looking for changes in the tabskip glue; append
 an alignrecord to the preamble list 955 ⟩ Used in section 953.
 ⟨ Scan the argument for command *c* 497 ⟩ Used in section 496.
 ⟨ Scan the font size specification 1436 ⟩ Used in section 1435.

- ⟨ Scan the next operator and set *o* 1784 ⟩ Used in section 1783.
- ⟨ Scan the parameters and make *link(r)* point to the macro body; but **return** if an illegal \par is detected 417 ⟩ Used in section 415.
- ⟨ Scan the preamble and record it in the *preamble* list 953 ⟩ Used in section 950.
- ⟨ Scan the template $\langle u_j \rangle$, putting the resulting token list in *hold_head* 959 ⟩ Used in section 955.
- ⟨ Scan the template $\langle v_j \rangle$, putting the resulting token list in *hold_head* 960 ⟩ Used in section 955.
- ⟨ Scan units and set *cur_val* to $x \cdot (cur_val + f/2^{16})$, where there are *x* sp per unit; **goto attach_sign** if the units are internal 479 ⟩ Used in section 474.
- ⟨ Search *eqtb* for equivalents equal to *p* 273 ⟩ Used in section 190.
- ⟨ Search *hyph_list* for pointers to *p* 1110 ⟩ Used in section 190.
- ⟨ Search *save_stack* for equivalents that point to *p* 307 ⟩ Used in section 190.
- ⟨ Select the appropriate case and **return** or **goto common-ending** 535 ⟩ Used in section 527.
- ⟨ Set initial values of key variables 21, 23, 24, 74, 77, 80, 97, 118, 184, 233, 272, 276, 294, 309, 390, 409, 465, 507, 516, 547, 577, 582, 620, 623, 633, 677, 681, 688, 697, 709, 711, 724, 820, 830, 838, 861, 947, 1105, 1167, 1210, 1445, 1460, 1478, 1523, 1551, 1571, 1584, 1629, 1634, 1706, 1751, 1817, 1836, 1860 ⟩ Used in section 8.
- ⟨ Set line length parameters in preparation for hanging indentation 1025 ⟩ Used in section 1024.
- ⟨ Set the glue in all the unset boxes of the current list 981 ⟩ Used in section 976.
- ⟨ Set the glue in node *r* and change it from an unset node 984 ⟩ Used in section 983.
- ⟨ Set the unset box *q* and the unset boxes in it 983 ⟩ Used in section 981.
- ⟨ Set the value of *b* to the badness for shrinking the line, and compute the corresponding *fit_class* 1029 ⟩ Used in section 1027.
- ⟨ Set the value of *b* to the badness for stretching the line, and compute the corresponding *fit_class* 1028 ⟩ Used in section 1027.
- ⟨ Set the value of *b* to the badness of the last line for shrinking, compute the corresponding *fit_class*, and **goto found** 1848 ⟩ Used in section 1846.
- ⟨ Set the value of *b* to the badness of the last line for stretching, compute the corresponding *fit_class*, and **goto found** 1847 ⟩ Used in section 1846.
- ⟨ Set the value of *output_penalty* 1190 ⟩ Used in section 1189.
- ⟨ Set the value of *x* to the text direction before the display 1731 ⟩ Used in sections 1732 and 1734.
- ⟨ Set up data structures with the cursor following position *j* 1085 ⟩ Used in section 1083.
- ⟨ Set up the hlist for the display line 1745 ⟩ Used in section 1744.
- ⟨ Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 879 ⟩ Used in sections 896, 902, 903, 906, 930, 936, 938, and 939.
- ⟨ Set variable *c* to the current escape character 261 ⟩ Used in section 63.
- ⟨ Set variable *w* to indicate if this case should be reported 1775 ⟩ Used in sections 1774 and 1776.
- ⟨ Ship box *p* out 668 ⟩ Used in section 666.
- ⟨ Show equivalent *n*, in region 1 or 2 241 ⟩ Used in section 270.
