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

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

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

2. The present implementation has a long ancestry, beginning in the summer of 1977, when Michael F. Plass and Frank M. Liang designed and coded a prototype based on some specifications that the author had made in May of that year. This original protoTEX included macro definitions and elementary manipulations on boxes and glue, but it did not have line-breaking, page-breaking, mathematical formulas, alignment routines, error recovery, or the present semantic nest; furthermore, it used character lists instead of token lists, so that a control sequence like `\halign` was represented by a list of seven characters. A complete version of TEX was designed and coded by the author in late 1977 and early 1978; that program, like its prototype, was written in the SAIL language, for which an excellent debugging system was available. Preliminary plans to convert the SAIL code into a form somewhat like the present "web" were developed by Luis Trabb Pardo and the author at the beginning of 1979, and a complete implementation was created by Ignacio A. Zabala in 1979 and 1980. The TEX82 program, which was written by the author during the latter part of 1981 and the early part of 1982, also incorporates ideas from the 1979 implementation of TEX in MESA that was written by Leonidas Guibas, Robert Sedgewick, and Douglas Wyatt at the Xerox Palo Alto Research Center. Several hundred refinements were introduced into TEX82 based on the experiences gained with the original implementations, so that essentially every part of the system has been substantially improved. After the appearance of "Version 0" in September 1982, this program benefited greatly from the comments of many other people, notably David R. Fuchs and Howard W. Trickey. A final revision in September 1989 extended the input character set to eight-bit codes and introduced the ability to hyphenate words from different languages, based on some ideas of Michael J. Ferguson.

No doubt there still is plenty of room for improvement, but the author is firmly committed to keeping TEX82 "frozen" from now on; stability and reliability are to be its main virtues.

On the other hand, the `WEB` description can be extended without changing the core of TEX82 itself, and the program has been designed so that such extensions are not extremely difficult to make. The *banner* string defined here should be changed whenever TEX undergoes any modifications, so that it will be clear which version of TEX might be the guilty party when a problem arises.

If this program is changed, the resulting system should not be called '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].

MLTEX will add new primitives changing the behaviour of TEX. The *banner* string has to be changed. We do not change the *banner* string, but will output an additional line to make clear that this is a modified TEX version.

define *TeX_banner_k* ≡ ´`This␣is␣TeXk,␣Version␣3.1415926`´ { printed when TEX starts }
define *TeX_banner* ≡ ´`This␣is␣TeX,␣Version␣3.1415926`´ { printed when TEX starts }

define *PUTeX_version_string* ≡ ´`-4.0`´ { current PUTEX version }

define *PUTeX_banner* ≡ ´`This␣is␣PUTeX,␣Version␣3.1415926`´, *PUTeX_version_string*
define *PUTeX_banner_k* ≡ *PUTeX_banner* { printed when PUTEX starts }

define *banner* ≡ *PUTeX_banner*
define *banner_k* ≡ *PUTeX_banner_k*

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

```
  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 }
  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 ⟩
  ⟨ PUTeX routines that will be used by TeX routines 1413 ⟩
      { end – putex }
```

5. The overall TEX 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 TEX'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. For Web2c, labels are not declared in the main program, but we still have to declare the symbolic names.

define *start_of_TEX* = 1 { go here when TEX's variables are initialized }
define *final_end* = 9999 { this label marks the ending of the program }

7. Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when TEX is being installed or when system wizards are fooling around with TEX 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 TEX's memory usage. The **stat** ... **tats** code also implements diagnostic information for `\tracingparagraphs` and `\tracingpages`.

define *debug* ≡ *ifdef* (´`TEXMF_DEBUG`´)
define *gubed* ≡ *endif* (´`TEXMF_DEBUG`´)
format *debug* ≡ *begin*
format *gubed* ≡ *end*

define *stat* ≡ *ifdef* (´`STAT`´)
define *tats* ≡ *endif* (´`STAT`´)
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 TEX'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**' for declarations and by the codewords '**Init** ... **Tini**' for executable code. This distinction is helpful for implementations where a run-time switch differentiates between the two versions of the program.

define *init* ≡ *ifdef* (´`INITEX`´)
define *tini* ≡ *endif* (´`INITEX`´)
define *Init* ≡
  init
  if *ini_version* **then**
   begin
define *Tini* ≡
  end ; **tini**
format *Init* ≡ *begin*
format *Tini* ≡ *end*
format *init* ≡ *begin*
format *tini* ≡ *end*

⟨ Initialize whatever TEX might access 8 ⟩ ≡
⟨ Set initial values of key variables 21 ⟩
Init ⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ **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 TEX 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 TEX 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 $x = 3$ ⟩;
othercases ⟨ code for $x \neq 1$ and $x \neq 3$ ⟩
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 TEX 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.

define *file_name_size* ≡ *maxint*
define *ssup_error_line* = 255
define *ssup_max_strings* ≡ 2097151
{ Larger values than 65536 cause the arrays to consume much more memory. }
define *ssup_trie_opcode* ≡ 65535
define *ssup_trie_size* ≡ `"3FFFFF`
define *ssup_hyph_size* ≡ 65535 { Changing this requires changing (un)dumping! }
define *iinf_hyphen_size* ≡ 610 { Must be not less than *hyph_prime*! }
define *max_font_max* = 9000 { maximum number of internal fonts; this can be increased, but *hash_size* + *max_font_max* should not exceed 29000. }
define *font_base* = 0 { smallest internal font number; must be ≥ *min_quarterword*; do not change this without modifying the dynamic definition of the font arrays. }

⟨ Constants in the outer block 11 ⟩ ≡
hash_offset = 514; { smallest index in hash array, i.e., *hash_base* }
{ Use *hash_offset* = 0 for compilers which cannot decrement pointers. }
trie_op_size = 35111;
{ space for "opcodes" in the hyphenation patterns; best if relatively prime to 313, 361, and 1009. }
neg_trie_op_size = −35111; { for lower *trie_op_hash* array bound; must be equal to −*trie_op_size*. }
min_trie_op = 0; { first possible trie op code for any language }
max_trie_op = *ssup_trie_opcode*; { largest possible trie opcode for any language }
pool_name = *TEXMF_POOL_NAME*; { this is configurable, for the sake of ML-TeX }
{ string of length *file_name_size*; tells where the string pool appears }
engine_name = *TEXMF_ENGINE_NAME*; { the name of this engine }
inf_mem_bot = 0; *sup_mem_bot* = 1; *inf_main_memory* = 3000; *sup_main_memory* = 256000000;
inf_trie_size = 8000; *sup_trie_size* = *ssup_trie_size*; *inf_max_strings* = 3000;
sup_max_strings = *ssup_max_strings*; *inf_strings_free* = 100; *sup_strings_free* = *sup_max_strings*;
inf_buf_size = 500; *sup_buf_size* = 30000000; *inf_nest_size* = 40; *sup_nest_size* = 4000;
inf_max_in_open = 6; *sup_max_in_open* = 127; *inf_param_size* = 60; *sup_param_size* = 32767;
inf_save_size = 600; *sup_save_size* = 80000; *inf_stack_size* = 200; *sup_stack_size* = 30000;
inf_dvi_buf_size = 800; *sup_dvi_buf_size* = 65536; *inf_font_mem_size* = 20000;
sup_font_mem_size = 147483647; { *integer*-limited, so 2 could be prepended? }
sup_font_max = *max_font_max*; *inf_font_max* = 50; { could be smaller, but why? }
inf_pool_size = 32000; *sup_pool_size* = 40000000; *inf_pool_free* = 1000; *sup_pool_free* = *sup_pool_size*;
inf_string_vacancies = 8000; *sup_string_vacancies* = *sup_pool_size* − 23000;
sup_hash_extra = *sup_max_strings*; *inf_hash_extra* = 0; *sup_hyph_size* = *ssup_hyph_size*;
inf_hyph_size = *iinf_hyphen_size*; { Must be not less than *hyph_prime*! }
inf_expand_depth = 10; *sup_expand_depth* = 10000000;

See also sections 1472 and 1499.

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 TEX's capacity. But if they are changed, it is necessary to rerun the initialization program `INITEX` to generate new tables for the production TEX 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 $hash\_size = 15000$ { maximum number of control sequences; it should be at most about $(mem\_max - mem\_min)/10$; see also *font_max* }
define $hash\_prime = 8501$ { a prime number equal to about 85% of *hash_size* }
define $hyph\_prime = 607$ { another prime for hashing `\hyphenation` exceptions; if you change this, you should also change *iinf_hyphen_size*. }

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

This is the first of many sections of TEX 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, 115, 116, 117, 118, 124, 165, 173, 181, 213, 246, 253, 256, 271, 286, 297, 301, 304, 305, 308, 309, 310, 333, 361, 367, 385, 390, 391, 413, 441, 450, 483, 492, 496, 515, 516, 523, 530, 535, 542, 552, 553, 558, 595, 598, 608, 619, 649, 650, 664, 687, 722, 727, 767, 773, 817, 824, 826, 828, 831, 836, 842, 850, 875, 895, 903, 908, 910, 924, 929, 946, 950, 953, 974, 983, 985, 992, 1035, 1077, 1269, 1284, 1302, 1308, 1334, 1345, 1348, 1382, 1384, 1386, 1393, 1394, 1399, 1409, 1420, 1448, 1474, 1502, 1518, 1559, and 1572.

This code is used in section 4.

14. Later on we will say '**if** $mem\_max \ge max\_halfword$ **then** $bad \leftarrow 14$', or something similar. (We can't do that until *max_halfword* has been defined.)

⟨ Check the "constant" values for consistency 14 ⟩ ≡
$bad \leftarrow 0$;
if $(half\_error\_line < 30) \lor (half\_error\_line > error\_line - 15)$ **then** $bad \leftarrow 1$;
if $max\_print\_line < 60$ **then** $bad \leftarrow 2$;
if dvi_buf_size **mod** $8 \ne 0$ **then** $bad \leftarrow 3$;
if $mem\_bot + 1100 > mem\_top$ **then** $bad \leftarrow 4$;
if $hash\_prime > hash\_size$ **then** $bad \leftarrow 5$;
if $max\_in\_open \ge 128$ **then** $bad \leftarrow 6$;
if $mem\_top < 256 + 11$ **then** $bad \leftarrow 7$; { we will want $null\_list > 255$ }

See also sections 111, 290, 525, and 1252.

This code is used in section 1335.

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 *common_ending* = 50 { go here when you want to merge with another branch }

16. Here are some macros for common programming idioms.

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.

⟨ Types in the outer block 18 ⟩ ≡
ASCII_code = 0 . . 255; { eight-bit numbers }

See also sections 25, 38, 101, 109, 113, 150, 212, 269, 300, 551, 597, 923, 928, 1439, 1473, 1500, and 1517.

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

⟨ Local variables for initialization 19 ⟩ ≡
i: *integer*;

See also sections 163 and 930.

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.

⟨ Global variables 13 ⟩ +≡
xord: **array** [*text_char*] **of** *ASCII_code*; { specifies conversion of input characters }
xchr: **array** [*ASCII_code*] **of** *text_char*; { specifies conversion of output characters }
xprn: **array** [*ASCII_code*] **of** *ASCII_code*; { non zero iff character is printable }

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 ⟩ ≡

```
putex – add –
    for k ← 0 to 255 do xchr[k] ← k;
putex – end –
    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, 166, 215, 254, 257, 272, 287, 368, 386, 442, 484, 493, 554, 559, 596, 599, 609, 651, 665, 688, 774, 931, 993, 1036, 1270, 1285, 1303, 1346, 1383, 1395, 1410, 1503, and 1560.

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 .. ´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 ⟩ +≡
  { Initialize *xchr* to the identity mapping. }
for $i \leftarrow 0$ **to** ´37 **do** $xchr[i] \leftarrow i$;
for $i \leftarrow$ ´177 **to** ´377 **do** $xchr[i] \leftarrow i$;

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 $i \leftarrow$ *first_text_char* **to** *last_text_char* **do** *xord*[*chr*(*i*)] ← *invalid_code*;
for $i \leftarrow$ ´200 **to** ´377 **do** *xord*[*xchr*[*i*]] ← *i*;
for $i \leftarrow 0$ **to** ´176 **do** *xord*[*xchr*[*i*]] ← *i*; { Set *xprn* for printable ASCII, unless *eight_bit_p* is set. }
for $i \leftarrow 0$ **to** 255 **do** *xprn*[*i*] ← (*eight_bit_p* ∨ ((*i* ≥ "␣") ∧ (*i* ≤ "~"))); { The idea for this dynamic translation comes from the patch by Libor Skarvada `<libor@informatics.muni.cz>` and Petr Sojka `<sojka@informatics.muni.cz>`. I didn't use any of the actual code, though, preferring a more general approach. }
  { This updates the *xchr*, *xord*, and *xprn* arrays from the provided *translate_filename*. See the function definition in `texmfmp.c` for more comments. }
if *translate_filename* **then** *read_tcx_file*;

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.

TEX 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 TEX; 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 TEX can open a file whose external name is specified by *name_of_file*.

⟨ Global variables 13 ⟩ +≡
name_of_file: ↑*text_char*;
name_length: 0 . . *file_name_size*;
{ this many characters are actually relevant in *name_of_file* (the rest are blank) }

27. All of the file opening functions are defined in C.

28. And all the file closing routines as well.

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. TEX'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: ↑*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 $f\uparrow$. 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 $f\uparrow$ 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.

We define *input_ln* in C, for efficiency. Nevertheless we quote the module 'Report overflow of the input buffer, and abort' here in order to make `WEAVE` happy, since part of that module is needed by e-TeX.

`@{`⟨ Report overflow of the input buffer, and abort 35 ⟩`@}`

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*.

define *term_in* ≡ *stdin* { the terminal as an input file }
define *term_out* ≡ *stdout* { the terminal as an output file }

⟨ Global variables 13 ⟩ +≡
init *ini_version*: *boolean*; { are we `INITEX`? }
dump_option: *boolean*; { was the dump name option used? }
dump_line: *boolean*; { was a `%&format` line seen? }
tini

bound_default: *integer*; { temporary for setup }
bound_name: *const_cstring*; { temporary for setup }

mem_bot: *integer*;
{ smallest index in the *mem* array dumped by `INITEX`; must not be less than *mem_min* }
main_memory: *integer*; { total memory words allocated in initex }
extra_mem_bot: *integer*; { $mem\_min \leftarrow mem\_bot - extra\_mem\_bot$ except in `INITEX` }
mem_min: *integer*; { smallest index in TeX's internal *mem* array; must be *min_halfword* or more; must be equal to *mem_bot* in `INITEX`, otherwise $\le mem\_bot$ }
mem_top: *integer*; { largest index in the *mem* array dumped by `INITEX`; must be substantially larger than *mem_bot*, equal to *mem_max* in `INITEX`, else not greater than *mem_max* }
extra_mem_top: *integer*; { $mem\_max \leftarrow mem\_top + extra\_mem\_top$ except in `INITEX` }
mem_max: *integer*; { greatest index in TeX's internal *mem* array; must be strictly less than *max_halfword*; must be equal to *mem_top* in `INITEX`, otherwise $\ge mem\_top$ }
error_line: *integer*; { width of context lines on terminal error messages }
half_error_line: *integer*; { width of first lines of contexts in terminal error messages; should be between 30 and $error\_line - 15$ }
max_print_line: *integer*; { width of longest text lines output; should be at least 60 }
max_strings: *integer*; { maximum number of strings; must not exceed *max_halfword* }
strings_free: *integer*; { strings available after format loaded }
string_vacancies: *integer*; { 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: *integer*; { 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 }
pool_free: *integer*; { pool space free after format loaded }
font_mem_size: *integer*; { number of words of *font_info* for all fonts }
font_max: *integer*; { maximum internal font number; ok to exceed *max_quarterword* and must be at most $font\_base + max\_font\_max$ }
font_k: *integer*; { loop variable for initialization }
hyph_size: *integer*; { maximun number of hyphen exceptions }
trie_size: *integer*; { space for hyphenation patterns; should be larger for `INITEX` than it is in production versions of TeX. 50000 is needed for English, German, and Portuguese. }
buf_size: *integer*; { 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* }
stack_size: *integer*; { maximum number of simultaneous input sources }
max_in_open: *integer*;
{ maximum number of input files and error insertions that can be going on simultaneously }
param_size: *integer*; { maximum number of simultaneous macro parameters }
nest_size: *integer*; { maximum number of semantic levels simultaneously active }
save_size: *integer*; { space for saving values outside of current group; must be at most *max_halfword* }
dvi_buf_size: *integer*; { size of the output buffer; must be a multiple of 8 }

expand_depth: *integer*; { limits recursive calls to the *expand* procedure }
parse_first_line_p: *cinttype*; { parse the first line for options }
file_line_error_style_p: *cinttype*; { format messages as file:line:error }
eight_bit_p: *cinttype*; { make all characters printable by default }
halt_on_error_p: *cinttype*; { stop at first error }
quoted_filename: *boolean*; { current filename is quoted }
{ Variables for source specials }
src_specials_p: *boolean*; { Whether *src_specials* are enabled at all }
insert_src_special_auto: *boolean*;
insert_src_special_every_par: *boolean*;
insert_src_special_every_parend: *boolean*;
insert_src_special_every_cr: *boolean*;
insert_src_special_every_math: *boolean*;
insert_src_special_every_hbox: *boolean*;
insert_src_special_every_vbox: *boolean*;
insert_src_special_every_display: *boolean*;

33. Here is how to open the terminal files. *t_open_out* does nothing. *t_open_in*, on the other hand, does the work of "rescanning," or getting any command line arguments the user has provided. It's defined in C.

define *t_open_out* ≡ { output already open 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 with UNIX. *update_terminal* does an *fflush*. *clear_terminal* is redefined to do nothing, since the user should control the terminal.

define *update_terminal* ≡ *fflush*(*term_out*)
define *clear_terminal* ≡ *do_nothing*
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.)

Routine is implemented in C; part of module is, however, needed for e-TeX.

⟨Report overflow of the input buffer, and abort 35⟩ ≡
begin *cur_input.loc_field* ← *first*; *cur_input.limit_field* ← *last* − 1; *overflow*("`buffer␣size`", *buf_size*);
end

This code is used in section 31.

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. Iff anything has been specified on the command line, then *t_open_in* will return with *last* > *first*.

```
function init_terminal: boolean;  { gets the terminal input started }
  label exit;
  begin t_open_in;
  if last > first then
    begin loc ← first;
    while (loc < last) ∧ (buffer[loc] = ´␣´) do incr(loc);
    if loc < last then
      begin init_terminal ← true; goto exit;
      end;
    end;
  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_ln(term_out, ´!␣End␣of␣file␣on␣the␣terminal...␣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 $str\_start[s] \le j < str\_start[s+1]$. 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 = *integer*; { for variables that point into *str_pool* }
str_number = 0 .. *ssup_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_ASCII_code*; { the characters }
str_start: ↑*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* and *append_wchar* macros, defined here, do not check to see if the value of *pool_ptr* has gotten too high; this test is supposed to be made before *append_char* (or *append_wchar*) 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

putex – add –

define *append_wchar*(#) ≡ { TCW: put a double-byte char # at the end of *str_pool* }
  begin *str_pool*[*pool_ptr*] ← # **div** 256; *str_pool*[*pool_ptr* + 1] ← # **mod** 256; *pool_ptr* ← *pool_ptr* + 2;
  end

putex – 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 $\TeX$.

⟨ Declare additional routines for string recycling 1391 ⟩
  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* **div** 16); *app_lc_hex*(*k* **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 $xchr[\'32] = \text{\'≠\'}$, would like string ´32 to be printed as 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 '$k \in [0, \'10 \,..\, \'12, \'14, \'15, \'33, \'177 \,..\, \'377]$'. If character k cannot be printed, and $k < \'200$, then character $k + \'100$ or $k - \'100$ must be printable; moreover, ASCII codes $[\'41 \,..\, \'46, \'60 \,..\, \'71, \'136, \'141 \,..\, \'146, \'160 \,..\, \'171]$ must be printable. Thus, at least 81 printable characters are needed.

⟨ Character k cannot be printed 49 ⟩ ≡
  $(k < \text{"␣"}) \lor (k > \text{"\~{}"})$

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_length ← *strlen*(*pool_name*); *name_of_file* ← *xmalloc_array*(*ASCII_code*, *name_length* + 1);
  strcpy(*stringcast*(*name_of_file* + 1), *pool_name*); { copy the string }
  if *a_open_in*(*pool_file*, *kpse_texpool_format*) **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*('`!␣I␣can''t␣read␣`', *pool_name*, '`;␣bad␣path?`')

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('!␣', pool_name, '␣has␣no␣check␣sum.');
read(pool_file, m); read(pool_file, 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('!␣', pool_name, '␣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('!␣', pool_name, '␣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('!␣', pool_name, '␣doesn''t␣match;␣tangle␣me␣again␣(or␣fix␣the␣path).');
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 .. *ssup_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, 531, and 536.
This code is used in section 1335.

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, 262, 263, 521, 702, 1358, 1385, 1387, 1422, 1423, 1424, and 1434.
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*, *print_char*, or *print_wchar*.

TCW: The *print_wchar* macro is used to print one DBCS character.

```
define print_wchar(#) ≡
        begin print_char((#) div 256); print_char((#) mod 256)
        end   { TCW }
⟨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 244⟩ 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 244 ⟩) 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 the banner and format identifier together will occupy at most *max_print_line* character positions.

define *putex_banner* ≡ ´␣(PUTeX␣4.0␣CJK␣version)´

⟨ Initialize the output routines 55 ⟩ +≡
if *src_specials_p* ∨ *file_line_error_style_p* ∨ *parse_first_line_p* **then** *wterm*(*banner_k*)
else *wterm*(*banner*);
wterm(*version_string*);
if *format_ident* > 0 **then** *slow_print*(*format_ident*);
print_ln;
if *shellenabledp* **then**
  begin *wterm*(´␣´);
  if *restrictedshell* **then**
    begin *wterm*(´restricted␣´);
    end;
  wterm_ln(´\write18␣enabled.´);
  end;
if *src_specials_p* **then**
  begin *wterm_ln*(´␣Source␣specials␣enabled.´)
  end;
if *translate_filename* **then**
  begin *wterm*(´␣(´); *fputs*(*translate_filename*, *stdout*); *wterm_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 243 ⟩;
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 $dig[k-1] \ldots dig[0]$ }
  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* : *integer*); { prints an integer in decimal form }
  var *k*: 0 . . 23; { index to current digit; we assume that $n < 10^{23}$ }
    m: *integer*; { 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 $m \leftarrow -1 - n$; $n \leftarrow m$ **div** 10; $m \leftarrow (m$ **mod** $10) + 1$; $k \leftarrow 1$;
      if $m < 10$ **then** $dig[0] \leftarrow m$
      else begin $dig[0] \leftarrow 0$; *incr*(*n*);
        end;
      end;
    end;
  repeat $dig[k] \leftarrow n$ **mod** 10; $n \leftarrow n$ **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 $0 \le n \le 99$.

procedure *print_two*(*n* : *integer*); { prints two least significant digits }
  begin $n \leftarrow abs(n)$ **mod** 100; *print_char*("0" + (*n* **div** 10)); *print_char*("0" + (*n* **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 $0 \le n < 16^{22}$ }
  begin $k \leftarrow 0$; *print_char*("""");
  repeat $dig[k] \leftarrow n$ **mod** 16; $n \leftarrow n$ **div** 16; *incr*(*k*);
  until $n = 0$;
  print_the_digs(*k*);
  end;

68. Old versions of TEX 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 archæologists.

  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 `mxm`.

```
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["m2d5c2l5x2v5i"]; 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.␣I´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 *unspecified_mode* = 4 { extra value for command-line switch }
define *print_err*(#) ≡
  begin if *interaction* = *error_stop_mode* **then** *wake_up_terminal*;
  if *file_line_error_style_p* **then** *print_file_line*
  else *print_nl*("!␣");
  print(#);
  end

⟨ Global variables 13 ⟩ +≡
interaction: *batch_mode* . . *error_stop_mode*; { current level of interaction }
interaction_option: *batch_mode* . . *unspecified_mode*; { set from command line }

74. ⟨ Set initial values of key variables 21 ⟩ +≡
if *interaction_option* = *unspecified_mode* **then** *interaction* ← *error_stop_mode*
else *interaction* ← *interaction_option*;

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 1268 and 1340.

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.

define *do_final_end* ≡
  begin *update_terminal*; *ready_already* ← 0;
  if (*history* ≠ *spotless*) ∧ (*history* ≠ *warning_issued*) **then** *uexit*(1)
  else *uexit*(0);
  end

⟨ Error handling procedures 78 ⟩ +≡
noreturn
procedure *jump_out*;
begin *close_files_and_terminate*; *do_final_end*;
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 (halt_on_error_p) then
    begin history ← fatal_error_stop; jump_out;
    end;
  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: clear_for_error_prompt; prompt_input("?␣");
    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. We do this by calling the external procedure *call_edit* with a pointer to the filename, its length, and the line number. However, here we just set up the variables that will be used as arguments, since we don't want to do the switch-to-editor until after TeX has closed its files.

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

```
define edit_file ≡ input_stack[base_ptr]

⟨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
      begin edit_name_start ← str_start[edit_file.name_field];
      edit_name_length ← str_start[edit_file.name_field + 1] − str_start[edit_file.name_field];
      edit_line ← line; 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␣<return>␣to␣proceed,␣S␣to␣scroll␣future␣error␣messages,");
  print_nl("R␣to␣run␣without␣stopping,␣Q␣to␣run␣quietly,");
  print_nl("I␣to␣insert␣something,␣");
  if base_ptr > 0 then print("E␣to␣edit␣your␣file,");
  if deletions_allowed then
    print_nl("1␣or␣...␣or␣9␣to␣ignore␣the␣next␣1␣to␣9␣tokens␣of␣input,");
  print_nl("H␣for␣help,␣X␣to␣quit.");
  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 TEX'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⟩ +≡
  noreturn
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 ⟩ +≡
  noreturn
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 TEX 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 ⟩ +≡
  noreturn
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␣can␣fix");
    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 TEX 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* $\neq 0$ **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␣some␣instructions␣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 TEX 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 TEX 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 TEX has been designed with care, and systems that claim to be implementations of TEX82 should follow precisely the calculations as they appear in the present program.

(Actually there are three places where TEX 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 . . ′*17777777777*; { $0 \le x < 2^{31}$ }
  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 $(.d_0 d_1 \ldots d_{k-1})$, where $0 \le k \le 17$. 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 $s - \delta \le 10 \cdot 2^{16} f < s$. 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** $x \ge$ ´*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.

```
define remainder ≡ tex_remainder
⟨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(#, ´7777777777)
  define mult_integers(#) ≡ mult_and_add(#, 0, ´17777777777)
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 $x \ge 0$ **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 $\le 2^{16}$ 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 $x \ge 0$? }
    t, u, v: *nonnegative_integer*; { intermediate quantities }
  begin if $x \ge 0$ **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** $d \ge$ ´*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

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

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 $t \geq 0$ }
var r: *integer*; { approximation to $\alpha t/s$, where $\alpha^3 \approx 100 \cdot 2^{18}$ }
begin if $t = 0$ **then** *badness* ← 0
else if $s \leq 0$ **then** *badness* ← *inf_bad*
else begin if $t \leq 7230584$ **then** $r \leftarrow (t * 297)$ **div** s { $297^3 = 99.94 \times 2^{18}$ }
else if $s \geq 1663497$ **then** $r \leftarrow t$ **div** (s **div** 297)
else $r \leftarrow t$;
if $r > 1290$ **then** *badness* ← *inf_bad* { $1290^3 < 2^{31} < 1291^3$ }
else *badness* ← $(r * r * r + {'}400000)$ **div** $'1000000$;
end; { that was $r^3/2^{18}$, 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 ⟩ +≡

110. 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:

$x.int$	(an *integer*)
$x.sc$	(a *scaled* integer)
$x.gr$	(a *glue_ratio*)
$x.hh.lh$, $x.hh.rh$	(two halfword fields)
$x.hh.b0$, $x.hh.b1$, $x.hh.rh$	(two quarterword fields, one halfword field)
$x.qqqq.b0$, $x.qqqq.b1$, $x.qqqq.b2$, $x.qqqq.b3$	(four quarterword 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* ≡ −"FFFFFFF { smallest allowable value in a *halfword* }
define *max_halfword* ≡ "FFFFFFF { largest allowable value in a *halfword* }

111. 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_bot* − *sup_main_memory* < *min_halfword*) ∨ (*mem_top* + *sup_main_memory* ≥ *max_halfword*)
    then *bad* ← 14;
if (*max_font_max* < *min_halfword*) ∨ (*max_font_max* > *max_halfword*) **then** *bad* ← 15;
if *font_max* > *font_base* + *max_font_max* **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;

112. 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 $min\_quarterword = 0$. So they have been simplified here in the obvious way.

The WEB source for TEX defines $hi(\#) \equiv \# + min\_halfword$ which can be simplified when $min\_halfword = 0$. The Web2C implemetation of TEX can use $hi(\#) \equiv \#$ together with $min\_halfword < 0$ as long as *max_halfword* is sufficiently large.

define $qi(\#) \equiv \#$ { to put an *eight_bits* item into a quarterword }
define $qo(\#) \equiv \#$ { to take an *eight_bits* item from a quarterword }
define $hi(\#) \equiv \#$ { to put a sixteen-bit item into a halfword }
define $ho(\#) \equiv \#$ { to take a sixteen-bit item from a halfword }

113. The reader should study the following definitions closely:

define $sc \equiv int$ { *scaled* data is equivalent to *integer* }

⟨ Types in the outer block 18 ⟩ +≡
quarterword = *min_quarterword* .. *max_quarterword*; *halfword* = *min_halfword* .. *max_halfword*;
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 }
`#include␣"texmfmem.h";` *word_file* = **file of** *memory_word*;

114. 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

115. 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 }

116. 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:

$$null \le mem\_min \le mem\_bot < lo\_mem\_max < hi\_mem\_min < mem\_top \le mem\_end \le mem\_max\,.$$

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 ⟩ +≡
yzmem: ↑*memory_word*; { the big dynamic storage area }
zmem: ↑*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 }

117. 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 }

118. 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

$$avail,\ link(avail),\ link(link(avail)),\ \ldots$$

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* }

119. 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* 292 ⟩
⟨ Declare the procedure called *runaway* 306 ⟩

120. The function *get_avail* returns a pointer to a new one-word node whose *link* field is null. However, TeX 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;
```

121. Conversely, a one-word node is recycled by calling *free_avail*. This routine is part of TeX'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
```

122. 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*(#) $\equiv$
  begin # $\leftarrow$ *avail*; { avoid *get_avail* if possible, to save time }
  if # = *null* **then** # $\leftarrow$ *get_avail*
  else begin *avail* $\leftarrow$ *link*(#); *link*(#) $\leftarrow$ *null*;
    stat *incr*(*dyn_used*); **tats**
    end;
  end

123. 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 $p \neq$ *null* **then**
    begin $r \leftarrow p$;
    repeat $q \leftarrow r$; $r \leftarrow$ *link*(r);
      stat *decr*(*dyn_used*); **tats**
    until $r =$ *null*; { now q is the last node on the list }
    link(q) $\leftarrow$ *avail*; *avail* $\leftarrow p$;
    end;
  end;

124. 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* $\equiv$ *max_halfword* { the *link* of an empty variable-size node }
define *is_empty*(#) $\equiv$ (*link*(#) = *empty_flag*) { tests for empty node }
define *node_size* $\equiv$ *info* { the size field in empty variable-size nodes }
define *llink*(#) $\equiv$ *info*(# + 1) { left link in doubly-linked list of empty nodes }
define *rlink*(#) $\equiv$ *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 }

125. 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*: $p \leftarrow rover$; { 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 127 ⟩;
    $p \leftarrow rlink(p)$; { move to the next node in the ring }
  until $p = rover$; { repeat until the whole list has been traversed }
  if $s =$ ´*10000000000* **then**
    begin $get\_node \leftarrow max\_halfword$; **return**;
    end;
  if $lo\_mem\_max + 2 < hi\_mem\_min$ **then**
    if $lo\_mem\_max + 2 \le mem\_bot + max\_halfword$ **then**
      ⟨ Grow more variable-size memory and **goto** *restart* 126 ⟩;
  overflow("main␣memory␣size", $mem\_max + 1 - mem\_min$); { sorry, nothing satisfactory is left }
found: $link(r) \leftarrow null$; { this node is now nonempty }
  stat $var\_used \leftarrow var\_used + s$; { maintain usage statistics }
  tats
  $get\_node \leftarrow r$;
exit: **end**;

126. 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* 126 ⟩ ≡
  begin if $hi\_mem\_min - lo\_mem\_max \ge 1998$ **then** $t \leftarrow lo\_mem\_max + 1000$
  else $t \leftarrow lo\_mem\_max + 1 + (hi\_mem\_min - lo\_mem\_max)$ **div** 2; { $lo\_mem\_max + 2 \le t < hi\_mem\_min$ }
  $p \leftarrow llink(rover)$; $q \leftarrow lo\_mem\_max$; $rlink(p) \leftarrow q$; $llink(rover) \leftarrow q$;
  if $t > mem\_bot + max\_halfword$ **then** $t \leftarrow mem\_bot + max\_halfword$;
  $rlink(q) \leftarrow rover$; $llink(q) \leftarrow p$; $link(q) \leftarrow empty\_flag$; $node\_size(q) \leftarrow t - lo\_mem\_max$;
  $lo\_mem\_max \leftarrow t$; $link(lo\_mem\_max) \leftarrow null$; $info(lo\_mem\_max) \leftarrow null$; $rover \leftarrow q$; **goto** *restart*;
  end

This code is used in section 125.

127. 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 127 ⟩ ≡
  $q \leftarrow p + node\_size(p)$; { find the physical successor }
  while $is\_empty(q)$ **do** { merge node p with node q }
    begin $t \leftarrow rlink(q)$;
    if $q = rover$ **then** $rover \leftarrow t$;
    $llink(t) \leftarrow llink(q)$; $rlink(llink(q)) \leftarrow t$;
    $q \leftarrow q + node\_size(q)$;
    end;
  $r \leftarrow q - s$;
  if $r > intcast(p + 1)$ **then** ⟨ Allocate from the top of node p and **goto** *found* 128 ⟩;
  if $r = p$ **then**
    if $rlink(p) \neq p$ **then** ⟨ Allocate entire node p and **goto** *found* 129 ⟩;
  $node\_size(p) \leftarrow q - p$ { reset the size in case it grew }

This code is used in section 125.

128. ⟨ Allocate from the top of node p and **goto** *found* 128 ⟩ ≡
  begin $node\_size(p) \leftarrow r - p$; { store the remaining size }
  $rover \leftarrow p$; { start searching here next time }
  goto *found*;
  end

This code is used in section 127.

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

⟨ Allocate entire node p and **goto** *found* 129 ⟩ ≡
  begin $rover \leftarrow rlink(p)$; $t \leftarrow llink(p)$; $llink(rover) \leftarrow t$; $rlink(t) \leftarrow rover$; **goto** *found*;
  end

This code is used in section 127.

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

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

131. 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*(´*10000000000*); { 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*) 132⟩;
$p \leftarrow$ *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

132. 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*) 132⟩ ≡
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 131.

133. 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 }

134. 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 saving feature about oriental characters is that most of them have the same box 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* }
define *is_wchar_node*(#) ≡ (*character*(#) > 255)
{ TCW: is the argument a double-byte character code? }
define *is_wchar*(#) ≡ ((#) > 255) { TCW: is the argument a double-byte character code? }

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

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 }

136. 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*, since that's the desired *span_count* value if this *hlist_node* is changed to an *unset_node*.

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;

137. 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 }

138. 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* { -2^{30}, signifies a missing item }
define *is_running*(#) ≡ (# = *null_flag*) { tests for a running dimension }

139. 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 $p \leftarrow$ *get_node*(*rule_node_size*); *type*(p) $\leftarrow$ *rule_node*; *subtype*(p) $\leftarrow 0$; { the *subtype* is not used }
  width(p) $\leftarrow$ *null_flag*; *depth*(p) $\leftarrow$ *null_flag*; *height*(p) $\leftarrow$ *null_flag*; *new_rule* $\leftarrow p$;
  end;

140. 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 $250 +$ *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*(#) $\equiv$ *mem*[# + 1].*int* { the *floating_penalty* to be used }
define *ins_ptr*(#) $\equiv$ *info*(# + 4) { the vertical list to be inserted }
define *split_top_ptr*(#) $\equiv$ *link*(# + 4) { the *split_top_skip* to be used }

141. A *mark_node* has a *mark_ptr* field that points to the reference count of a token list that contains the user's `\mark` text. This field occupies a full word instead of a halfword, because there's nothing to put in the other halfword; it is easier in Pascal to use the full word than to risk leaving garbage in the unused half.

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*(#) $\equiv$ *mem*[# + 1].*int* { head of the token list for a mark }

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

define *adjust_node* $= 5$ { *type* of an adjust node }
define *adjust_ptr* $\equiv$ *mark_ptr* { vertical list to be moved out of horizontal list }

143. 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*(#) $\equiv$ # $+ 1$ { the word where the ligature is to be found }
define *lig_ptr*(#) $\equiv$ *link*(*lig_char*(#)) { the list of characters }

144. 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* : *internal_font_number*; *c* : *quarterword*; *q* : *pointer*): *pointer*;
  var *p*: *pointer*; { the new node }
  begin $p \leftarrow$ *get_node*(*small_node_size*); *type*(*p*) $\leftarrow$ *ligature_node*; *font*(*lig_char*(*p*)) $\leftarrow f$;
  character(*lig_char*(*p*)) $\leftarrow c$; *lig_ptr*(*p*) $\leftarrow q$; *subtype*(*p*) $\leftarrow 0$; *new_ligature* $\leftarrow p$;
  end;

function *new_lig_item*(*c* : *quarterword*): *pointer*;
  var *p*: *pointer*; { the new node }
  begin $p \leftarrow$ *get_node*(*small_node_size*); *character*(*p*) $\leftarrow c$; *lig_ptr*(*p*) $\leftarrow$ *null*; *new_lig_item* $\leftarrow p$;
  end;

145. 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* $\equiv$ *subtype* { how many subsequent nodes to replace }
  define *pre_break* $\equiv$ *llink* { text that precedes a discretionary break }
  define *post_break* $\equiv$ *rlink* { text that follows a discretionary break }

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

146. A *whatsit_node* is a wild card reserved for extensions to TEX. 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 TEX are able to make TEX do new things by adding code at the end of the program. For example, there might be a 'TEXnicolor' extension to specify different colors of ink, and the whatsit node might contain the desired parameters.

The present implementation of TEX 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 }

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

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 }

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

148. TEX 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*)

149. 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 }

150. 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*(#) $\equiv$ *link*(#) { reference count of a glue specification }
define *stretch*(#) $\equiv$ *mem*[# + 2].*sc* { the stretchability of this glob of glue }
define *shrink*(#) $\equiv$ *mem*[# + 3].*sc* { the shrinkability of this glob of glue }
define *stretch_order* $\equiv$ *type* { order of infinity for stretching }
define *shrink_order* $\equiv$ *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 }

151. 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 $q \leftarrow$ *get_node*(*glue_spec_size*);
mem[*q*] $\leftarrow$ *mem*[*p*]; *glue_ref_count*(*q*) $\leftarrow$ *null*;
width(*q*) $\leftarrow$ *width*(*p*); *stretch*(*q*) $\leftarrow$ *stretch*(*p*); *shrink*(*q*) $\leftarrow$ *shrink*(*p*); *new_spec* $\leftarrow q$;
end;

152. 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 $p \leftarrow$ *get_node*(*small_node_size*); *type*(*p*) $\leftarrow$ *glue_node*; *subtype*(*p*) $\leftarrow n + 1$; *leader_ptr*(*p*) $\leftarrow$ *null*;
$q \leftarrow$ ⟨ Current *mem* equivalent of glue parameter number *n* 224 ⟩; *glue_ptr*(*p*) $\leftarrow q$;
incr(*glue_ref_count*(*q*)); *new_param_glue* $\leftarrow p$;
end;

153. 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 $p \leftarrow$ *get_node*(*small_node_size*); *type*(*p*) $\leftarrow$ *glue_node*; *subtype*(*p*) $\leftarrow$ *normal*;
leader_ptr(*p*) $\leftarrow$ *null*; *glue_ptr*(*p*) $\leftarrow q$; *incr*(*glue_ref_count*(*q*)); *new_glue* $\leftarrow p$;
end;

154. 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* 224 ⟩);
  p ← *new_glue*(*temp_ptr*); *glue_ref_count*(*temp_ptr*) ← *null*; *subtype*(*p*) ← $n + 1$; *new_skip_param* ← *p*;
  end;

155. 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 `\/` 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 `\/` }
  define *acc_kern* = 2 { *subtype* of kern nodes from accents }

156. 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* ← *get_node*(*small_node_size*); *type*(*p*) ← *kern_node*; *subtype*(*p*) ← *normal*; *width*(*p*) ← *w*;
  new_kern ← *p*;
  end;

157. 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*(#) ≡ *mem*[# + 1].*int* { the added cost of breaking a list here }

158. 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;

159. 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 $glue\_stretch(\#) \equiv mem[\# + glue\_offset].sc$ { total stretch in an unset node }
define $glue\_shrink \equiv shift\_amount$ { total shrink in an unset node }
define $span\_count \equiv subtype$ { indicates the number of spanned columns }

160. 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.