- ⟨ Show equivalent *n*, in region 3 247 ⟩ Used in section 270.
- ⟨ Show equivalent *n*, in region 4 251 ⟩ Used in section 270.
- ⟨ Show equivalent *n*, in region 5 260 ⟩ Used in section 270.
- ⟨ Show equivalent *n*, in region 6 269 ⟩ Used in section 270.
- ⟨ Show the auxiliary field, *a* 237 ⟩ Used in section 236.
- ⟨ Show the box context 1681 ⟩ Used in section 1679.
- ⟨ Show the box packaging info 1680 ⟩ Used in section 1679.
- ⟨ Show the current contents of a box 1474 ⟩ Used in section 1471.
- ⟨ Show the current meaning of a token, then **goto common-ending** 1472 ⟩ Used in section 1471.
- ⟨ Show the current value of some parameter or register, then **goto common-ending** 1475 ⟩ Used in section 1471.
- ⟨ Show the font identifier in *eqtb[n]* 252 ⟩ Used in section 251.
- ⟨ Show the halfword code in *eqtb[n]* 253 ⟩ Used in section 251.
- ⟨ Show the status of the current page 1163 ⟩ Used in section 236.
- ⟨ Show the text of the macro being expanded 427 ⟩ Used in section 415.

⟨ Simplify a trivial box 897 ⟩ Used in section 896.
 ⟨ Skip to \else or \fi, then goto common_ending 526 ⟩ Used in section 524.
 ⟨ Skip to node ha, or goto done1 if no hyphenation should be attempted 1073 ⟩ Used in section 1071.
 ⟨ Skip to node hb, putting letters into hu and hc 1074 ⟩ Used in section 1071.
 ⟨ Sort p into the list starting at rover and advance p to rlink(p) 150 ⟩ Used in section 149.
 ⟨ Sort the hyphenation op tables into proper order 1122 ⟩ Used in section 1129.
 ⟨ Split off part of a vertical box, make cur_box point to it 1260 ⟩ Used in section 1257.
 ⟨ Squeeze the equation as much as possible; if there is an equation number that should go on a separate line by itself, set $e \leftarrow 0$ 1379 ⟩ Used in section 1377.
 ⟨ Start a new current page 1168 ⟩ Used in sections 233 and 1194.
 ⟨ Start stream of page/form contents 757 ⟩ Used in section 752.
 ⟨ Store additional data for this feasible break 1850 ⟩ Used in section 1031.
 ⟨ Store additional data in the new active node 1851 ⟩ Used in section 1021.
 ⟨ Store cur_box in a box register 1255 ⟩ Used in section 1253.
 ⟨ Store maximum values in the hyf table 1101 ⟩ Used in section 1100.
 ⟨ Store save_stack[save_ptr] in eqtb[p], unless eqtb[p] holds a global value 305 ⟩ Used in section 304.
 ⟨ Store all current lc_code values 1856 ⟩ Used in section 1855.
 ⟨ Store hyphenation codes for current language 1855 ⟩ Used in section 1137.
 ⟨ Store the current token, but goto continue if it is a blank space that would become an undelimited parameter 419 ⟩ Used in section 418.
 ⟨ Store the packet being built 718 ⟩ Used in section 717.
 ⟨ Subtract glue from break_width 1014 ⟩ Used in section 1013.
 ⟨ Subtract the width of node v from break_width 1017 ⟩ Used in section 1016.
 ⟨ Suppress expansion of the next token 393 ⟩ Used in section 391.
 ⟨ Swap the subscript and superscript into box x 918 ⟩ Used in section 914.
 ⟨ Switch to a larger accent if available and appropriate 916 ⟩ Used in section 914.
 ⟨ Tell the user what has run away and try to recover 360 ⟩ Used in section 358.
 ⟨ Terminate the current conditional and skip to \fi 536 ⟩ Used in section 391.