161. 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 TEX's other algorithms intact.

162. 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 *hi_mem_stat_min* ≡ *mem_top* − 13 { smallest statically allocated word in the one-word *mem* }
define *hi_mem_stat_usage* = 14 { the number of one-word nodes always present }

163. 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. }

164. ⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ ≡
for $k \gets mem\_bot + 1$ **to** $lo\_mem\_stat\_max$ **do** $mem[k].sc \gets 0$; { all glue dimensions are zeroed }
$k \gets mem\_bot$; **while** $k \le lo\_mem\_stat\_max$ **do** { set first words of glue specifications }
  begin $glue\_ref\_count(k) \gets null + 1$; $stretch\_order(k) \gets normal$; $shrink\_order(k) \gets normal$;
  $k \gets k + glue\_spec\_size$;
  end;
$stretch(fil\_glue) \gets unity$; $stretch\_order(fil\_glue) \gets fil$;
$stretch(fill\_glue) \gets unity$; $stretch\_order(fill\_glue) \gets fill$;
$stretch(ss\_glue) \gets unity$; $stretch\_order(ss\_glue) \gets fil$;
$shrink(ss\_glue) \gets unity$; $shrink\_order(ss\_glue) \gets fil$;
$stretch(fil\_neg\_glue) \gets -unity$; $stretch\_order(fil\_neg\_glue) \gets fil$;
$rover \gets lo\_mem\_stat\_max + 1$; $link(rover) \gets empty\_flag$; { now initialize the dynamic memory }
$node\_size(rover) \gets 1000$; { which is a 1000-word available node }
$llink(rover) \gets rover$; $rlink(rover) \gets rover$;
$lo\_mem\_max \gets rover + 1000$; $link(lo\_mem\_max) \gets null$; $info(lo\_mem\_max) \gets null$;
for $k \gets hi\_mem\_stat\_min$ **to** mem_top **do** $mem[k] \gets mem[lo\_mem\_max]$; { clear list heads }
⟨ Initialize the special list heads and constant nodes 793 ⟩;
$avail \gets null$; $mem\_end \gets mem\_top$; $hi\_mem\_min \gets hi\_mem\_stat\_min$;
  { initialize the one-word memory }
$var\_used \gets lo\_mem\_stat\_max + 1 - mem\_bot$; $dyn\_used \gets hi\_mem\_stat\_usage$; { initialize statistics }

See also sections 222, 228, 232, 240, 250, 258, 555, 949, 954, 1219, 1304, 1372, 1411, 1412, 1417, 1480, 1501, 1504, 1519, and 1532.

This code is used in section 8.

165. 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.)

define $free \equiv free\_arr$

⟨ Global variables 13 ⟩ +≡
  { The debug memory arrays have not been mallocated yet. }
debug *free*: **packed array** [0 . . 9] **of** *boolean*; { free cells }
was_free: **packed array** [0 . . 9] **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

166. ⟨ Set initial values of key variables 21 ⟩ +≡
debug $was\_mem\_end \gets mem\_min$; { indicate that everything was previously free }
$was\_lo\_max \gets mem\_min$; $was\_hi\_min \gets mem\_max$; $panicking \gets false$;
gubed

167. 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 168 ⟩;
⟨ Check variable-size *avail* list 169 ⟩;
⟨ Check flags of unavailable nodes 170 ⟩;
if *print_locs* **then** ⟨ Print newly busy locations 171 ⟩;
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

168. ⟨ Check single-word *avail* list 168 ⟩ $\equiv$
$p \leftarrow$ *avail*; $q \leftarrow$ *null*; *clobbered* $\leftarrow$ *false*;
while $p \neq$ *null* **do**
  begin if ($p >$ *mem_end*) $\lor$ ($p <$ *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 167.

169. ⟨ Check variable-size *avail* list 169 ⟩ $\equiv$
$p \leftarrow$ *rover*; $q \leftarrow$ *null*; *clobbered* $\leftarrow$ *false*;
repeat if ($p \geq$ *lo_mem_max*) $\lor$ ($p <$ *mem_min*) **then** *clobbered* $\leftarrow$ *true*
  else if (*rlink*(*p*) $\geq$ *lo_mem_max*) $\lor$ (*rlink*(*p*) $<$ *mem_min*) **then** *clobbered* $\leftarrow$ *true*
    else if $\neg$(*is_empty*(*p*)) $\lor$ (*node_size*(*p*) < 2) $\lor$ ($p +$ *node_size*(*p*) $>$ *lo_mem_max*) $\lor$
        (*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 167.

170. ⟨Check flags of unavailable nodes 170⟩ ≡
```
p ← mem_min;
while p ≤ lo_mem_max do  { node p should not be empty }
  begin if is_empty(p) then
    begin print_nl("Bad␣flag␣at␣"); print_int(p);
    end;
  while (p ≤ lo_mem_max) ∧ ¬free[p] do incr(p);
  while (p ≤ lo_mem_max) ∧ free[p] do incr(p);
  end
```
This code is used in section 167.

171. ⟨Print newly busy locations 171⟩ ≡
```
begin print_nl("New␣busy␣locs:");
for p ← mem_min to lo_mem_max do
  if ¬free[p] ∧ ((p > was_lo_max) ∨ was_free[p]) then
    begin print_char("␣"); print_int(p);
    end;
for p ← hi_mem_min to mem_end do
  if ¬free[p] ∧ ((p < was_hi_min) ∨ (p > was_mem_end) ∨ was_free[p]) then
    begin print_char("␣"); print_int(p);
    end;
end
```
This code is used in section 167.

172. 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 ← 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 ← 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 255⟩;
⟨Search save_stack for equivalents that point to p 285⟩;
⟨Search hyph_list for pointers to p 936⟩;
end;
gubed
```

173. 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 }
cfont_in_short_display: *integer*; { TCW: an internal CJK font number }

174. Boxes, rules, inserts, whatsits, marks, and things in general that are sort of "complicated" are indicated only by printing '`[]`'.

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 $p \le mem\_end$ **then**
        begin if $font(p) \ne font\_in\_short\_display \land font(p) \ne cfont\_in\_short\_display$ **then**
          begin if $(font(p) > cfont\_max)$ **then** *print_char*("`*`")
          else ⟨Print the font identifier for *font*(*p*) 267⟩;
          print_char("␣");
          if $font(p) \le font\_max$ **then** $font\_in\_short\_display \leftarrow font(p)$
          else $cfont\_in\_short\_display \leftarrow font(p)$;
          end;
        if *is_wchar_node*(*p*) **then** *print_wchar*(*character*(*p*))
        else *print_ASCII*(*qo*(*character*(*p*)));
        end;
      end
    else ⟨Print a short indication of the contents of node *p* 175⟩;
    $p \leftarrow link(p)$;
    end;
  end;

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

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

This code is used in section 174.

176. 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) > cfont_max) then print_char("*")
    else ⟨Print the font identifier for font(p) 267⟩;
    print_char("␣");
    if is_wchar_node(p) then print_wchar(character(p))
    else 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;
```

177. 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) ∨ (order > filll) then print("foul")
  else if order > normal then
      begin print("fil");
      while order > fil do
        begin print_char("l"); decr(order);
        end;
      end
    else if s ≠ 0 then print(s);
  end;
```

178. 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) ∨ (p ≥ lo_mem_max) then print_char("*")
  else begin print_scaled(width(p));
    if s ≠ 0 then print(s);
    if stretch(p) ≠ 0 then
      begin print("␣plus␣"); print_glue(stretch(p), stretch_order(p), s);
      end;
    if shrink(p) ≠ 0 then
      begin print("␣minus␣"); print_glue(shrink(p), shrink_order(p), s);
      end;
    end;
  end;
```

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

⟨ Declare procedures needed for displaying the elements of mlists 694 ⟩
⟨ Declare the procedure called *print_skip_param* 225 ⟩

180. 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(#) ≡
        begin append_char("."); show_node_list(#); flush_char;
        end  { str_room need not be checked; see show_box below }
```

181. 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 ⟩ +≡
depth_threshold: integer;  { maximum nesting depth in box displays }
breadth_max: integer;  { maximum number of items shown at the same list level }
```

182. 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 183⟩;
    p ← link(p);
    end;
exit: end;
```

183. ⟨Display node p 183⟩ ≡

```
  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 184⟩;
    rule_node: ⟨Display rule p 187⟩;
    ins_node: ⟨Display insertion p 188⟩;
    whatsit_node: ⟨Display the whatsit node p 1359⟩;
    glue_node: ⟨Display glue p 189⟩;
    kern_node: ⟨Display kern p 191⟩;
    math_node: ⟨Display math node p 192⟩;
    ligature_node: ⟨Display ligature p 193⟩;
    penalty_node: ⟨Display penalty p 194⟩;
    disc_node: ⟨Display discretionary p 195⟩;
    mark_node: ⟨Display mark p 196⟩;
    adjust_node: ⟨Display adjustment p 197⟩;
    ⟨Cases of show_node_list that arise in mlists only 693⟩
    othercases print("Unknown␣node␣type!")
    endcases
```

This code is used in section 182.

184. ⟨ Display box p 184 ⟩ ≡

```
begin if type(p) = hlist_node then print_esc("h")
else if type(p) = vlist_node then print_esc("v")
  else print_esc("unset");
print("box("); print_scaled(height(p)); print_char("+"); print_scaled(depth(p)); print(")x");
print_scaled(width(p));
if type(p) = unset_node then ⟨ Display special fields of the unset node p 185 ⟩
else begin ⟨ Display the value of glue_set(p) 186 ⟩;
  if shift_amount(p) ≠ 0 then
    begin print(",␣shifted␣"); print_scaled(shift_amount(p));
    end;
  end;
node_list_display(list_ptr(p)); { recursive call }
end
```

This code is used in section 183.

185. ⟨ Display special fields of the unset node p 185 ⟩ ≡

```
begin if span_count(p) ≠ min_quarterword then
  begin print("␣("); print_int(qo(span_count(p)) + 1); print("␣columns)");
  end;
if glue_stretch(p) ≠ 0 then
  begin print(",␣stretch␣"); print_glue(glue_stretch(p), glue_order(p), 0);
  end;
if glue_shrink(p) ≠ 0 then
  begin print(",␣shrink␣"); print_glue(glue_shrink(p), glue_sign(p), 0);
  end;
end
```

This code is used in section 184.

186. 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.

⟨ Display the value of *glue_set*(p) 186 ⟩ ≡

```
g ← float(glue_set(p));
if (g ≠ float_constant(0)) ∧ (glue_sign(p) ≠ normal) then
  begin print(",␣glue␣set␣");
  if glue_sign(p) = shrinking then print("-␣"); { The Unix pc folks removed this restriction with a
        remark that invalid bit patterns were vanishingly improbable, so we follow their example without
        really understanding it. if abs(mem[p + glue_offset].int) < '4000000 then print('?.?') else }
  if fabs(g) > float_constant(20000) then
    begin if g > float_constant(0) then print_char(">")
    else print("<␣-");
    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 184.

187. ⟨ Display rule *p* 187 ⟩ ≡
```
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 183.

188. ⟨ Display insertion *p* 188 ⟩ ≡
```
begin print_esc("insert"); print_int(qo(subtype(p))); print(",␣natural␣size␣");
print_scaled(height(p)); print(";␣split("); print_spec(split_top_ptr(p),0); print_char(",");
print_scaled(depth(p)); print(");␣float␣cost␣"); print_int(float_cost(p)); node_list_display(ins_ptr(p));
    { recursive call }
end
```
This code is used in section 183.

189. ⟨ Display glue *p* 189 ⟩ ≡
```
if subtype(p) ≥ a_leaders then ⟨ Display leaders p 190 ⟩
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("␣");
    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 183.

190. ⟨ Display leaders *p* 190 ⟩ ≡
```
begin print_esc("");
if subtype(p) = c_leaders then print_char("c")
else if subtype(p) = x_leaders then print_char("x");
print("leaders␣"); print_spec(glue_ptr(p),0); node_list_display(leader_ptr(p));   { recursive call }
end
```
This code is used in section 189.

191. An "explicit" kern value is indicated implicitly by an explicit space.

⟨ Display kern *p* 191 ⟩ ≡
```
if subtype(p) ≠ mu_glue then
  begin print_esc("kern");
  if subtype(p) ≠ normal then print_char("␣");
  print_scaled(width(p));
  if subtype(p) = acc_kern then print("␣(for␣accent)");
  end
else begin print_esc("mkern"); print_scaled(width(p)); print("mu");
  end
```
This code is used in section 183.

192. ⟨ Display math node p 192 ⟩ ≡
begin *print_esc*("math");
if *subtype*(p) = *before* **then** *print*("on")
else *print*("off");
if *width*(p) $\neq 0$ **then**
begin *print*(",␣surrounded␣"); *print_scaled*(*width*(p));
end;
end

This code is used in section 183.

193. ⟨ Display ligature p 193 ⟩ ≡
begin *print_font_and_char*(*lig_char*(p)); *print*("␣(ligature␣");
if *subtype*(p) > 1 **then** *print_char*("|");
font_in_short_display ← *font*(*lig_char*(p)); *short_display*(*lig_ptr*(p));
if *odd*(*subtype*(p)) **then** *print_char*("|");
print_char(")");
end

This code is used in section 183.

194. ⟨ Display penalty p 194 ⟩ ≡
begin *print_esc*("penalty␣"); *print_int*(*penalty*(p));
end

This code is used in section 183.

195. The *post_break* list of a discretionary node is indicated by a prefixed '|' instead of the '.' before the *pre_break* list.

⟨ Display discretionary p 195 ⟩ ≡
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("|"); *show_node_list*(*post_break*(p)); *flush_char*; { recursive call }
end

This code is used in section 183.

196. ⟨ Display mark p 196 ⟩ ≡
begin *print_esc*("mark"); *print_mark*(*mark_ptr*(p));
end

This code is used in section 183.

197. ⟨ Display adjustment p 197 ⟩ ≡
begin *print_esc*("vadjust"); *node_list_display*(*adjust_ptr*(p)); { recursive call }
end

This code is used in section 183.

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

procedure *show_box*(*p* : *pointer*);
  begin ⟨Assign the values *depth_threshold* ← *show_box_depth* and *breadth_max* ← *show_box_breadth* 236⟩;
  if *breadth_max* ≤ 0 **then** *breadth_max* ← 5;
  if *pool_ptr* + *depth_threshold* ≥ *pool_size* **then** *depth_threshold* ← *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;

199. 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.

200. 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;

201. 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);

202. 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 $p \neq null$ **do**
    begin $q \leftarrow link(p)$;
    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* 1361 ⟩;
      glue_node: **begin** *fast_delete_glue_ref*(*glue_ptr*(*p*));
        if $leader\_ptr(p) \neq null$ **then** *flush_node_list*(*leader_ptr*(*p*));
        end;
      kern_node, *math_node*, *penalty_node*: *do_nothing*;
      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 701 ⟩
      othercases *confusion*("flushing")
      endcases;
      free_node(*p*, *small_node_size*);
    done: **end**;
    $p \leftarrow q$;
    end;
  end;

203. 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 }

204. 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 205⟩;
    link(q) ← r; q ← r; p ← link(p);
    end;
  link(q) ← null; q ← link(h); free_avail(h); copy_node_list ← q;
  end;
```

205. ⟨Make a copy of node p in node r 205⟩ ≡

```
  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 206⟩;
  while words > 0 do
    begin decr(words); mem[r + words] ← mem[p + words];
    end
```

This code is used in section 204.

206. ⟨ Case statement to copy different types and set *words* to the number of initial words not yet copied 206 ⟩ ≡

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 1360 ⟩;
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;
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 205.

207. 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 (␣) }
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 }
define *boundary_normal* = 0 { CJK characters can be in any positions of lines }
define *tail_forbidden* = 1 { CJK characters can't be put in the head of lines }
define *head_forbidden* = 2 { CJK characters can't be put in the tail of lines }
define *max_type_code* = 2 { largest boundary code for CJK characters }
define *set_type_code_end*(#) ≡ #
  end
define *set_type_code*(#) ≡
  begin *type_code*(#) ← *set_type_code_end*

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

define *char_num* = 16 { character specified numerically (`\char`) }
define *math_char_num* = 17 { explicit math code (`\mathchar`) }
define *mark* = 18 { mark definition (`\mark`) }
define *xray* = 19 { peek inside of TEX (`\show`, `\showbox`, etc.) }
define *make_box* = 20 { make a box (`\box`, `\copy`, `\hbox`, etc.) }
define *hmove* = 21 { horizontal motion (`\moveleft`, `\moveright`) }
define *vmove* = 22 { vertical motion (`\raise`, `\lower`) }
define *un_hbox* = 23 { unglue a box (`\unhbox`, `\unhcopy`) }
define *un_vbox* = 24 { unglue a box (`\unvbox`, `\unvcopy`) }
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`) }
define *no_align* = 34 { temporary escape from alignment (`\noalign`) }
define *vrule* = 35 { vertical rule (`\vrule`) }
define *hrule* = 36 { horizontal rule (`\hrule`) }
define *insert* = 37 { vlist inserted in box (`\insert`) }
define *vadjust* = 38 { vlist inserted in enclosing paragraph (`\vadjust`) }
define *ignore_spaces* = 39 { gobble *spacer* tokens (`\ignorespaces`) }
define *after_assignment* = 40 { save till assignment is done (`\afterassignment`) }
define *after_group* = 41 { save till group is done (`\aftergroup`) }
define *break_penalty* = 42 { additional badness (`\penalty`) }
define *start_par* = 43 { begin paragraph (`\indent`, `\noindent`) }
define *ital_corr* = 44 { italic correction (`\/`) }
define *accent* = 45 { attach accent in text (`\accent`) }
define *math_accent* = 46 { attach accent in math (`\mathaccent`) }
define *discretionary* = 47 { discretionary texts (`\-`, `\discretionary`) }
define *eq_no* = 48 { equation number (`\eqno`, `\leqno`) }
define *left_right* = 49 { variable delimiter (`\left`, `\right`) }
define *math_comp* = 50 { component of formula (`\mathbin`, etc.) }
define *limit_switch* = 51 { diddle limit conventions (`\displaylimits`, etc.) }
define *above* = 52 { generalized fraction (`\above`, `\atop`, etc.) }
define *math_style* = 53 { style specification (`\displaystyle`, etc.) }
define *math_choice* = 54 { choice specification (`\mathchoice`) }
define *non_script* = 55 { conditional math glue (`\nonscript`) }
define *vcenter* = 56 { vertically center a vbox (`\vcenter`) }
define *case_shift* = 57 { force specific case (`\lowercase`, `\uppercase`) }
define *message* = 58 { send to user (`\message`, `\errmessage`) }
define *extension* = 59 { extensions to TEX (`\write`, `\special`, etc.) }
define *in_stream* = 60 { files for reading (`\openin`, `\closein`) }
define *begin_group* = 61 { begin local grouping (`\begingroup`) }
define *end_group* = 62 { end local grouping (`\endgroup`) }
define *omit* = 63 { omit alignment template (`\omit`) }
define *ex_space* = 64 { explicit space (`\␣`) }
define *no_boundary* = 65 { suppress boundary ligatures (`\noboundary`) }

define *radical* = 66 { square root and similar signs (`\radical`) }
define *end_cs_name* = 67 { end control sequence (`\endcsname`) }
define *min_internal* = 68 { the smallest code that can follow `\the` }
define *char_given* = 68 { character code defined by `\chardef` }
define *math_given* = 69 { math code defined by `\mathchardef` }
define *last_item* = 70 { most recent item (`\lastpenalty`, `\lastkern`, `\lastskip`) }
define *max_non_prefixed_command* = 70 { largest command code that can't be `\global` }

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

TCW: Add 3 internal commands: *set_cfont*, *puxg_assign_flag*, and *puxg_assign_int*. Add 12 user commands: *pux_cface_def*, *pux_face_match*, *pux_font_match*, *pux_set_cface*, *pux_set_cface_attrib*, *pux_set_cfont_attrib*, *pux_char_num*, *pux_char_given*, *pux_space*, *pux_range_catcode*, *pux_range_type_code*, and *pux_dump_font_info*.

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) }
define *set_box_dimen* = 83 { change dimension of box (\wd, \ht, \dp) }
define *set_shape* = 84 { specify fancy paragraph shape (\parshape) }
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 *set_cfont* = 88 { TCW: set current chinese font (font identifiers) }
define *def_font* = 89 { define a font file (\font) }
define *register* = 90 { internal register (\count, \dimen, etc.) }
define *puxg_assign_flag* = 91 { TCW: set a PUTeX global flag (\puxgCdiOut, \puxgRotateCtext) }
define *puxg_assign_int* = 92 { TCW: set a PUTeX global integer (\puxgCspace, \puxgCEspace) }
define *pux_get_int* = 93 { TCW: get internal integer values (\PUXnumdigits, \PUXsign, \PUXdigit) }
define *max_internal* = 93 { the largest code that can follow \the }
define *advance* = 94 { advance a register or parameter (\advance) }
define *multiply* = 95 { multiply a register or parameter (\multiply) }
define *divide* = 96 { divide a register or parameter (\divide) }
define *prefix* = 97 { qualify a definition (\global, \long, \outer) }
define *let* = 98 { assign a command code (\let, \futurelet) }
define *shorthand_def* = 99 { code definition (\chardef, \countdef, etc.) }
{ or \charsubdef }
define *read_to_cs* = 100 { read into a control sequence (\read) }
define *def* = 101 { macro definition (\def, \gdef, \xdef, \edef) }
define *set_box* = 102 { set a box (\setbox) }
define *hyph_data* = 103 { hyphenation data (\hyphenation, \patterns) }
define *set_interaction* = 104 { define level of interaction (\batchmode, etc.) }
define *pux_cface_def* = 105 { TCW: define a chinese font face (\PUXcfacedef) }
define *pux_face_match* = 106 { TCW: English and Chinese face matching pair (\PUXfacematch) }
define *pux_font_match* = 107 { TCW: English and CJK font matching pair (\PUXfontmatch) }
define *pux_set_cface* = 108 { TCW: Set Chinese face }
define *pux_set_cface_attrib* = 109
{ TCW: Set attributes of a Chinese face (\PUXsetcfacecspace, etc.) }
define *pux_set_cfont_attrib* = 110
{ TCW: Set attributes of a CJK font (\PUXsetcfontcspace, etc.) }
define *pux_char_num* = 111 { TCW: Chinese character number (\PUXchar) }
define *pux_char_given* = 112 { TCW: define a Chinese character (\PUXchardef) }

define *pux_space* = 113 { Append space glue between Chinese and Tex characters (`\PUXcespace`) }
define *pux_range_catcode* = 114 { TCW: set catcodes for a range of characters(`\PUXrangecatcode`) }
define *pux_range_type_code* = 115 { TCW: set catcodes for a range of characters(`\PUXrangecatcode`) }
define *pux_split_number* = 116 { TCW: split a number to digits (`\PUXsplitnumber`) }
define *puxg_assign_space* = 117 { TCW: set a PUTEX global integer (`\puxgCspace`, `\puxgCEspace`) }
define *pux_set_default_cface* = 118 { TCW: set default CJK font face (`\PUXsetdefaultcface`) }
define *pux_dump_font_info* = 119 { TCW: dump font information (`\PUXdumpfontinfo`) }
define *max_command* = 119 { the largest command code seen at *big_switch* }

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

define *undefined_cs* = *max_command* + 1 { initial state of most *eq_type* fields }
define *expand_after* = *max_command* + 2 { special expansion (`\expandafter`) }
define *no_expand* = *max_command* + 3 { special nonexpansion (`\noexpand`) }
define *input* = *max_command* + 4 { input a source file (`\input`, `\endinput`) }
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`) }
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 }

211. 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 in the *ship_out* routine.

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␣mode");
    1: print("horizontal␣mode");
    2: print("display␣math␣mode");
    end
  else if m = 0 then print("no␣mode")
    else case (−m) div (max_command + 1) of
      0: print("internal␣vertical␣mode");
      1: print("restricted␣horizontal␣mode");
      2: print("math␣mode");
      end;
  end;
procedure print_in_mode(m : integer);  { prints the mode represented by m }
  begin if m > 0 then
    case m div (max_command + 1) of
    0: print("´␣in␣vertical␣mode");
    1: print("´␣in␣horizontal␣mode");
    2: print("´␣in␣display␣math␣mode");
    end
  else if m = 0 then print("´␣in␣no␣mode")
    else case (−m) div (max_command + 1) of
      0: print("´␣in␣internal␣vertical␣mode");
      1: print("´␣in␣restricted␣horizontal␣mode");
      2: print("´␣in␣math␣mode");
      end;
  end;
```

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

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

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

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

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

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

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

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

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

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

define *ignore_depth* ≡ −65536000 { *prev_depth* value that is ignored }

⟨ Types in the outer block 18 ⟩ +≡
list_state_record = **record** *mode_field*: −*mmode* .. *mmode*; *head_field*, *tail_field*: *pointer*;
pg_field, *ml_field*: *integer*; *aux_field*: *memory_word*;
end;

213. **define** *mode* ≡ *cur_list.mode_field* { current mode }
define *head* ≡ *cur_list.head_field* { header node of current list }
define *tail* ≡ *cur_list.tail_field* { final node on current list }
define *prev_graf* ≡ *cur_list.pg_field* { number of paragraph lines accumulated }
define *aux* ≡ *cur_list.aux_field* { auxiliary data about the current list }
define *prev_depth* ≡ *aux.sc* { the name of *aux* in vertical mode }
define *space_factor* ≡ *aux.hh.lh* { part of *aux* in horizontal mode }
define *clang* ≡ *aux.hh.rh* { the other part of *aux* in horizontal mode }
define *incompleat_noad* ≡ *aux.int* { the name of *aux* in math mode }
define *mode_line* ≡ *cur_list.ml_field* { source file line number at beginning of list }

⟨ Global variables 13 ⟩ +≡
nest: ↑*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` }

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

define *tail_append*(#) ≡
begin *link*(*tail*) ← #; *tail* ← *link*(*tail*);
end

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

⟨ Set initial values of key variables 21 ⟩ +≡
  nest_ptr ← 0; *max_nest_stack* ← 0; *mode* ← *vmode*; *head* ← *contrib_head*; *tail* ← *contrib_head*;
  prev_depth ← *ignore_depth*; *mode_line* ← 0; *prev_graf* ← 0; *shown_mode* ← 0;
    { The following piece of code is a copy of module 991: }
  page_contents ← *empty*; *page_tail* ← *page_head*; { *link*(*page_head*) ← *null*; }
  last_glue ← *max_halfword*; *last_penalty* ← 0; *last_kern* ← 0; *page_depth* ← 0; *page_max_depth* ← 0;

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

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

217. Conversely, when TEX 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;

218. Here is a procedure that displays what TEX 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_mode(m);
    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); print_char(",");
        print_int((nest[p].pg_field div ´200000) mod ´100); print_char(")");
        end;
    if nest[p].ml_field < 0 then print("␣(\output␣routine)");
    if p = 0 then
      begin ⟨Show the status of the current page 989⟩;
      if link(contrib_head) ≠ null then print_nl("###␣recent␣contributions:");
      end;
    show_box(link(nest[p].head_field)); ⟨Show the auxiliary field, a 219⟩;
    end;
  end;
```

219. ⟨Show the auxiliary field, a 219⟩ ≡

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

This code is used in section 218.

220. The table of equivalents. Now that we have studied the data structures for TEX'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 TEX 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 TEX's grouping mechanism. There are six parts to *eqtb*:

1) $eqtb[active\_base \mathrel{..} (hash\_base - 1)]$ holds the current equivalents of single-character control sequences.
2) $eqtb[hash\_base \mathrel{..} (glue\_base - 1)]$ holds the current equivalents of multiletter control sequences.
3) $eqtb[glue\_base \mathrel{..} (local\_base - 1)]$ holds the current equivalents of glue parameters like the current baselineskip.
4) $eqtb[local\_base \mathrel{..} (int\_base - 1)]$ 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. Additionally region 4 contains the table with MLTEX's character substitution definitions.
5) $eqtb[int\_base \mathrel{..} (dimen\_base - 1)]$ holds the current equivalents of fullword integer parameters like the current hyphenation penalty.
6) $eqtb[dimen\_base \mathrel{..} eqtb\_size]$ 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.

221. 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 TEX 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 $eq\_level\_field(\#) \equiv \#.hh.b1$
define $eq\_type\_field(\#) \equiv \#.hh.b0$
define $equiv\_field(\#) \equiv \#.hh.rh$
define $eq\_level(\#) \equiv eq\_level\_field(eqtb[\#])$ { level of definition }
define $eq\_type(\#) \equiv eq\_type\_field(eqtb[\#])$ { command code for equivalent }
define $equiv(\#) \equiv equiv\_field(eqtb[\#])$ { equivalent value }
define $level\_zero = min\_quarterword$ { level for undefined quantities }
define $level\_one = level\_zero + 1$ { outermost level for defined quantities }

222. 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 65536 equivalents for "active characters" that act as control sequences, followed by 65536 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 $single\_base = active\_base + 65536$ { equivalents of one-character control sequences }
define $null\_cs = single\_base + 65536$ { 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 $frozen\_special = frozen\_control\_sequence + 10$ { permanent '`\special`' }
define $frozen\_null\_font = frozen\_control\_sequence + 11$ { permanent '`\nullfont`' }
define $font\_id\_base = frozen\_null\_font - font\_base$
{ begins table of 257 permanent English font identifiers }
define $cfont\_id\_base = font\_id\_base + font\_max\_limit + 1$
{ TCW: begins table of 'font_max_limit' permanent CJK font identifiers }
define $cfont\_max\_limit = font\_max\_limit$
define $cface\_id\_base = cfont\_id\_base + cfont\_max\_limit + 1$
{ TCW: begins table of 257 permanent Chinese face identifiers }
define $undefined\_control\_sequence = cface\_id\_base + 257$ { dummy location }
define $glue\_base = undefined\_control\_sequence + 1$ { beginning of region 3 }

⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
$eq\_type(undefined\_control\_sequence) \leftarrow undefined\_cs$; $equiv(undefined\_control\_sequence) \leftarrow null$;
$eq\_level(undefined\_control\_sequence) \leftarrow level\_zero$;
for $k \leftarrow active\_base$ **to** $eqtb\_top$ **do** $eqtb[k] \leftarrow eqtb[undefined\_control\_sequence]$;

223. 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 2 223 ⟩ ≡
begin $sprint\_cs(n)$; $print\_char("=")$; $print\_cmd\_chr(eq\_type(n), equiv(n))$;
if $eq\_type(n) \ge call$ **then**
  begin $print\_char(":")$; $show\_token\_list(link(equiv(n)), null, 32)$;
  end;
end

This code is used in section 252.

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

⟨ Current *mem* equivalent of glue parameter number *n* 224 ⟩ ≡
glue_par(*n*)

This code is used in sections 152 and 154.

225. 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* 225 ⟩ ≡

```
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 179.

226. 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 226 ⟩ ≡
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 230, 238, 248, 265, 334, 379, 387, 414, 419, 471, 490, 494, 556, 783, 986, 1055, 1061, 1074, 1091, 1110, 1117, 1144, 1159, 1172, 1181, 1191, 1211, 1222, 1225, 1233, 1253, 1257, 1265, 1275, 1280, 1289, 1294, 1347, 1414, 1425, 1429, 1442, 1465, 1469, 1475, 1520, 1533, 1542, 1548, 1556, 1561, and 1565.

This code is used in section 1339.

227. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ ≡
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 231, 239, 249, 266, 335, 380, 388, 415, 420, 472, 491, 495, 784, 987, 1056, 1062, 1075, 1092, 1111, 1118, 1146, 1160, 1173, 1182, 1192, 1212, 1223, 1226, 1234, 1254, 1258, 1264, 1266, 1276, 1281, 1290, 1295, 1298, 1349, 1426, 1443, 1446, 1466, 1470, 1476, 1515, 1521, 1543, 1549, 1557, 1562, and 1566.

This code is used in section 298.

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

⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
equiv(*glue_base*) ← *zero_glue*; *eq_level*(*glue_base*) ← *level_one*; *eq_type*(*glue_base*) ← *glue_ref*;
for k ← *glue_base* + 1 **to** *local_base* − 1 **do** *eqtb*[k] ← *eqtb*[*glue_base*];
glue_ref_count(*zero_glue*) ← *glue_ref_count*(*zero_glue*) + *local_base* − *glue_base*;

229. ⟨Show equivalent n, in region 3 229⟩ ≡

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 252.

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

TCW: Define *cur_cfont_loc* for two-byte char and the macro *cur_cfont*.

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 *toks_base* = *local_base* + 10 { table of 256 token list registers }
define *box_base* = *toks_base* + 256 { table of 256 box registers }
define *cur_font_loc* = *box_base* + 256 { internal font number outside math mode }
define *cur_cface_loc* = *cur_font_loc* + 1 { TCW: internal chinese font number outside math mode }
define *cur_cfont_loc* = *cur_cface_loc* + 1 { TCW: internal chinese font number outside math mode }
define *ectbl_cface_num_base* = *cur_cfont_loc* + 1
{ TCW: table of 257 CJK face numbers matched with TeX face }
define *max_cface* = 256 { maximal CJK font faces number }
define *font_matching_table_base* = *ectbl_cface_num_base* + *max_cface* + 1 { table of font matches }
define *math_font_base* = *font_matching_table_base* + *font_max_limit* + 1
{ table of 48 math font numbers }
define *cat_code_base* = *math_font_base* + 48 { TCW: table of 65536 command codes (the "catcodes") }
define *pux_cat_code_base* = *cat_code_base* + 256
define *pux_type_code_base* = *cat_code_base* + 65536 { TCW: table of 65536 type codes }
define *lc_code_base* = *pux_type_code_base* + 65536 { 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 *pux_local_names_base* = *sf_code_base* + 256 { TCW: table of 256 CJK name mappings. }
define *math_code_base* = *pux_local_names_base* + 256 { table of 256 math mode mappings }
define *char_sub_code_base* = *math_code_base* + 256 { table of character substitutions }
define *int_base* = *char_sub_code_base* + 256 { beginning of region 5 }

define *par_shape_ptr* ≡ *equiv*(*par_shape_loc*)
define *output_routine* ≡ *equiv*(*output_routine_loc*)
define *every_par* ≡ *equiv*(*every_par_loc*)
define *every_math* ≡ *equiv*(*every_math_loc*)
define *every_display* ≡ *equiv*(*every_display_loc*)
define *every_hbox* ≡ *equiv*(*every_hbox_loc*)
define *every_vbox* ≡ *equiv*(*every_vbox_loc*)
define *every_job* ≡ *equiv*(*every_job_loc*)
define *every_cr* ≡ *equiv*(*every_cr_loc*)
define *err_help* ≡ *equiv*(*err_help_loc*)
define *toks*(#) ≡ *equiv*(*toks_base* + #)
define *box*(#) ≡ *equiv*(*box_base* + #)
define *cur_font* ≡ *equiv*(*cur_font_loc*)
define *cur_cface* ≡ *equiv*(*cur_cface_loc*) { TCW }
define *cur_cfont* ≡ *equiv*(*cur_cfont_loc*) { TCW }
define *ectbl_cface_num*(#) ≡ *equiv*(*ectbl_cface_num_base* + (#)) { TCW }

define *font_matching_table*(#) ≡ *equiv*(*font_matching_table_base* + ((#) − *font_base*)) { TCW }
define *fam_fnt*(#) ≡ *equiv*(*math_font_base* + #)
define *cat_code*(#) ≡ *equiv*(*cat_code_base* + #)
define *type_code*(#) ≡ *equiv*(*pux_type_code_base* + #)
define *local_names*(#) ≡ *equiv*(*pux_local_names_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* }
define *char_sub_code*(#) ≡ *equiv*(*char_sub_code_base* + #)
{ Note: *char_sub_code*(c) is the true substitution info plus *min_halfword* }

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("output", *assign_toks*, *output_routine_loc*); *primitive*("everypar", *assign_toks*, *every_par_loc*);
primitive("everymath", *assign_toks*, *every_math_loc*);
primitive("everydisplay", *assign_toks*, *every_display_loc*);
primitive("everyhbox", *assign_toks*, *every_hbox_loc*); *primitive*("everyvbox", *assign_toks*, *every_vbox_loc*);
primitive("everyjob", *assign_toks*, *every_job_loc*); *primitive*("everycr", *assign_toks*, *every_cr_loc*);
primitive("errhelp", *assign_toks*, *err_help_loc*);

231. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
assign_toks: **if** *chr_code* ≥ *toks_base* **then**
begin *print_esc*("toks"); *print_int*(*chr_code* − *toks_base*);
end
else case *chr_code* **of**
output_routine_loc: *print_esc*("output");
every_par_loc: *print_esc*("everypar");
every_math_loc: *print_esc*("everymath");
every_display_loc: *print_esc*("everydisplay");
every_hbox_loc: *print_esc*("everyhbox");
every_vbox_loc: *print_esc*("everyvbox");
every_job_loc: *print_esc*("everyjob");
every_cr_loc: *print_esc*("everycr");
othercases *print_esc*("errhelp")
endcases;

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

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

TCW: Add *null_cfont* and initialization for *cur_font*.

define *null_font* ≡ *font_base*
define *null_cfont* ≡ *cfont_base*
define *default_cfont* ≡ *null_cfont* + 1
define *var_code* ≡ ´70000 { math code meaning "use the current family" }

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

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

```
if n = par_shape_loc then
  begin print_esc("parshape"); print_char("=");
  if par_shape_ptr = null then print_char("0")
  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) ≠ 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) ≠ 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 ← 0; breadth_max ← 1; show_node_list(equiv(n));
          end;
        end
      else if n < cat_code_base then ⟨ Show the font identifier in eqtb[n] 234 ⟩
        else ⟨ Show the halfword code in eqtb[n] 235 ⟩
```

This code is used in section 252.

234. ⟨ Show the font identifier in $eqtb[n]$ 234 ⟩ ≡

```
begin if n = cur_font_loc then print("current␣font")
else if n = cur_cface_loc then print("current␣cface")
  else if n = cur_cfont_loc then print("current␣cfont")
    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 233.

235. ⟨Show the halfword code in *eqtb*[n] 235⟩ ≡

```
if n < math_code_base then
  begin if n < pux_type_code_base then
    begin if n < pux_cat_code_base then print_esc("catcode")
    else print_esc("PUXcatcode");
    print_int(n − cat_code_base);
    end
  else if n < lc_code_base then
      begin print_esc("PUXtypecode"); print_int(n − pux_type_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 if n < pux_local_names_base then
            begin print_esc("sfcode"); print_int(n − sf_code_base);
            end
          else begin print_esc("PUXlocalnames"); print_int(n − pux_local_names_base);
            end;
  print_char("=");
  if n ≥ pux_local_names_base then
    if n < 256 then print_char(equiv(n))
    else print_wchar(equiv(n))
  else 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 233.