 ⟨ Test box register status 531 ⟩ Used in section 527.
 ⟨ Test if an integer is odd 530 ⟩ Used in section 527.
 ⟨ Test if two characters match 532 ⟩ Used in section 527.
 ⟨ Test if two macro texts match 534 ⟩ Used in section 533.
 ⟨ Test if two tokens match 533 ⟩ Used in section 527.
 ⟨ Test relation between integers or dimensions 529 ⟩ Used in section 527.
 ⟨ The em width for cur_font 584 ⟩ Used in section 481.
 ⟨ The x-height for cur_font 585 ⟩ Used in section 481.
 ⟨ Tidy up the parameter just scanned, and tuck it away 426 ⟩ Used in section 418.
 ⟨ Transfer node p to the adjustment list 831 ⟩ Used in section 825.
 ⟨ Transplant the post-break list 1060 ⟩ Used in section 1058.
 ⟨ Transplant the pre-break list 1061 ⟩ Used in section 1058.
 ⟨ Treat cur_chr as an active character 1330 ⟩ Used in sections 1329 and 1333.
 ⟨ Try the final line break at the end of the paragraph, and goto done if the desired breakpoints have been found 1049 ⟩ Used in section 1039.
 ⟨ Try to allocate within node p and its physical successors, and goto found if allocation was possible 145 ⟩ Used in section 143.
 ⟨ Try to break after a discretionary fragment, then goto done5 1045 ⟩ Used in section 1042.
 ⟨ Try to get a different log file name 561 ⟩ Used in section 560.
 ⟨ Try to hyphenate the following word 1071 ⟩ Used in section 1042.
 ⟨ Try to recover from mismatched \right 1370 ⟩ Used in section 1369.
 ⟨ Types in the outer block 18, 25, 38, 101, 109, 131, 168, 230, 291, 322, 574, 621, 694, 707, 722, 1097, 1102, 1627, 1632, 1678 ⟩ Used in section 4.
 ⟨ Undump a couple more things and the closing check word 1507 ⟩ Used in section 1481.

⟨ Undump constants for consistency check 1486 ⟩ Used in section 1481.
⟨ Undump pdftex data 1505 ⟩ Used in section 1481.
⟨ Undump regions 1 to 6 of *eqtb* 1495 ⟩ Used in section 1492.
⟨ Undump the ε -TEX state 1655 ⟩ Used in section 1486.
⟨ Undump the array info for internal font number *k* 1501 ⟩ Used in section 1499.
⟨ Undump the dynamic memory 1490 ⟩ Used in section 1481.
⟨ Undump the font information 1499 ⟩ Used in section 1481.
⟨ Undump the hash table 1497 ⟩ Used in section 1492.
⟨ Undump the hyphenation tables 1503 ⟩ Used in section 1481.
⟨ Undump the string pool 1488 ⟩ Used in section 1481.
⟨ Undump the table of equivalents 1492 ⟩ Used in section 1481.
⟨ Update the active widths, since the first active node has been deleted 1037 ⟩ Used in section 1036.
⟨ Update the current height and depth measurements with respect to a glue or kern node *p* 1153 ⟩
 Used in section 1149.
⟨ Update the current marks for *fire_up* 1830 ⟩ Used in section 1191.
⟨ Update the current marks for *vsplit* 1827 ⟩ Used in section 1156.
⟨ Update the current page measurements with respect to the glue or kern specified by node *p* 1181 ⟩
 Used in section 1174.
⟨ Update the value of *printed_node* for symbolic displays 1034 ⟩ Used in section 1005.
⟨ Update the values of *first_mark* and *bot_mark* 1193 ⟩ Used in section 1191.
⟨ Update the values of *last_glue*, *last_penalty*, and *last_kern* 1173 ⟩ Used in section 1171.
⟨ Update the values of *max_h* and *max_v*; but if the page is too large, **goto done** 669 ⟩
 Used in sections 668 and 751.
⟨ Update width entry for spanned columns 974 ⟩ Used in section 972.