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

define *pretolerance_code* = 0 { badness tolerance before hyphenation }
define *tolerance_code* = 1 { badness tolerance after hyphenation }
define *line_penalty_code* = 2 { added to the badness of every line }
define *hyphen_penalty_code* = 3 { penalty for break after discretionary hyphen }
define *ex_hyphen_penalty_code* = 4 { penalty for break after explicit hyphen }
define *club_penalty_code* = 5 { penalty for creating a club line }
define *widow_penalty_code* = 6 { penalty for creating a widow line }
define *display_widow_penalty_code* = 7 { ditto, just before a display }
define *broken_penalty_code* = 8 { penalty for breaking a page at a broken line }
define *bin_op_penalty_code* = 9 { penalty for breaking after a binary operation }
define *rel_penalty_code* = 10 { penalty for breaking after a relation }
define *pre_display_penalty_code* = 11 { penalty for breaking just before a displayed formula }
define *post_display_penalty_code* = 12 { penalty for breaking just after a displayed formula }
define *inter_line_penalty_code* = 13 { additional penalty between lines }
define *double_hyphen_demerits_code* = 14 { demerits for double hyphen break }
define *final_hyphen_demerits_code* = 15 { demerits for final hyphen break }
define *adj_demerits_code* = 16 { demerits for adjacent incompatible lines }
define *mag_code* = 17 { magnification ratio }
define *delimiter_factor_code* = 18 { ratio for variable-size delimiters }
define *looseness_code* = 19 { change in number of lines for a paragraph }
define *time_code* = 20 { current time of day }
define *day_code* = 21 { current day of the month }
define *month_code* = 22 { current month of the year }
define *year_code* = 23 { current year of our Lord }
define *show_box_breadth_code* = 24 { nodes per level in *show_box* }
define *show_box_depth_code* = 25 { maximum level in *show_box* }
define *hbadness_code* = 26 { hboxes exceeding this badness will be shown by *hpack* }
define *vbadness_code* = 27 { vboxes exceeding this badness will be shown by *vpack* }
define *pausing_code* = 28 { pause after each line is read from a file }
define *tracing_online_code* = 29 { show diagnostic output on terminal }
define *tracing_macros_code* = 30 { show macros as they are being expanded }
define *tracing_stats_code* = 31 { show memory usage if TEX knows it }
define *tracing_paragraphs_code* = 32 { show line-break calculations }
define *tracing_pages_code* = 33 { show page-break calculations }
define *tracing_output_code* = 34 { show boxes when they are shipped out }
define *tracing_lost_chars_code* = 35 { show characters that aren't in the font }
define *tracing_commands_code* = 36 { show command codes at *big_switch* }
define *tracing_restores_code* = 37 { show equivalents when they are restored }
define *uc_hyph_code* = 38 { hyphenate words beginning with a capital letter }
define *output_penalty_code* = 39 { penalty found at current page break }
define *max_dead_cycles_code* = 40 { bound on consecutive dead cycles of output }
define *hang_after_code* = 41 { hanging indentation changes after this many lines }
define *floating_penalty_code* = 42 { penalty for insertions heldover after a split }
define *global_defs_code* = 43 { override `\global` specifications }
define *cur_fam_code* = 44 { current family }
define *escape_char_code* = 45 { escape character for token output }
define *default_hyphen_char_code* = 46 { value of `\hyphenchar` when a font is loaded }

define *default_skew_char_code* = 47 { value of `\skewchar` when a font is loaded }
define *end_line_char_code* = 48 { character placed at the right end of the buffer }
define *new_line_char_code* = 49 { character that prints as *print_ln* }
define *language_code* = 50 { current hyphenation table }
define *left_hyphen_min_code* = 51 { minimum left hyphenation fragment size }
define *right_hyphen_min_code* = 52 { minimum right hyphenation fragment size }
define *holding_inserts_code* = 53 { do not remove insertion nodes from `\box255` }
define *error_context_lines_code* = 54 { maximum intermediate line pairs shown }
define *puxg_rotate_ctext_code* = 55
define *puxg_cface_depth_code* = 56
define *pux_xspace_code* = 57
define *pux_wcharother_code* = 58
define *pux_CJKinput_code* = 59
define *pux_charset_code* = 60
define *pux_default_cface_code* = 61
define *pux_digit_num_code* = 62 { number of digits of the splitted number }
define *pux_sign_code* = 63 { sign of the splitted number }
define *pux_digit_base* = 64 { 10 digits of splitted number }
define *tex_int_pars* = 74 { total number of TeX's integer parameters }
define *web2c_int_base* = *tex_int_pars* { base for web2c's integer parameters }
define *char_sub_def_min_code* = *web2c_int_base* { smallest value in the charsubdef list }
define *char_sub_def_max_code* = *web2c_int_base* + 1 { largest value in the charsubdef list }
define *tracing_char_sub_def_code* = *web2c_int_base* + 2 { traces changes to a charsubdef def }
define *web2c_int_pars* = *web2c_int_base* + 3 { total number of web2c's integer parameters }
define *int_pars* = *web2c_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 *puxg_rotate_ctext* ≡ *int_par*(*puxg_rotate_ctext_code*)
define *puxg_cface_depth* ≡ *int_par*(*puxg_cface_depth_code*)
define *pux_xspace* ≡ *int_par*(*pux_xspace_code*)
define *pux_wcharother* ≡ *int_par*(*pux_wcharother_code*)
define *pux_CJKinput* ≡ *int_par*(*pux_CJKinput_code*)
define *pux_charset* ≡ *int_par*(*pux_charset_code*)
define *pux_default_cface* ≡ *int_par*(*pux_default_cface_code*)
define *pux_digit_num* ≡ *int_par*(*pux_digit_num_code*)
define *pux_num_sign* ≡ *int_par*(*pux_sign_code*)
define *pux_nth_digit*(#) ≡ *int_par*(*pux_digit_base* + #)
define *default_csp* = 50
define *default_cesp* = 150
define *default_depth* = 200

define *char_sub_def_min* ≡ *int_par*(*char_sub_def_min_code*)
define *char_sub_def_max* ≡ *int_par*(*char_sub_def_max_code*)
define *tracing_char_sub_def* ≡ *int_par*(*tracing_char_sub_def_code*)

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

This code is used in section 198.

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

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

new_line_char_code: *print_esc*("newlinechar");
language_code: *print_esc*("language");
left_hyphen_min_code: *print_esc*("lefthyphenmin");
right_hyphen_min_code: *print_esc*("righthyphenmin");
holding_inserts_code: *print_esc*("holdinginserts");
error_context_lines_code: *print_esc*("errorcontextlines");
char_sub_def_min_code: *print_esc*("charsubdefmin");
char_sub_def_max_code: *print_esc*("charsubdefmax");
tracing_char_sub_def_code: *print_esc*("tracingcharsubdef");
pux_xspace_code: *print_esc*("puxXspace");
pux_wcharother_code: *print_esc*("puxCJKcharOther");
pux_CJKinput_code: *print_esc*("puxCJKinput");
pux_charset_code: *print_esc*("puxCharSet");
puxg_rotate_ctext_code: *print_esc*("puxgRotateCtext");
puxg_cface_depth_code: *print_esc*("puxgCfaceDepth");
othercases *print*("[unknown␣integer␣parameter!]")
endcases;
end;

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

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
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_hyph_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*);
if *mltex_p* **then**
begin *mltex_enabled_p* ← *true*; { enable character substitution }
if *false* **then** { remove the if-clause to enable \charsubdefmin }
primitive("charsubdefmin", *assign_int*, *int_base* + *char_sub_def_min_code*);
primitive("charsubdefmax", *assign_int*, *int_base* + *char_sub_def_max_code*);
primitive("tracingcharsubdef", *assign_int*, *int_base* + *tracing_char_sub_def_code*);
end;

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

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

⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
for *k* ← *int_base* **to** *del_code_base* − 1 **do** *eqtb*[*k*].*int* ← 0;
char_sub_def_min ← 256; *char_sub_def_max* ← −1; { allow \charsubdef for char 0 }
{ *tracing_char_sub_def* ← 0 is already done }
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 }
puxg_cface_depth ← *default_depth*; *pux_CJKinput* ← 1;

241. The following procedure, which is called just before TEX initializes its input and output, establishes the initial values of the date and time. It calls a macro-defined *date_and_time* routine. *date_and_time* in turn is a C macro, which calls *get_date_and_time*, passing it the addresses of the day, month, etc., so they can be set by the routine. *get_date_and_time* also sets up interrupt catching if that is conditionally compiled in the C code.

define *fix_date_and_time* ≡ *date_and_time*(*time*, *day*, *month*, *year*)

242. ⟨ Show equivalent *n*, in region 5 242 ⟩ ≡
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 252.

243. ⟨ Set variable *c* to the current escape character 243 ⟩ ≡
c ← *escape_char*

This code is used in section 63.

244. ⟨ Character s is the current new-line character 244 ⟩ ≡
$s = new\_line\_char$

This code is used in sections 58 and 59.

245. 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 $old\_setting \leftarrow selector$;
  if $(tracing\_online \le 0) \land (selector = term\_and\_log)$ **then**
    begin $decr(selector)$;
    if $history = spotless$ **then** $history \leftarrow warning\_issued$;
    end;
  end;

procedure *end_diagnostic*(*blank_line* : *boolean*); { restore proper conditions after tracing }
  begin *print_nl*("");
  if *blank_line* **then** *print_ln*;
  $selector \leftarrow old\_setting$;
  end;

246. Of course we had better declare another global variable, if the previous routines are going to work.

⟨ Global variables 13 ⟩ +≡
$old\_setting$: $0 \,.\,.\, max\_selector$;

247. 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 *dimen_pars* = 21 { 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*(#) ≡ *eqtb*[*scaled_base* + #].*sc*
define *dimen_par*(#) ≡ *eqtb*[*dimen_base* + #].*sc* { a scaled quantity }
define *par_indent* ≡ *dimen_par*(*par_indent_code*)
define *math_surround* ≡ *dimen_par*(*math_surround_code*)
define *line_skip_limit* ≡ *dimen_par*(*line_skip_limit_code*)
define *hsize* ≡ *dimen_par*(*hsize_code*)
define *vsize* ≡ *dimen_par*(*vsize_code*)
define *max_depth* ≡ *dimen_par*(*max_depth_code*)
define *split_max_depth* ≡ *dimen_par*(*split_max_depth_code*)
define *box_max_depth* ≡ *dimen_par*(*box_max_depth_code*)
define *hfuzz* ≡ *dimen_par*(*hfuzz_code*)
define *vfuzz* ≡ *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*)

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");
  vsize_code: print_esc("vsize");
  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("delimitershortfall");
  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");
  othercases print("[unknown␣dimen␣parameter!]")
  endcases;
  end;
```

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

```
  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);
```

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

```
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;
```

250. ⟨Initialize table entries (done by INITEX only) 164⟩ +≡
for $k \leftarrow$ *dimen_base* **to** *eqtb_size* **do** *eqtb*[k].*sc* $\leftarrow 0$;

251. ⟨Show equivalent n, in region 6 251⟩ ≡
begin if $n <$ *scaled_base* **then** *print_length_param*($n -$ *dimen_base*)
else begin *print_esc*("dimen"); *print_int*($n -$ *scaled_base*);
end;
print_char("="); *print_scaled*(*eqtb*[n].*sc*); *print*("pt");
end

This code is used in section 252.

252. Here is a procedure that displays the contents of *eqtb*[n] symbolically.

⟨Declare the procedure called *print_cmd_chr* 298⟩
stat procedure *show_eqtb*(n : *pointer*);
begin if $n <$ *active_base* **then** *print_char*("?") { this can't happen }
else if ($n <$ *glue_base*) $\lor$ (($n >$ *eqtb_size*) $\land$ ($n \leq$ *eqtb_top*)) **then** ⟨Show equivalent n, in region 1 or 2 223⟩
else if $n <$ *local_base* **then** ⟨Show equivalent n, in region 3 229⟩
else if $n <$ *int_base* **then** ⟨Show equivalent n, in region 4 233⟩
else if $n <$ *dimen_base* **then** ⟨Show equivalent n, in region 5 242⟩
else if $n \leq$ *eqtb_size* **then** ⟨Show equivalent n, in region 6 251⟩
else *print_char*("?"); { this can't happen either }
end;
tats

253. 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⟩ +≡
zeqtb: ↑*memory_word*;
xeq_level: **array** [*int_base* . . *eqtb_size*] **of** *quarterword*;

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

255. 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 255⟩ ≡
for $q \leftarrow$ *active_base* **to** *box_base* + 255 **do**
begin if *equiv*(q) = p **then**
begin *print_nl*("EQUIV("); *print_int*(q); *print_char*(")");
end;
end

This code is used in section 172.

256. 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 $p \geq hash\_used$ 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: ↑*two_halves*; { the hash table }
yhash: ↑*two_halves*; { auxiliary pointer for freeing hash }
hash_used: *pointer*; { allocation pointer for *hash* }
hash_extra: *pointer*; { *hash_extra* = *hash* above *eqtb_size* }
hash_top: *pointer*; { maximum of the hash array }
eqtb_top: *pointer*; { maximum of the *eqtb* }
hash_high: *pointer*; { pointer to next high hash location }
no_new_control_sequence: *boolean*; { are new identifiers legal? }
cs_count: *integer*; { total number of known identifiers }

257. ⟨ Set initial values of key variables 21 ⟩ +≡
no_new_control_sequence ← *true*; { new identifiers are usually forbidden }

258. ⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
hash_used ← *frozen_control_sequence*; { nothing is used }
hash_high ← 0; *cs_count* ← 0; *eq_type*(*frozen_dont_expand*) ← *dont_expand*;
text(*frozen_dont_expand*) ← `"notexpanded:"`;

259. Here is the subroutine that searches the hash table for an identifier that matches a given string of length $l > 1$ appearing in $buffer[j \mathrel{..} (j + l - 1)]$. If the identifier is found, the corresponding hash table address is returned. Otherwise, if the global variable *no_new_control_sequence* is *true*, the dummy address *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 261⟩;
  p ← h + hash_base;  { we start searching here; note that 0 ≤ h < 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 260⟩;
      goto found;
      end;
    p ← next(p);
    end;
found: id_lookup ← p;
  end;
```

260. ⟨Insert a new control sequence after p, then make p point to it 260⟩ ≡

```
  begin if text(p) > 0 then
    begin if hash_high < hash_extra then
      begin incr(hash_high); next(p) ← hash_high + eqtb_size; p ← hash_high + eqtb_size;
      end
    else begin repeat if hash_is_full then overflow("hash␣size", hash_size + hash_extra);
        decr(hash_used);
      until text(hash_used) = 0;  { search for an empty location in hash }
      next(p) ← hash_used; p ← hash_used;
      end;
    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 259.

261. 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 261⟩ ≡
```
  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 259.

262. 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 ≥ single_base then
      if p = null_cs then
        begin print_esc("csname"); print_esc("endcsname");
        end
      else begin print_esc(p − single_base);
        if get_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 ≥ undefined_control_sequence) ∧ (p ≤ eqtb_size)) ∨ (p > eqtb_top) then
      print_esc("IMPOSSIBLE.")
    else if (text(p) ≥ str_ptr) then print_esc("NONEXISTENT.")
      else begin print_esc(text(p)); print_char("␣");
        end;
  end;
```

263. 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 print_esc(text(p));
  end;
```

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

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

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

⟨Put each of TEX's primitives into the hash table 226⟩ +≡
primitive("␣", *ex_space*, 0);
primitive("/", *ital_corr*, 0);
primitive("accent", *accent*, 0);
primitive("advance", *advance*, 0);
primitive("afterassignment", *after_assignment*, 0);
primitive("aftergroup", *after_group*, 0);
primitive("begingroup", *begin_group*, 0);
primitive("char", *char_num*, 0);
primitive("csname", *cs_name*, 0);
primitive("delimiter", *delim_num*, 0);
primitive("divide", *divide*, 0);
primitive("endcsname", *end_cs_name*, 0);
primitive("endgroup", *end_group*, 0); *text*(*frozen_end_group*) ← "endgroup";
eqtb[*frozen_end_group*] ← *eqtb*[*cur_val*];
primitive("expandafter", *expand_after*, 0);
primitive("font", *def_font*, 0);
primitive("fontdimen", *assign_font_dimen*, 0);
primitive("halign", *halign*, 0);
primitive("hrule", *hrule*, 0);
primitive("ignorespaces", *ignore_spaces*, 0);
primitive("insert", *insert*, 0);
primitive("mark", *mark*, 0);
primitive("mathaccent", *math_accent*, 0);
primitive("mathchar", *math_char_num*, 0);
primitive("mathchoice", *math_choice*, 0);
primitive("multiply", *multiply*, 0);
primitive("noalign", *no_align*, 0);
primitive("noboundary", *no_boundary*, 0);
primitive("noexpand", *no_expand*, 0);
primitive("nonscript", *non_script*, 0);
primitive("omit", *omit*, 0);
primitive("parshape", *set_shape*, 0);
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*, 0);
primitive("vadjust", *vadjust*, 0);
primitive("valign", *valign*, 0);
primitive("vcenter", *vcenter*, 0);
primitive("vrule", *vrule*, 0);

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

⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
accent: *print_esc*("accent");
advance: *print_esc*("advance");
after_assignment: *print_esc*("afterassignment");
after_group: *print_esc*("aftergroup");
assign_font_dimen: *print_esc*("fontdimen");
begin_group: *print_esc*("begingroup");
break_penalty: *print_esc*("penalty");
char_num: *print_esc*("char");
cs_name: *print_esc*("csname");
def_font: *print_esc*("font");
pux_font_match: *print_esc*("PUXfontmatch"); { TCW }
pux_set_cface: *print_esc*("cface"); { TCW }
pux_range_catcode: *print_esc*("PUXrangecatcode"); { TCW }
pux_range_type_code: *print_esc*("PUXrangetypecode"); { TCW }
pux_split_number: *print_esc*("PUXsplitnumber"); { TCW }
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: *print_esc*("expandafter");
halign: *print_esc*("halign");
hrule: *print_esc*("hrule");
ignore_spaces: *print_esc*("ignorespaces");
insert: *print_esc*("insert");
ital_corr: *print_esc*("/");
mark: *print_esc*("mark");
math_accent: *print_esc*("mathaccent");
math_char_num: *print_esc*("mathchar");
math_choice: *print_esc*("mathchoice");
multiply: *print_esc*("multiply");
no_align: *print_esc*("noalign");
no_boundary: *print_esc*("noboundary");
no_expand: *print_esc*("noexpand");
non_script: *print_esc*("nonscript");
omit: *print_esc*("omit");
radical: *print_esc*("radical");
read_to_cs: *print_esc*("read");
relax: *print_esc*("relax");
set_box: *print_esc*("setbox");
set_prev_graf: *print_esc*("prevgraf");
set_shape: *print_esc*("parshape");
the: *print_esc*("the");
toks_register: *print_esc*("toks");
vadjust: *print_esc*("vadjust");
valign: *print_esc*("valign");
vcenter: *print_esc*("vcenter");
vrule: *print_esc*("vrule");

267. 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:

⟨ Print the font identifier for *font*(*p*) 267 ⟩ ≡
print_esc(*font_id_text*(*font*(*p*)))

This code is used in sections 174 and 176.

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

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

1) If *save_type*(p) = *restore_old_value*, then *save_index*(p) is a location in *eqtb* whose current value should be destroyed at the end of the current group and replaced by *save_stack*[$p - 1$]. Furthermore if *save_index*(p) ≥ *int_base*, then *save_level*(p) should replace the corresponding entry in *xeq_level*.
2) If *save_type*(p) = *restore_zero*, then *save_index*(p) is a location in *eqtb* whose current value should be destroyed at the end of the current group, when it should be replaced by the current value of *eqtb*[*undefined_control_sequence*].
3) If *save_type*(p) = *insert_token*, then *save_index*(p) is a token that should be inserted into TEX's input when the current group ends.
4) If *save_type*(p) = *level_boundary*, then *save_level*(p) is a code explaining what kind of group we were previously in, and *save_index*(p) points to the level boundary word at the bottom of the entries for that group.

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 }

269. 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 }

270. 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*.

271. ⟨ Global variables 13 ⟩ +≡
save_stack: ↑*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 }

272. 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;

273. The following macro is used to test if there is room for up to six 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* − 6 **then** *overflow*("save␣size", *save_size*);
    end

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

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

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

define *saved*(#) ≡ *save_stack*[*save_ptr* + #].*int*

procedure *new_save_level*(*c* : *group_code*); { begin a new level of grouping }
  begin *check_full_save_stack*; *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*; *incr*(*cur_level*); *incr*(*save_ptr*); *cur_group* ← *c*;
  end;

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

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

276. 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;

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

procedure *eq_define*(*p* : *pointer*; *t* : *quarterword*; *e* : *halfword*); { new data for *eqtb* }
  begin 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*;
  end;

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

procedure *eq_word_define*(*p* : *pointer*; *w* : *integer*);
  begin if *xeq_level*[*p*] ≠ *cur_level* **then**
    begin *eq_save*(*p*, *xeq_level*[*p*]); *xeq_level*[*p*] ← *cur_level*;
    end;
  eqtb[*p*].*int* ← *w*;
  end;

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

procedure *geq_define*(*p* : *pointer*; *t* : *quarterword*; *e* : *halfword*); { global *eq_define* }
  begin *eq_destroy*(*eqtb*[*p*]); *eq_level*(*p*) ← *level_one*; *eq_type*(*p*) ← *t*; *equiv*(*p*) ← *e*;
  end;

procedure *geq_word_define*(*p* : *pointer*; *w* : *integer*); { global *eq_word_define* }
  begin *eqtb*[*p*].*int* ← *w*; *xeq_level*[*p*] ← *level_one*;
  end;

280. 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;

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

⟨ Declare the procedure called *restore_trace* 284 ⟩
procedure *back_input*; *forward*;
procedure *unsave*; { pops the top level off the save stack }
  label *done*;
  var *p*: *pointer*; { position to be restored }
    l: *quarterword*; { saved level, if in fullword regions of *eqtb* }
    t: *halfword*; { saved value of *cur_tok* }
  begin if *cur_level* > *level_one* **then**
    begin *decr*(*cur_level*); ⟨ Clear off top level from *save_stack* 282 ⟩;
    end
  else *confusion*("curlevel"); { *unsave* is not used when *cur_group* = *bottom_level* }
  end;

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

```
loop begin decr(save_ptr);
  if save_type(save_ptr) = level_boundary then goto done;
  p ← save_index(save_ptr);
  if save_type(save_ptr) = insert_token then ⟨Insert token p into TEX's input 326⟩
  else begin if save_type(save_ptr) = restore_old_value then
      begin l ← save_level(save_ptr); decr(save_ptr);
      end
    else save_stack[save_ptr] ← eqtb[undefined_control_sequence];
    ⟨Store save_stack[save_ptr] in eqtb[p], unless eqtb[p] holds a global value 283⟩;
    end;
  end;
done: cur_group ← save_level(save_ptr); cur_boundary ← save_index(save_ptr)
```

This code is used in section 281.

283. A global definition, which sets the level to *level_one*, will not be undone by *unsave*. 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.

⟨Store *save_stack*[*save_ptr*] in *eqtb*[*p*], unless *eqtb*[*p*] holds a global value 283⟩ ≡

```
if (p < int_base) ∨ (p > eqtb_size) 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] ← 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] ≠ level_one then
    begin eqtb[p] ← save_stack[save_ptr]; xeq_level[p] ← 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 282.

284. ⟨Declare the procedure called *restore_trace* 284⟩ ≡

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

This code is used in section 281.

285. 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.

```
⟨Search save_stack for equivalents that point to p 285⟩ ≡
  if save_ptr > 0 then
    for q ← 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 172.

286. 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.

```
⟨Global variables 13⟩ +≡
mag_set: integer;   { if nonzero, this magnification should be used henceforth }
```

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

```
  mag_set ← 0;
```

288. The *prepare_mag* subroutine is called whenever TEX wants to use *mag* for magnification.

```
procedure prepare_mag;
  begin if (mag_set > 0) ∧ (mag ≠ 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 ← mag_set }
    end;
  if (mag ≤ 0) ∨ (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 ← mag;
  end;
```

289. Token lists. A TEX token is either a character or a control sequence, and it is represented internally in one of two ways: (1) A character whose ASCII code number is c and whose command code is m is represented as the number $2^{16}m + c$; the command code is in the range $1 \le m \le 14$. (2) A control sequence whose *eqtb* address is p is represented as the number $cs\_token\_flag + p$. Here $cs\_token\_flag = 2^{20} - 1$ is larger than $2^{16}m + c$, yet it is small enough that $cs\_token\_flag + p < max\_halfword$; thus, a token fits comfortably in a halfword.

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

define $cs\_token\_flag \equiv$ ˝`FFFFF` { amount added to the *eqtb* location in a token that stands for a control sequence; is a multiple of 65536, less 1 }
define $left\_brace\_token \equiv$ ˝`10000` { $2^{16} \cdot left\_brace$ }
define $left\_brace\_limit \equiv$ ˝`20000` { $2^{16} \cdot (left\_brace + 1)$ }
define $right\_brace\_token \equiv$ ˝`20000` { $2^{16} \cdot right\_brace$ }
define $right\_brace\_limit \equiv$ ˝`30000` { $2^{16} \cdot (right\_brace + 1)$ }
define $math\_shift\_token \equiv$ ˝`30000` { $2^{16} \cdot math\_shift$ }
define $tab\_token \equiv$ ˝`40000` { $2^{16} \cdot tab\_mark$ }
define $out\_param\_token \equiv$ ˝`50000` { $2^{16} \cdot out\_param$ }
define $space\_token \equiv$ ˝`A0020` { $2^{16} \cdot spacer +$ "␣" }
define $letter\_token \equiv$ ˝`B0000` { $2^{16} \cdot letter$ }
define $other\_token \equiv$ ˝`C0000` { $2^{16} \cdot other\_char$ }
define $match\_token \equiv$ ˝`D0000` { $2^{16} \cdot match$ }
define $end\_match\_token \equiv$ ˝`E0000` { $2^{16} \cdot end\_match$ }

290. ⟨ Check the "constant" values for consistency 14 ⟩ +≡
if $cs\_token\_flag + eqtb\_size + hash\_extra > max\_halfword$ **then** $bad \leftarrow 21$;
if $(hash\_offset < 0) \lor (hash\_offset > hash\_base)$ **then** $bad \leftarrow 42$;

291. 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 the final right brace of an output routine is included at the end of its token list.

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.

292. 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* 292⟩ ≡
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 ($p \neq$ *null*) ∧ (*tally* $< l$) **do**
    begin if $p = q$ **then** ⟨Do magic computation 320⟩;
    ⟨Display token p, and **return** if there are problems 293⟩;
    p ← *link*(p);
    end;
  if $p \neq$ *null* **then** *print_esc*(`"ETC."`);
exit: **end**;

This code is used in section 119.

293. ⟨Display token p, and **return** if there are problems 293⟩ ≡
  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** ″`10000`; c ← *info*(p) **mod** ″`10000`;
    if *info*(p) < 0 **then** *print_esc*(`"BAD."`)
    else ⟨Display the token (m, c) 294⟩;
    end

This code is used in section 292.

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

⟨ Display the token (m, c) 294 ⟩ ≡

```
case m of
letter, other_char: if is_wchar(c) then print_wchar(c)
  else print(c);
left_brace, right_brace, math_shift, tab_mark, sup_mark, sub_mark, spacer: 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: print("->");
othercases print_esc("BAD.")
endcases
```

This code is used in section 293.

295. 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;
```

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

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

297. 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* }

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

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

define *chr_cmd*(#) ≡
  begin *print*(#); *print_ASCII*(*chr_code*);
  end
define *wchr_cmd*(#) ≡
  begin *print*(#);
  if *is_wchar*(*chr_code*) **then** *print_wchar*(*chr_code*)
  else *print_ASCII*(*chr_code*);
  end

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

This code is used in section 252.

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

procedure *show_cur_cmd_chr*;
  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*); *print_char*("`}`"); *end_diagnostic*(*false*);
  end;

300. 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*$[0\,..\,(ptr-1)]$. 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*$[0\,..\,(ptr-1)]$. 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;

301. ⟨ Global variables 13 ⟩ +≡
input_stack: ↑*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) }

302. We've already defined the special variable *loc* ≡ *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 }

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

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

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

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

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 }

304. 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: ↑*alpha_file*;
line: *integer*; { current line number in the current source file }
line_stack: ↑*integer*;
source_filename_stack: ↑*str_number*;
full_source_filename_stack: ↑*str_number*;

305. 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 }

306. 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* 306 ⟩ ≡
procedure *runaway*;
  var *p*: *pointer*; { head of runaway list }
  begin if *scanner_status* > *skipping* **then**
    begin case *scanner_status* **of**
    defining: **begin** *print_nl*("Runaway␣definition"); *p* ← *def_ref*;
      end;
    matching: **begin** *print_nl*("Runaway␣argument"); *p* ← *temp_head*;
      end;
    aligning: **begin** *print_nl*("Runaway␣preamble"); *p* ← *hold_head*;
      end;
    absorbing: **begin** *print_nl*("Runaway␣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 119.

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

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

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

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

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

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

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

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

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

define *token_list* $= 0$ { *state* code when scanning a token list }
define *token_type* $\equiv$ *index* { type of current token list }
define *param_start* $\equiv$ *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 *write_text* = 15 { *token_type* code for `\write` }

308. 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: ↑*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 $\le$ *param_size* + 9 }

309. 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 }

310. 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* }

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

procedure *show_context*; { prints where the scanner is }
  label *done*;
  var *old_setting*: 0 . . *max_selector*; { saved *selector* setting }
    nn: *integer*; { number of contexts shown so far, less one }
    bottom_line: *boolean*; { have we reached the final context to be shown? }
    ⟨ Local variables for formatting calculations 315 ⟩
  begin *base_ptr* ← *input_ptr*; *input_stack*[*base_ptr*] ← *cur_input*; { store current state }
  nn ← −1; *bottom_line* ← *false*;
  loop begin *cur_input* ← *input_stack*[*base_ptr*]; { enter into the context }
    if (*state* ≠ *token_list*) **then**
      if (*name* > 17) ∨ (*base_ptr* = 0) **then** *bottom_line* ← *true*;
    if (*base_ptr* = *input_ptr*) ∨ *bottom_line* ∨ (*nn* < *error_context_lines*) **then**
      ⟨ Display the current context 312 ⟩
    else if *nn* = *error_context_lines* **then**
        begin *print_nl*("..."); *incr*(*nn*); { omitted if *error_context_lines* < 0 }
        end;
    if *bottom_line* **then goto** *done*;
    decr(*base_ptr*);
    end;
done: *cur_input* ← *input_stack*[*input_ptr*]; { restore original state }
  end;

312. ⟨ Display the current context 312 ⟩ ≡
  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 313 ⟩;
      ⟨ Pseudoprint the line 318 ⟩;
      end
    else begin ⟨ Print type of token list 314 ⟩;
      ⟨ Pseudoprint the token list 319 ⟩;
      end;
    selector ← *old_setting*; { stop pseudoprinting }
    ⟨ Print two lines using the tricky pseudoprinted information 317 ⟩;
    incr(*nn*);
    end;
  end

This code is used in section 311.

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

⟨ Print location of current line 313 ⟩ ≡

```
if name ≤ 17 then
  if terminal_input then
    if base_ptr = 0 then print_nl("<*>")
    else print_nl("<insert>␣")
  else begin print_nl("<read␣");
    if name = 17 then print_char("*") else print_int(name − 1);
    print_char(">");
    end
else begin print_nl("l."); print_int(line);
  end;
print_char("␣")
```

This code is used in section 312.

314. ⟨ Print type of token list 314 ⟩ ≡

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

This code is used in section 312.

315. 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 *trick_buf* $[k \,\mathbf{mod}\, \mathit{error\_line}]$ if $k < \mathit{trick\_count}$, otherwise character k is dropped. Initially we set $\mathit{tally} \leftarrow 0$ and $\mathit{trick\_count} \leftarrow 1000000$; then when we reach the point where transition from line 1 to line 2 should occur, we set $\mathit{first\_count} \leftarrow \mathit{tally}$ and $\mathit{trick\_count} \leftarrow \max(\mathit{error\_line}, \mathit{tally} + 1 + \mathit{error\_line} - \mathit{half\_error\_line})$. 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 $k = \mathit{first\_count}$, and the length of the context information gathered for line 2 is $m = \min(\mathit{tally}, \mathit{trick\_count}) - k$. If $l + k \le h$, where $h = \mathit{half\_error\_line}$, we print $\mathit{trick\_buf}[0 \mathinner{..} k-1]$ 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

$$\mathit{trick\_buf}[(l + k - h + 3) \mathinner{..} k - 1],$$

where subscripts of *trick_buf* are circular modulo *error_line*. The second line consists of n spaces followed by $\mathit{trick\_buf}[k \mathinner{..} (k + m - 1)]$, unless $n + m > \mathit{error\_line}$; in the latter case, further cropping is done. This is easier to program than to explain.

⟨Local variables for formatting calculations 315⟩ ≡
i: $0 \mathinner{..} \mathit{buf\_size}$; { index into *buffer* }
j: $0 \mathinner{..} \mathit{buf\_size}$; { end of current line in *buffer* }
l: $0 \mathinner{..} \mathit{half\_error\_line}$; { length of descriptive information on line 1 }
m: *integer*; { context information gathered for line 2 }
n: $0 \mathinner{..} \mathit{error\_line}$; { length of line 1 }
p: *integer*; { starting or ending place in *trick_buf* }
q: *integer*; { temporary index }

This code is used in section 311.

316. The following code sets up the print routines so that they will gather the desired information.

define *begin_pseudoprint* ≡
  begin $l \leftarrow \mathit{tally}$; $\mathit{tally} \leftarrow 0$; $\mathit{selector} \leftarrow \mathit{pseudo}$; $\mathit{trick\_count} \leftarrow 1000000$;
  end

define *set_trick_count* ≡
  begin $\mathit{first\_count} \leftarrow \mathit{tally}$; $\mathit{trick\_count} \leftarrow \mathit{tally} + 1 + \mathit{error\_line} - \mathit{half\_error\_line}$;
  if $\mathit{trick\_count} < \mathit{error\_line}$ **then** $\mathit{trick\_count} \leftarrow \mathit{error\_line}$;
  end

317. And the following code uses the information after it has been gathered.

⟨ Print two lines using the tricky pseudoprinted information 317 ⟩ ≡
  if *trick_count* = 1000000 **then** *set_trick_count*; { *set_trick_count* must be performed }
  if *tally* < *trick_count* **then** $m \gets$ *tally* − *first_count*
  else $m \gets$ *trick_count* − *first_count*; { context on line 2 }
  if l + *first_count* ≤ *half_error_line* **then**
    begin $p \gets 0$; $n \gets l$ + *first_count*;
    end
  else begin *print*("..."); $p \gets l$ + *first_count* − *half_error_line* + 3; $n \gets$ *half_error_line*;
    end;
  for $q \gets p$ **to** *first_count* − 1 **do** *print_char*(*trick_buf*[q **mod** *error_line*]);
  print_ln;
  for $q \gets 1$ **to** n **do** *print_char*("␣"); { print n spaces to begin line 2 }
  if $m + n \le$ *error_line* **then** $p \gets$ *first_count* + m
  else $p \gets$ *first_count* + (*error_line* − n − 3);
  for $q \gets$ *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 312.

318. 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 318 ⟩ ≡
  begin_pseudoprint;
  if *buffer*[*limit*] = *end_line_char* **then** $j \gets$ *limit*
  else $j \gets$ *limit* + 1; { determine the effective end of the line }
  if $j > 0$ **then**
    for $i \gets$ *start* **to** $j - 1$ **do**
      begin if i = *loc* **then** *set_trick_count*;
      print(*buffer*[i]);
      end

This code is used in section 312.

319. ⟨ Pseudoprint the token list 319 ⟩ ≡
  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 312.

320. 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 320 ⟩ ≡
  set_trick_count

This code is used in section 292.

321. 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
```

322. 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
```

323. 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;
```

324. 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* $\geq$ *backed_up* **then** { token list to be deleted }
    begin if *token_type* $\leq$ *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;

325. 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;

326. ⟨ Insert token *p* into TEX's input 326 ⟩ ≡
  begin *t* ← *cur_tok*; *cur_tok* ← *p*; *back_input*; *cur_tok* ← *t*;
  end

This code is used in section 282.

327. 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;

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

procedure *begin_file_reading*;
  begin if *in_open* $=$ *max_in_open* **then** *overflow*("text␣input␣levels", *max_in_open*);
  if *first* $=$ *buf_size* **then** *overflow*("buffer␣size", *buf_size*);
  incr(*in_open*); *push_input*; *index* $\leftarrow$ *in_open*; *source_filename_stack*[*index*] $\leftarrow 0$;
  full_source_filename_stack[*index*] $\leftarrow 0$; *line_stack*[*index*] $\leftarrow$ *line*; *start* $\leftarrow$ *first*; *state* $\leftarrow$ *mid_line*;
  name $\leftarrow 0$; { *terminal_input* is now *true* }
  end;

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

procedure *end_file_reading*;
  begin *first* $\leftarrow$ *start*; *line* $\leftarrow$ *line_stack*[*index*];
  if *name* > 17 **then** *a_close*(*cur_file*); { forget it }
  pop_input; *decr*(*in_open*);
  end;

330. 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* $\neq$ *token_list*) $\land$ *terminal_input* $\land$ (*input_ptr* > 0) $\land$ (*loc* $>$ *limit*) **do** *end_file_reading*;
  print_ln; *clear_terminal*;
  end;

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

⟨Initialize the input routines 331⟩ ≡
  begin *input_ptr* $\leftarrow 0$; *max_in_stack* $\leftarrow 0$; *source_filename_stack*[0] $\leftarrow 0$;
  full_source_filename_stack[0] $\leftarrow 0$; *in_open* $\leftarrow 0$; *open_parens* $\leftarrow 0$; *max_buf_stack* $\leftarrow 0$; *param_ptr* $\leftarrow 0$;
  max_param_stack $\leftarrow 0$; *first* $\leftarrow$ *buf_size*;
  repeat *buffer*[*first*] $\leftarrow 0$; *decr*(*first*);
  until *first* $= 0$;
  scanner_status $\leftarrow$ *normal*; *warning_index* $\leftarrow$ *null*; *first* $\leftarrow 1$; *state* $\leftarrow$ *new_line*; *start* $\leftarrow 1$; *index* $\leftarrow 0$;
  line $\leftarrow 0$; *name* $\leftarrow 0$; *force_eof* $\leftarrow$ *false*; *align_state* $\leftarrow 1000000$;
  if $\neg$*init_terminal* **then goto** *final_end*;
  limit $\leftarrow$ *last*; *first* $\leftarrow$ *last* $+ 1$; { *init_terminal* has set *loc* and *last* }
  end

This code is used in section 1340.

332. 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.)

333. 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`' }

334. ⟨Put each of TEX's primitives into the hash table 226⟩ +≡
primitive("par", *par_end*, 256); { cf. *scan_file_name* }
par_loc ← *cur_val*; *par_token* ← *cs_token_flag* + *par_loc*;

335. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
par_end: *print_esc*("par");