⟨ Use code *c* to distinguish between generalized fractions 1360 ⟩ Used in section 1359.
⟨ Use node *p* to update the current height and depth measurements; if this node is not a legal breakpoint,
 goto not_found or *update_heights*, otherwise set *pi* to the associated penalty at the break 1150 ⟩
 Used in section 1149.
⟨ Use size fields to allocate font information 592 ⟩ Used in section 588.
⟨ Wipe out the whatsit node *p* and **goto done** 1605 ⟩ Used in section 220.
⟨ Wrap up the box specified by node *r*, splitting node *p* if called for; set *wait* \leftarrow true if node *p* holds a
 remainder after splitting 1198 ⟩ Used in section 1197.
⟨ Write out Form stream header 756 ⟩ Used in section 752.
⟨ Write out PDF annotations 781 ⟩ Used in section 780.
⟨ Write out PDF bead rectangle specifications 786 ⟩ Used in section 780.
⟨ Write out PDF link annotations 782 ⟩ Used in section 780.
⟨ Write out PDF mark destinations 784 ⟩ Used in section 780.
⟨ Write out page object 769 ⟩ Used in section 759.
⟨ Write out pending PDF marks 780 ⟩ Used in section 759.
⟨ Write out pending forms 775 ⟩ Used in section 761.
⟨ Write out pending images 779 ⟩ Used in section 761.
⟨ Write out pending raw objects 773 ⟩ Used in section 761.
⟨ Write out resource lists 761 ⟩ Used in section 759.
⟨ Write out resources dictionary 762 ⟩ Used in section 759.

	Section	Page
1. Introduction	1	3
2. The character set	17	11
3. Input and output	25	14
4. String handling	38	20
5. On-line and off-line printing	54	25
6. Reporting errors	72	31
7. Arithmetic with scaled dimensions	99	39
7b. Random numbers	110	43
8. Packed data	128	50
9. Dynamic memory allocation	133	52
10. Data structures for boxes and their friends	151	58
11. Memory layout	180	67
12. Displaying boxes	191	71
13. Destroying boxes	217	79
14. Copying boxes	221	81
15. The command codes	225	83
16. The semantic nest	229	87
17. The table of equivalents	238	92
18. The hash table	274	118
19. Saving and restoring equivalents	290	127
20. Token lists	311	134
21. Introduction to the syntactic routines	319	138
22. Input stacks and states	322	141
23. Maintaining the input stacks	343	151
24. Getting the next token	354	154
25. Expanding the next token	388	165
26. Basic scanning subroutines	428	177
27. Building token lists	490	200
28. Conditional processing	513	215
29. File names	537	223
30. Font metric data	565	232
31. Device-independent file format	610	251
32. Shipping pages out	619	257
32a. pdfTEX basic	672	277
32b. pdfTEX output low-level subroutines	679	281
32c. PDF page description	691	291
32d. The cross-reference table	694	298
32e. Font processing	703	311
32f. PDF shipping out	727	330
33. Packaging	816	362
34. Data structures for math mode	856	378
35. Subroutines for math mode	875	387
36. Typesetting math formulas	895	394
37. Alignment	944	414
38. Breaking paragraphs into lines	989	431
39. Breaking paragraphs into lines, continued	1038	454
40. Pre-hyphenation	1068	468
41. Post-hyphenation	1077	472
42. Hyphenation	1096	482
43. Initializing the hyphenation tables	1119	488
44. Breaking vertical lists into pages	1144	498
45. The page builder	1157	504

46. The chief executive	1206	521
47. Building boxes and lists	1233	533
48. Building math lists	1314	556
49. Mode-independent processing	1386	575
50. Dumping and undumping the tables	1477	598
51. The main program	1510	609
52. Debugging	1518	615
53. Extensions	1520	617
53a. The extended features of ε -TeX	1648	673
54. System-dependent changes	1867	738
55. Index	1868	739