336. 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 337⟩;
    if scanner_status > skipping then ⟨Tell the user what has run away and try to recover 338⟩
    else begin print_err("Incomplete␣"); print_cmd_chr(if_test, cur_if);
      print(";␣all␣text␣was␣ignored␣after␣line␣"); print_int(skip_line);
      help3("A␣forbidden␣control␣sequence␣occurred␣in␣skipped␣text.")
      ("This␣kind␣of␣error␣happens␣when␣you␣say␣`\if...´␣and␣forget")
      ("the␣matching␣`\fi´.␣I´ve␣inserted␣a␣`\fi´;␣this␣might␣work.");
      if cur_cs ≠ 0 then cur_cs ← 0
      else help_line[2] ← "The␣file␣ended␣while␣I␣was␣skipping␣conditional␣text.";
      cur_tok ← cs_token_flag + frozen_fi; ins_error;
      end;
    deletions_allowed ← true;
    end;
  end;
```

337. 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 337⟩ ≡
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 336.

338. ⟨Tell the user what has run away and try to recover 338⟩ ≡
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 either '`definition`' or '`use`' or '`preamble`' or '`text`', and insert tokens that should lead to recovery 339⟩;
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 336.

339. 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 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 339⟩ ≡
p ← *get_avail*;
case *scanner_status* **of**
defining: **begin** *print*(`"␣while␣scanning␣definition"`); *info*(*p*) ← *right_brace_token* + `"}"`;
end;
matching: **begin** *print*(`"␣while␣scanning␣use"`); *info*(*p*) ← *par_token*; *long_state* ← *outer_call*;
end;
aligning: **begin** *print*(`"␣while␣scanning␣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*(`"␣while␣scanning␣text"`); *info*(*p*) ← *right_brace_token* + `"}"`;
end;
end; { there are no other cases }
ins_list(*p*)

This code is used in section 338.

340. We need to mention a procedure here that may be called by *get_next*.

procedure *firm_up_the_line*; *forward*;

341. 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 TEX.

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 }
first_control_char: *integer*; { the first character code of control sequence }
begin *restart*: *cur_cs* ← 0;
if *state* ≠ *token_list* **then** ⟨ Input from external file, **goto** *restart* if no input found 343 ⟩
else ⟨ Input from token list, **goto** *restart* if end of list or if a parameter needs to be expanded 357 ⟩;
⟨ If an alignment entry has just ended, take appropriate action 342 ⟩;
exit: **end**;

342. 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 342 ⟩ ≡
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* 792 ⟩

This code is used in section 341.

343. The *get_wchar* macro tries to read a double-byte character from *buffer* at the position specified by the parameter. The code value is stored in the global variable *cur_chr*.

define *get_wchar*(#) ≡
  begin $cur\_chr \leftarrow buffer[\#]$; *incr*(#);
  if $cur\_chr > 127 \land pux\_CJKinput = 1$ **then**
   begin $cur\_chr \leftarrow cur\_chr * 256 + buffer[\#]$; *incr*(#)
   end
  end

⟨Input from external file, **goto** *restart* if no input found 343⟩ ≡
begin *switch*: **if** $loc \le limit$ **then** { current line not yet finished }
 begin *get_wchar*(*loc*);
reswitch: $cur\_cmd \leftarrow get\_cat\_code(cur\_chr)$; ⟨Change state if necessary, and **goto** *switch* if the current character should be ignored, or **goto** *reswitch* if the current character changes to another 344⟩;
 end
else begin $state \leftarrow new\_line$;
 ⟨Move to next line of file, or **goto** *restart* if there is no next line, or **return** if a `\read` line has finished 360⟩;
 check_interrupt; **goto** *switch*;
 end;
end

This code is used in section 341.

344. 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 344⟩ ≡
case $state + cur\_cmd$ **of**
⟨Cases where character is ignored 345⟩: **goto** *switch*;
any_state_plus(*escape*): ⟨Scan a control sequence and set $state \leftarrow skip\_blanks$ or *mid_line* 354⟩;
any_state_plus(*active_char*): ⟨Process an active-character control sequence and set $state \leftarrow mid\_line$ 353⟩;
any_state_plus(*sup_mark*): ⟨If this *sup_mark* starts an expanded character like `^^A` or `^^df`, then **goto** *reswitch*, otherwise set $state \leftarrow mid\_line$ 352⟩;
any_state_plus(*invalid_char*): ⟨Decry the invalid character and **goto** *restart* 346⟩;
⟨Handle situations involving spaces, braces, changes of state 347⟩
othercases *do_nothing*
endcases

This code is used in section 343.

345. ⟨Cases where character is ignored 345⟩ ≡
any_state_plus(*ignore*), $skip\_blanks + spacer$, $new\_line + spacer$

This code is used in section 344.

346. 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* 346⟩ ≡
begin *print_err*("Text␣line␣contains␣an␣invalid␣character");
help2("A␣funny␣symbol␣that␣I␣can´t␣read␣has␣just␣been␣input.")
("Continue,␣and␣I´ll␣forget␣that␣it␣ever␣happened.");
deletions_allowed ← *false*; *error*; *deletions_allowed* ← *true*; **goto** *restart*;
end

This code is used in section 344.

347. **define** *add_delims_to*(#) ≡ # + *math_shift*, # + *tab_mark*, # + *mac_param*, # + *sub_mark*, # + *letter*, # + *other_char*

⟨Handle situations involving spaces, braces, changes of state 347⟩ ≡
mid_line + *spacer*: ⟨Enter *skip_blanks* state, emit a space 349⟩;
mid_line + *car_ret*: ⟨Finish line, emit a space 348⟩;
skip_blanks + *car_ret*, *any_state_plus*(*comment*): ⟨Finish line, **goto** *switch* 350⟩;
new_line + *car_ret*: ⟨Finish line, emit a \par 351⟩;
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 344.

348. When a character of type *spacer* gets through, its character code is changed to "␣" = ´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 348⟩ ≡
begin *loc* ← *limit* + 1; *cur_cmd* ← *spacer*; *cur_chr* ← "␣";
end

This code is used in section 347.

349. The following code is performed only when *cur_cmd* = *spacer*.

⟨Enter *skip_blanks* state, emit a space 349⟩ ≡
begin *state* ← *skip_blanks*; *cur_chr* ← "␣";
end

This code is used in section 347.

350. ⟨Finish line, **goto** *switch* 350⟩ ≡
begin *loc* ← *limit* + 1; **goto** *switch*;
end

This code is used in section 347.

351. ⟨Finish line, emit a \par 351⟩ ≡
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 347.

352. Notice that a code like `^^8` becomes `x` if not followed by a hex digit.

define $is\_hex(\#) \equiv (((\# \geq \texttt{"0"}) \land (\# \leq \texttt{"9"})) \lor ((\# \geq \texttt{"a"}) \land (\# \leq \texttt{"f"})))$
define $hex\_to\_cur\_chr \equiv$
  if $c \leq \texttt{"9"}$ **then** $cur\_chr \leftarrow c - \texttt{"0"}$ **else** $cur\_chr \leftarrow c - \texttt{"a"} + 10$;
  if $cc \leq \texttt{"9"}$ **then** $cur\_chr \leftarrow 16 * cur\_chr + cc - \texttt{"0"}$
  else $cur\_chr \leftarrow 16 * cur\_chr + cc - \texttt{"a"} + 10$

⟨If this *sup_mark* starts an expanded character like `^^A` or `^^df`, then **goto** *reswitch*, otherwise set $state \leftarrow mid\_line$ 352⟩ ≡
begin if $cur\_chr = buffer[loc]$ **then**
  if $loc < limit$ **then**
    begin $c \leftarrow buffer[loc + 1]$; **if** $c <$ ´200 **then** { yes we have an expanded char }
      begin $loc \leftarrow loc + 2$;
      if $is\_hex(c)$ **then**
        if $loc \leq limit$ **then**
          begin $cc \leftarrow buffer[loc]$; **if** $is\_hex(cc)$ **then**
            begin $incr(loc)$; *hex_to_cur_chr*; **goto** *reswitch*;
            end;
          end;
      if $c <$ ´100 **then** $cur\_chr \leftarrow c +$ ´100 **else** $cur\_chr \leftarrow c -$ ´100;
      goto *reswitch*;
      end;
    end;
$state \leftarrow mid\_line$;
end

This code is used in section 344.

353. ⟨Process an active-character control sequence and set $state \leftarrow mid\_line$ 353⟩ ≡
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 344.

354. 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.

TCW: When the flag *expand_char* is true, we stop using *get_wchar* but merely get a one-byte character so that reading DBCS characters will not be confused. Besides, we neet to handle alphabetic numbers of the form `'\c`, where `c` is a DBCS characters.

⟨Scan a control sequence and set *state* ← *skip_blanks* or *mid_line* 354⟩ ≡
  begin if *loc* > *limit* **then** *cur_cs* ← *null_cs* { *state* is irrelevant in this case }
  else begin *first_control_char* ← −1;
  start_cs: *k* ← *loc*;
    if *expand_char* **then**
      begin *cur_chr* ← *buffer*[*k*]; *incr*(*k*); *expand_char* ← *false*;
      end
    else *get_wchar*(*k*);
    cat ← *get_cat_code*(*cur_chr*);
    if *first_control_char* = −1 **then** *first_control_char* ← *cur_chr*;
    if *cat* = *letter* **then** *state* ← *skip_blanks*
    else if *cat* = *spacer* **then** *state* ← *skip_blanks*
      else *state* ← *mid_line*;
    if (*cat* = *letter*) ∧ (*k* ≤ *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* 356⟩
    else ⟨If an expanded code is present, reduce it and **goto** *start_cs* 355⟩;
      { the control sequence is a control symbol, i.e., its name consisits of only one letter. }
    if *is_wchar*(*first_control_char*) **then**
      begin *cur_cs* ← *single_base* + *first_control_char*; *loc* ← *loc* + 2;
      end
    else begin *cur_cs* ← *single_base* + *buffer*[*loc*]; *incr*(*loc*);
      end;
    end;
found: *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 344.

355. Whenever we reach the following piece of code, we will have $cur\_chr = buffer[k-1]$ and $k \le limit + 1$ and $cat = cat\_code(cur\_chr)$. If an expanded code like `^^A` or `^^df` appears in $buffer[(k-1) \mathinner{\ldotp\ldotp} (k+1)]$ or $buffer[(k-1) \mathinner{\ldotp\ldotp} (k+2)]$, we will store the corresponding code in $buffer[k-1]$ and shift the rest of the buffer left two or three places.

TCW: If it is indeed an expanded code, set the flag *expand_char*.

⟨If an expanded code is present, reduce it and **goto** *start_cs* 355⟩ ≡
begin if $buffer[k] = cur\_chr$ **then if** $cat = sup\_mark$ **then if** $k < limit$ **then**
  begin $c \leftarrow buffer[k+1]$; **if** $c <$ ´200 **then** { yes, one is indeed present }
   begin $d \leftarrow 2$; $expand\_char \leftarrow true$;
   if $is\_hex(c)$ **then if** $k + 2 \le limit$ **then**
     begin $cc \leftarrow buffer[k+2]$; **if** $is\_hex(cc)$ **then** $incr(d)$;
     end;
   if $d > 2$ **then**
    begin $hex\_to\_cur\_chr$; $buffer[k-1] \leftarrow cur\_chr$;
    end
   else if $c <$ ´100 **then** $buffer[k-1] \leftarrow c +$ ´100
    else $buffer[k-1] \leftarrow c -$ ´100;
   $limit \leftarrow limit - d$; $first \leftarrow first - d$;
   while $k \le limit$ **do**
    begin $buffer[k] \leftarrow buffer[k+d]$; $incr(k)$;
    end;
   goto *start_cs*;
   end;
  end;
end

This code is used in sections 354 and 356.

356. ⟨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* 356⟩ ≡
begin repeat $get\_wchar(k)$; $cat \leftarrow get\_cat\_code(cur\_chr)$;
until $(cat \ne letter) \lor (k > limit)$;
⟨If an expanded code is present, reduce it and **goto** *start_cs* 355⟩;
if $cat \ne letter$ **then**
 if $cur\_chr > 256$ **then** $k \leftarrow k - 2$ { go back 2 steps for a non-letter DBCS code }
 else $decr(k)$; { now k points to first nonletter }
if $k > loc + 1 \land \neg(k = loc + 2 \land first\_control\_char > 255)$ **then**
  { multiletter control sequence has been scanned }
 begin $cur\_cs \leftarrow id\_lookup(loc, k - loc)$; $loc \leftarrow k$; **goto** *found*;
 end;
end

This code is used in section 354.

357. 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 357⟩ ≡
if $loc \neq null$ **then** { list not exhausted }
begin $t \leftarrow info(loc)$; $loc \leftarrow link(loc)$; { move to next }
if $t \geq cs\_token\_flag$ **then** { a control sequence token }
begin $cur\_cs \leftarrow t - cs\_token\_flag$; $cur\_cmd \leftarrow eq\_type(cur\_cs)$; $cur\_chr \leftarrow equiv(cur\_cs)$;
if $cur\_cmd \geq outer\_call$ **then**
if $cur\_cmd = dont\_expand$ **then** ⟨Get the next token, suppressing expansion 358⟩
else *check_outer_validity*;
end
else begin $cur\_cmd \leftarrow t$ **div** ″10000; $cur\_chr \leftarrow t$ **mod** ″10000;
case *cur_cmd* **of**
left_brace: *incr*(*align_state*);
right_brace: *decr*(*align_state*);
out_param: ⟨Insert macro parameter and **goto** *restart* 359⟩;
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 341.

358. 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 358⟩ ≡
begin $cur\_cs \leftarrow info(loc) - cs\_token\_flag$; $loc \leftarrow null$;
$cur\_cmd \leftarrow eq\_type(cur\_cs)$; $cur\_chr \leftarrow equiv(cur\_cs)$;
if $cur\_cmd > max\_command$ **then**
begin $cur\_cmd \leftarrow relax$; $cur\_chr \leftarrow no\_expand\_flag$;
end;
end

This code is used in section 357.

359. ⟨Insert macro parameter and **goto** *restart* 359⟩ ≡
begin $begin\_token\_list(param\_stack[param\_start + cur\_chr - 1], parameter)$; **goto** *restart*;
end

This code is used in section 357.

360. All of the easy branches of *get_next* have now been taken care of. There is one more branch.

define *end_line_char_inactive* $\equiv$ (*end_line_char* < 0) $\vee$ (*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 360⟩ $\equiv$

```
if name > 17 then ⟨Read next line of file into buffer, or goto restart if the file has ended 362⟩
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 343.

361. The global variable *force_eof* is normally *false*; it is set *true* by an `\endinput` command.

⟨Global variables 13⟩ $+\equiv$

force_eof: *boolean*; { should the next `\input` be aborted early? }

362. ⟨Read next line of file into *buffer*, or **goto** *restart* if the file has ended 362⟩ $\equiv$

```
begin incr(line); first ← start;
if ¬force_eof then
  begin if input_ln(cur_file, true) then   { not end of file }
    firm_up_the_line   { this sets limit }
  else force_eof ← true;
  end;
if force_eof then
  begin print_char(")"); decr(open_parens); update_terminal;   { show user that file has been read }
  force_eof ← false; end_file_reading;   { resume previous level }
  check_outer_validity; goto restart;
  end;
if end_line_char_inactive then decr(limit)
else buffer[limit] ← end_line_char;
first ← limit + 1; loc ← start;   { ready to read }
end
```

This code is used in section 360.

363. 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;

364. 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*.

365. 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* ∗ ″`10000`) + *cur_chr*
  else *cur_tok* ← *cs_token_flag* + *cur_cs*;
  end;

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

367. Sometimes, recursive calls to the following *expand* routine may cause exhaustion of the run-time calling stack, resulting in forced execution stops by the operating system. To diminish the chance of this happening, a counter is used to keep track of the recursion depth, in conjunction with a constant called *expand_depth*.

This does not catch all possible infinite recursion loops, just the ones that exhaust the application calling stack. The actual maximum value of *expand_depth* is outside of our control, but the initial setting of 10000 should be enough to prevent problems.

⟨ Global variables 13 ⟩ +≡
expand_depth_count: *integer*;

368. ⟨ Set initial values of key variables 21 ⟩ +≡
  expand_depth_count ← 0;

369. 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 392 ⟩
⟨ Declare the procedure called insert_relax 382 ⟩
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;
  var t: halfword; { token that is being "expanded after" }
    p, q, r: pointer; { for list manipulation }
    j: 0 .. buf_size; { index into buffer }
    cv_backup: integer; { to save the global quantity cur_val }
    cvl_backup, radix_backup, co_backup: small_number; { to save cur_val_level, etc. }
    backup_backup: pointer; { to save link(backup_head) }
    save_scanner_status: small_number; { temporary storage of scanner_status }
  begin incr(expand_depth_count);
  if expand_depth_count ≥ expand_depth then overflow("expansion␣depth", expand_depth);
  cv_backup ← cur_val; cvl_backup ← cur_val_level; radix_backup ← radix; co_backup ← cur_order;
  backup_backup ← link(backup_head);
  if cur_cmd < call then ⟨ Expand a nonmacro 370 ⟩
  else if cur_cmd < end_template then macro_call
    else ⟨ Insert a token containing frozen_endv 378 ⟩;
  cur_val ← cv_backup; cur_val_level ← cvl_backup; radix ← radix_backup; cur_order ← co_backup;
  link(backup_head) ← backup_backup; decr(expand_depth_count);
  end;
```

370. ⟨ Expand a nonmacro 370 ⟩ ≡
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 389 ⟩;
expand_after: ⟨ Expand the token after the next token 371 ⟩;
no_expand: ⟨ Suppress expansion of the next token 372 ⟩;
cs_name: ⟨ Manufacture a control sequence name 375 ⟩;
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` 513 ⟩;
input: ⟨ Initiate or terminate input from a file 381 ⟩;
othercases ⟨ Complain about an undefined macro 373 ⟩
endcases;
end

This code is used in section 370.

371. It takes only a little shuffling to do what TEX calls `\expandafter`.

⟨ Expand the token after the next token 371 ⟩ ≡
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 370.

372. The implementation of `\noexpand` is a bit trickier, because it is necessary to insert a special '*dont_expand*' marker into TEX'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.

⟨ Suppress expansion of the next token 372 ⟩ ≡
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 370.

373. ⟨ Complain about an undefined macro 373 ⟩ ≡
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.,␣`\hobx´),␣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 370.

374. 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 $q \gets$ *get_avail*; *link*(p) $\gets q$; *info*(q) $\gets$ #; $p \gets q$; { *link*(p) is *null* }
  end
define *fast_store_new_token*(#) ≡
  begin *fast_get_avail*(q); *link*(p) $\gets q$; *info*(q) $\gets$ #; $p \gets q$; { *link*(p) is *null* }
  end

375. ⟨ Manufacture a control sequence name 375 ⟩ ≡
begin $r \gets$ *get_avail*; $p \gets r$; { head of the list of characters }
repeat *get_x_token*;
  if *cur_cs* $= 0$ **then** *store_new_token*(*cur_tok*);
until *cur_cs* $\neq 0$;
if *cur_cmd* $\neq$ *end_cs_name* **then** ⟨ Complain about missing `\endcsname` 376 ⟩;
⟨ Look up the characters of list r in the hash table, and set *cur_cs* 377 ⟩;
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 $\gets$ *cur_cs* $+$ *cs_token_flag*; *back_input*;
end

This code is used in section 370.

376. ⟨ Complain about missing `\endcsname` 376 ⟩ ≡
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 section 375.

377. ⟨Look up the characters of list r in the hash table, and set *cur_cs* 377⟩ ≡

$j \leftarrow first$; $p \leftarrow link(r)$;
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;
  $db\_char \leftarrow info(p)$ **mod** ″10000;
  if *is_wchar*(*db_char*) **then** { a double-byte char }
    begin $buffer[j] \leftarrow db\_char$ **div** 256; $buffer[j+1] \leftarrow db\_char$ **mod** 256; $j \leftarrow j + 2$;
    end
  else begin $buffer[j] \leftarrow db\_char$; *incr*($j$);
    end;
  $p \leftarrow link(p)$; { fix this for 2-byte code }
  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 375.

378. 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* 378⟩ ≡
  begin $cur\_tok \leftarrow cs\_token\_flag + frozen\_endv$; *back_input*;
  end

This code is used in section 369.

379. 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 226⟩ +≡
  primitive("input", *input*, 0);
  primitive("endinput", *input*, 1);

380. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
input: **if** $chr\_code = 0$ **then** *print_esc*("input") **else** *print_esc*("endinput");

381. ⟨Initiate or terminate input from a file 381⟩ ≡
  if $cur\_chr > 0$ **then** $force\_eof \leftarrow true$
  else if *name_in_progress* **then** *insert_relax*
    else *start_input*

This code is used in section 370.

382. 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* 382 ⟩ ≡
procedure *insert_relax*;
  begin $cur\_tok \gets cs\_token\_flag + cur\_cs$; *back_input*; $cur\_tok \gets cs\_token\_flag + frozen\_relax$; *back_input*;
  $token\_type \gets inserted$;
  end;

This code is used in section 369.

383. 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 $cur\_cmd \le max\_command$ **then goto** *done*;
  if $cur\_cmd \ge call$ **then**
    if $cur\_cmd < end\_template$ **then** *macro_call*
    else begin $cur\_cs \gets frozen\_endv$; $cur\_cmd \gets endv$; **goto** *done*; { $cur\_chr = null\_list$ }
      end
  else *expand*;
  goto *restart*;
done: **if** $cur\_cs = 0$ **then** $cur\_tok \gets (cur\_cmd * "10000) + cur\_chr$
  else $cur\_tok \gets cs\_token\_flag + cur\_cs$;
  end;

384. The *get_x_token* procedure is 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** $cur\_tok \gets (cur\_cmd * "10000) + cur\_chr$
  else $cur\_tok \gets cs\_token\_flag + cur\_cs$;
  end;

385. 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 $top\_mark\_code = 0$ { 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 $top\_mark \equiv cur\_mark[top\_mark\_code]$
  define $first\_mark \equiv cur\_mark[first\_mark\_code]$
  define $bot\_mark \equiv cur\_mark[bot\_mark\_code]$
  define $split\_first\_mark \equiv cur\_mark[split\_first\_mark\_code]$
  define $split\_bot\_mark \equiv cur\_mark[split\_bot\_mark\_code]$

⟨ Global variables 13 ⟩ +≡
cur_mark: **array** [*top_mark_code* .. *split_bot_mark_code*] **of** *pointer*; { token lists for marks }

386. ⟨ Set initial values of key variables 21 ⟩ +≡
top_mark ← *null*; *first_mark* ← *null*; *bot_mark* ← *null*; *split_first_mark* ← *null*; *split_bot_mark* ← *null*;

387. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
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*);

388. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
top_bot_mark: **case** *chr_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;

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

⟨ Insert the appropriate mark text into the scanner 389 ⟩ ≡
begin if *cur_mark*[*cur_chr*] ≠ *null* **then** *begin_token_list*(*cur_mark*[*cur_chr*], *mark_text*);
end

This code is used in section 370.

390. 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.

⟨ Global variables 13 ⟩ +≡
long_state: *call* .. *long_outer_call*; { governs the acceptance of \par }

391. 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.)

⟨ Global variables 13 ⟩ +≡
pstack: **array** [0 .. 8] **of** *pointer*; { arguments supplied to a macro }

392. 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* 392 ⟩ ≡
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 404 ⟩;
  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 394 ⟩;
  ⟨ Feed the macro body and its parameters to the scanner 393 ⟩;
exit: *scanner_status* ← *save_scanner_status*; *warning_index* ← *save_warning_index*;
  end;

This code is used in section 369.

393. 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 393 ⟩ ≡
  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 392.

394. 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 394⟩ ≡
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 395⟩;
    { 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 392.

395. 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 395⟩ ≡
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* 397⟩;
⟨Contribute the recently matched tokens to the current parameter, and **goto** *continue* if a partial match is still in effect; but abort if $s = null$ 400⟩;
if *cur_tok* = *par_token* **then**
  if *long_state* ≠ *long_call* **then** ⟨Report a runaway argument and abort 399⟩;
if *cur_tok* < *right_brace_limit* **then**
  if *cur_tok* < *left_brace_limit* **then** ⟨Contribute an entire group to the current parameter 402⟩
  else ⟨Report an extra right brace and **goto** *continue* 398⟩
else ⟨Store the current token, but **goto** *continue* if it is a blank space that would become an undelimited parameter 396⟩;
incr(m);
if *info*(r) > *end_match_token* **then goto** *continue*;
if *info*(r) < *match_token* **then goto** *continue*;
found: **if** $s \neq null$ **then** ⟨Tidy up the parameter just scanned, and tuck it away 403⟩

This code is used in section 394.

396. ⟨ Store the current token, but **goto** *continue* if it is a blank space that would become an undelimited parameter 396 ⟩ ≡

```
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 395.

397. 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* 397 ⟩ ≡

```
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 395.

398. ⟨ Report an extra right brace and **goto** *continue* 398 ⟩ ≡

```
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 395.

399. If $long\_state = outer\_call$, a runaway argument has already been reported.

⟨ Report a runaway argument and abort 399 ⟩ ≡

```
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 395 and 402.

400. When the following code becomes active, we have matched tokens from s to the predecessor of r, and we have found that $cur\_tok \neq info(r)$. 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 `\par` tokens to sneak in to certain parameters of non-`\long` macros. For example, consider a case like '`\def\a#1\par!{...}`' where the first `\par` is not followed by an exclamation point. In such situations it does not seem appropriate to prohibit the `\par`, so TEX 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 $s = null$ 400⟩ ≡

```
if s ≠ r then
  if s = null then ⟨Report an improper use of the macro and abort 401⟩
  else begin t ← s;
    repeat store_new_token(info(t)); incr(m); u ← link(t); v ← s;
      loop begin if u = r then
          if cur_tok ≠ info(v) then goto done
          else begin r ← link(v); goto continue;
            end;
        if info(u) ≠ info(v) then goto done;
        u ← link(u); v ← link(v);
        end;
    done: t ← link(t);
    until t = r;
    r ← s; { at this point, no tokens are recently matched }
    end
```

This code is used in section 395.

401. ⟨Report an improper use of the macro and abort 401⟩ ≡

```
begin print_err("Use␣of␣"); sprint_cs(warning_index); print("␣doesn´t␣match␣its␣definition");
help4("If␣you␣say,␣e.g.,␣`\def\a1{...}´,␣then␣you␣must␣always")
("put␣`1´␣after␣`\a´,␣since␣control␣sequence␣names␣are")
("made␣up␣of␣letters␣only.␣The␣macro␣here␣has␣not␣been")
("followed␣by␣the␣required␣stuff,␣so␣I´m␣ignoring␣it."); error; return;
end
```

This code is used in section 400.

402. ⟨Contribute an entire group to the current parameter 402⟩ ≡

```
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 399⟩;
  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;
done1: rbrace_ptr ← p; store_new_token(cur_tok);
end
```

This code is used in section 395.

403. 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 403 ⟩ ≡
  begin if $(m = 1) \land (\mathit{info}(p) < \mathit{right\_brace\_limit}) \land (p \neq \mathit{temp\_head})$ **then**
    begin $\mathit{link}(\mathit{rbrace\_ptr}) \leftarrow \mathit{null}$; $\mathit{free\_avail}(p)$; $p \leftarrow \mathit{link}(\mathit{temp\_head})$; $\mathit{pstack}[n] \leftarrow \mathit{link}(p)$; $\mathit{free\_avail}(p)$;
    end
  else $\mathit{pstack}[n] \leftarrow \mathit{link}(\mathit{temp\_head})$;
  $\mathit{incr}(n)$;
  if $\mathit{tracing\_macros} > 0$ **then**
    begin *begin_diagnostic*; *print_nl*(*match_chr*); *print_int*(*n*); *print*("<-");
    $\mathit{show\_token\_list}(\mathit{pstack}[n-1], \mathit{null}, 1000)$; *end_diagnostic*(*false*);
    end;
  end

This code is used in section 395.

404. ⟨ Show the text of the macro being expanded 404 ⟩ ≡
  begin *begin_diagnostic*; *print_ln*; *print_cs*(*warning_index*); *token_show*(*ref_count*);
  end_diagnostic(*false*);
  end

This code is used in section 392.

405. 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.

406. 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 407⟩;
  if $cur\_cmd \neq left\_brace$ **then**
    begin *print_err*("Missing␣{␣inserted");
    help4("A␣left␣brace␣was␣mandatory␣here,␣so␣I´ve␣put␣one␣in.")
    ("You␣might␣want␣to␣delete␣and/or␣insert␣some␣corrections")
    ("so␣that␣I␣will␣find␣a␣matching␣right␣brace␣soon.")
    ("(If␣you´re␣confused␣by␣all␣this,␣try␣typing␣`I}´␣now.)"); *back_error*;
    $cur\_tok \leftarrow left\_brace\_token +$ "{"; $cur\_cmd \leftarrow left\_brace$; $cur\_chr \leftarrow$ "{"; *incr*(*align_state*);
    end;
  end;

407. ⟨Get the next non-blank non-relax non-call token 407⟩ ≡
  repeat *get_x_token*;
  until $(cur\_cmd \neq spacer) \land (cur\_cmd \neq relax)$

This code is used in sections 406, 1081, 1087, 1154, 1163, 1214, 1229, and 1273.

408. 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 409⟩;
  if $cur\_tok \neq other\_token +$ "=" **then** *back_input*;
  end;

409. ⟨Get the next non-blank non-call token 409⟩ ≡
  repeat *get_x_token*;
  until $cur\_cmd \neq spacer$

This code is used in sections 408, 444, 458, 506, 529, 580, 788, 794, 1048, 1418, 1477, 1487, 1493, 1528, 1535, 1535, 1550, and 1558.

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

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

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

411. 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;
```

412. 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 436 ⟩
⟨ Declare procedures that scan font-related stuff 580 ⟩
```

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

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

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

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

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

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

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

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

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

define *scanned_result_end*(#) ≡ *cur_val_level* ← #; **end**
define *scanned_result*(#) ≡ **begin** *cur_val* ← #; *scanned_result_end*

procedure *scan_something_internal*(*level* : *small_number*; *negative* : *boolean*);
{ fetch an internal parameter }
var *m*: *halfword*; { *chr_code* part of the operand token }
p: 0 . . *nest_size*; { index into *nest* }
begin *m* ← *cur_chr*;
case *cur_cmd* **of**
def_code: ⟨ Fetch a character code from some table 417 ⟩;
toks_register, *assign_toks*, *def_family*, *set_font*, *def_font*, *set_cfont*: ⟨ Fetch a token list or font identifier, provided that *level* = *tok_val* 418 ⟩;
assign_int, *puxg_assign_flag*, *puxg_assign_int*: *scanned_result*(*eqtb*[*m*].*int*)(*int_val*);
pux_get_int: ⟨ scan PUTEX internal values 1428 ⟩;
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* 421 ⟩;
set_prev_graf: ⟨ Fetch the *prev_graf* 425 ⟩;
set_page_int: ⟨ Fetch the *dead_cycles* or the *insert_penalties* 422 ⟩;
set_page_dimen: ⟨ Fetch something on the *page_so_far* 424 ⟩;
set_shape: ⟨ Fetch the *par_shape* size 426 ⟩;
set_box_dimen: ⟨ Fetch a box dimension 423 ⟩;
char_given, *math_given*, *pux_char_given*: *scanned_result*(*cur_chr*)(*int_val*);
assign_font_dimen: ⟨ Fetch a font dimension 428 ⟩;
assign_font_int: ⟨ Fetch a font integer 429 ⟩;
register: ⟨ Fetch a register 430 ⟩;
last_item: ⟨ Fetch an item in the current node, if appropriate 427 ⟩;
othercases ⟨ Complain that `\the` can't do this; give zero result 431 ⟩
endcases;
while *cur_val_level* $>$ *level* **do** ⟨ Convert *cur_val* to a lower level 432 ⟩;
⟨ Fix the reference count, if any, and negate *cur_val* if *negative* 433 ⟩;
end;

417. ⟨ Fetch a character code from some table 417 ⟩ ≡
begin if (*m* = *pux_cat_code_base*) ∨ (*m* = *pux_type_code_base*) **then** *scan_wchar_num*
else if *m* = *pux_local_names_base* **then**
begin *char_val_flag* ← *true*; *scan_eight_bit_int*;
end
else *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 416.

418. ⟨Fetch a token list or font identifier, provided that *level* = *tok_val* 418⟩ ≡

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

This code is used in section 416.

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

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

⟨Put each of TEX's primitives into the hash table 226⟩ +≡
primitive("spacefactor", *set_aux*, *hmode*); *primitive*("prevdepth", *set_aux*, *vmode*);
primitive("deadcycles", *set_page_int*, 0); *primitive*("insertpenalties", *set_page_int*, 1);
primitive("wd", *set_box_dimen*, *width_offset*); *primitive*("ht", *set_box_dimen*, *height_offset*);
primitive("dp", *set_box_dimen*, *depth_offset*); *primitive*("lastpenalty", *last_item*, *int_val*);
primitive("lastkern", *last_item*, *dimen_val*); *primitive*("lastskip", *last_item*, *glue_val*);
primitive("inputlineno", *last_item*, *input_line_no_code*); *primitive*("badness", *last_item*, *badness_code*);

420. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
set_aux: **if** *chr_code* = *vmode* **then** *print_esc*("prevdepth") **else** *print_esc*("spacefactor");
set_page_int: **if** *chr_code* = 0 **then** *print_esc*("deadcycles") **else** *print_esc*("insertpenalties");
set_box_dimen: **if** *chr_code* = *width_offset* **then** *print_esc*("wd")
  else if *chr_code* = *height_offset* **then** *print_esc*("ht")
    else *print_esc*("dp");
last_item: **case** *chr_code* **of**
  int_val: *print_esc*("lastpenalty");
  dimen_val: *print_esc*("lastkern");
  glue_val: *print_esc*("lastskip");
  input_line_no_code: *print_esc*("inputlineno");
  othercases *print_esc*("badness")
  endcases;

421. ⟨Fetch the *space_factor* or the *prev_depth* 421⟩ ≡

```
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 416.

422. ⟨Fetch the *dead_cycles* or the *insert_penalties* 422⟩ ≡

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

This code is used in section 416.

423. ⟨Fetch a box dimension 423⟩ ≡

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

This code is used in section 416.

424. 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* ≡ ´7777777777 { $2^{30} - 1$ }

⟨Fetch something on the *page_so_far* 424⟩ ≡

```
begin if (page_contents = empty) ∧ (¬output_active) then
  if m = 0 then cur_val ← max_dimen else cur_val ← 0
else cur_val ← page_so_far[m];
cur_val_level ← dimen_val;
end
```

This code is used in section 416.

425. ⟨Fetch the *prev_graf* 425⟩ ≡

```
if mode = 0 then scanned_result(0)(int_val) { prev_graf = 0 within \write }
else begin nest[nest_ptr] ← cur_list; p ← nest_ptr;
  while abs(nest[p].mode_field) ≠ vmode do decr(p);
  scanned_result(nest[p].pg_field)(int_val);
  end
```

This code is used in section 416.

426. ⟨ Fetch the *par_shape* size 426 ⟩ ≡

```
begin if par_shape_ptr = null then cur_val ← 0
else cur_val ← info(par_shape_ptr);
cur_val_level ← int_val;
end
```

This code is used in section 416.

427. Here is where `\lastpenalty`, `\lastkern`, and `\lastskip` 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.

⟨ Fetch an item in the current node, if appropriate 427 ⟩ ≡

```
if cur_chr > glue_val then
  begin if cur_chr = input_line_no_code then cur_val ← line
  else cur_val ← last_badness;  { cur_chr = badness_code }
  cur_val_level ← int_val;
  end
else begin if cur_chr = glue_val then cur_val ← zero_glue else cur_val ← 0;
  cur_val_level ← cur_chr;
  if ¬is_char_node(tail) ∧ (mode ≠ 0) then
    case cur_chr of
    int_val: if type(tail) = penalty_node then cur_val ← penalty(tail);
    dimen_val: if type(tail) = kern_node then cur_val ← width(tail);
    glue_val: if type(tail) = glue_node then
        begin cur_val ← glue_ptr(tail);
        if subtype(tail) = mu_glue then cur_val_level ← mu_val;
        end;
    end  { there are no other cases }
  else if (mode = vmode) ∧ (tail = head) then
      case cur_chr of
      int_val: cur_val ← last_penalty;
      dimen_val: cur_val ← last_kern;
      glue_val: if last_glue ≠ max_halfword then cur_val ← last_glue;
      end;  { there are no other cases }
  end
```

This code is used in section 416.

428. ⟨ Fetch a font dimension 428 ⟩ ≡

```
begin find_font_dimen(false); font_info[fmem_ptr].sc ← 0;
scanned_result(font_info[cur_val].sc)(dimen_val);
end
```

This code is used in section 416.

429. ⟨ Fetch a font integer 429 ⟩ ≡

```
begin scan_font_ident;
if cur_val ≤ font_max then
  if m = 0 then scanned_result(hyphen_char[cur_val])(int_val)
  else scanned_result(skew_char[cur_val])(int_val);
end
```

This code is used in section 416.

430. ⟨Fetch a register 430⟩ ≡
begin *scan_eight_bit_int*;
case m **of**
int_val: *cur_val* ← *count*(*cur_val*);
dimen_val: *cur_val* ← *dimen*(*cur_val*);
glue_val: *cur_val* ← *skip*(*cur_val*);
mu_val: *cur_val* ← *mu_skip*(*cur_val*);
end; { there are no other cases }
cur_val_level ← m;
end

This code is used in section 416.

431. ⟨Complain that `\the` can't do this; give zero result 431⟩ ≡
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* ≠ *tok_val* **then** *scanned_result*(0)(*dimen_val*)
else *scanned_result*(0)(*int_val*);
end

This code is used in section 416.

432. 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 432⟩ ≡
begin if *cur_val_level* = *glue_val* **then** *cur_val* ← *width*(*cur_val*)
else if *cur_val_level* = *mu_val* **then** *mu_error*;
decr(*cur_val_level*);
end

This code is used in section 416.

433. 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 ≤ *mu_val*.

⟨Fix the reference count, if any, and negate *cur_val* if *negative* 433⟩ ≡
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* 434⟩;
    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 416.

434. ⟨Negate all three glue components of *cur_val* 434⟩ ≡
begin *negate*(*width*(*cur_val*)); *negate*(*stretch*(*cur_val*)); *negate*(*shrink*(*cur_val*));
end

This code is used in section 433.

435. 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*.

436. ⟨ Declare procedures that scan restricted classes of integers 436 ⟩ ≡
procedure *scan_eight_bit_int*;
  begin *scan_int*;
  if $(cur\_val < 0) \lor (cur\_val > 255)$ **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*); $cur\_val \leftarrow 0$;
    end;
  end;

See also sections 437, 438, 439, 440, 1388, and 1419.

This code is used in section 412.

437. ⟨ Declare procedures that scan restricted classes of integers 436 ⟩ +≡
procedure *scan_char_num*;
  begin *scan_int*;
  if $(cur\_val < 0) \lor (cur\_val > 255)$ **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*); $cur\_val \leftarrow 0$;
    end;
  end;

438. While we're at it, we might as well deal with similar routines that will be needed later.

⟨ Declare procedures that scan restricted classes of integers 436 ⟩ +≡
procedure *scan_four_bit_int*;
  begin *scan_int*;
  if $(cur\_val < 0) \lor (cur\_val > 15)$ **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*); $cur\_val \leftarrow 0$;
    end;
  end;

439. ⟨ Declare procedures that scan restricted classes of integers 436 ⟩ +≡
procedure *scan_fifteen_bit_int*;
  begin *scan_int*;
  if $(cur\_val < 0) \lor (cur\_val > \text{´}77777)$ **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*); $cur\_val \leftarrow 0$;
    end;
  end;

440. ⟨ Declare procedures that scan restricted classes of integers 436 ⟩ +≡
procedure *scan_twenty_seven_bit_int*;
  begin *scan_int*;
  if $(cur\_val < 0) \lor (cur\_val > \text{´}777777777)$ **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*); $cur\_val \leftarrow 0$;
    end;
  end;

441. 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 }

442. 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.

⟨ Set initial values of key variables 21 ⟩ +≡
cur_val ← 0; *cur_val_level* ← *int_val*; *radix* ← 0; *cur_order* ← *normal*;

443. 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*;
  var *negative*: *boolean*; { should the answer be negated? }
    m: *integer*; { 2^{31} **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 444 ⟩;
  if *cur_tok* = *alpha_token* **then** ⟨ Scan an alphabetic character code into *cur_val* 445 ⟩
  else if (*cur_cmd* ≥ *min_internal*) ∧ (*cur_cmd* ≤ *max_internal*) **then**
      scan_something_internal(*int_val*, *false*)
    else ⟨ Scan a numeric constant 447 ⟩;
  if *negative* **then** *negate*(*cur_val*);
  end;

444. ⟨ Get the next non-blank non-sign token; set *negative* appropriately 444 ⟩ ≡
  negative ← *false*;
  repeat ⟨ Get the next non-blank non-call token 409 ⟩;
    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 443, 451, and 464.

445. 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* 445⟩ ≡
```
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 > 65535 then
  begin print_err("Improper␣alphabetic␣constant");
  help2("A␣one-character␣control␣sequence␣belongs␣after␣a␣`␣mark.")
  ("So␣I´m␣essentially␣inserting␣\0␣here."); cur_val ← "0"; back_error;
  end
else ⟨Scan an optional space 446⟩;
end
```
This code is used in section 443.

446. ⟨Scan an optional space 446⟩ ≡
```
begin get_x_token;
if cur_cmd ≠ spacer then back_input;
end
```
This code is used in sections 445, 451, 458, and 1203.

447. ⟨Scan a numeric constant 447⟩ ≡
```
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 448⟩;
if vacuous then ⟨Express astonishment that no number was here 449⟩
else if cur_cmd ≠ spacer then back_input;
end
```
This code is used in section 443.

448. **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 448 ⟩ ≡

```
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 447.

449. ⟨ Express astonishment that no number was here 449 ⟩ ≡

```
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 447.

450. 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* }

451. 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 2^{16} }
    ⟨ Local variables for dimension calculations 453 ⟩
  begin $f \leftarrow 0$; *arith_error* ← *false*; *cur_order* ← *normal*; *negative* ← *false*;
  if ¬*shortcut* **then**
    begin ⟨ Get the next non-blank non-sign token; set *negative* appropriately 444 ⟩;
    if (*cur_cmd* ≥ *min_internal*) ∧ (*cur_cmd* ≤ *max_internal*) **then**
      ⟨ Fetch an internal dimension and **goto** *attach_sign*, or fetch an internal integer 452 ⟩
    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 455 ⟩;
      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 $x \cdot (cur\_val + f/2^{16})$, where there are x sp per unit; **goto** *attach_sign* if the units are internal 456 ⟩;
  ⟨ Scan an optional space 446 ⟩;
attach_sign: **if** *arith_error* ∨ (*abs*(*cur_val*) ≥ '*10000000000*) **then**
    ⟨ Report that this dimension is out of range 463 ⟩;
  if *negative* **then** *negate*(*cur_val*);
  end;

452. ⟨ Fetch an internal dimension and **goto** *attach_sign*, or fetch an internal integer 452 ⟩ ≡
  if *mu* **then**
    begin *scan_something_internal*(*mu_val*, *false*); ⟨ Coerce glue to a dimension 454 ⟩;
    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 451.

453. ⟨ Local variables for dimension calculations 453 ⟩ ≡
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 451.

454. The following code is executed when *scan_something_internal* was called asking for *mu_val*, when we really wanted a "mudimen" instead of "muglue."

⟨ Coerce glue to a dimension 454 ⟩ ≡
  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 452 and 458.

455. 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.

⟨ Scan decimal fraction 455 ⟩ ≡
  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) $\lor$ (*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 451.

456. 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 456 ⟩ ≡
  if *inf* **then** ⟨ Scan for `fil` units; **goto** *attach_fraction* if found 457 ⟩;
  ⟨ Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 458 ⟩;
  if *mu* **then** ⟨ Scan for `mu` units and **goto** *attach_fraction* 459 ⟩;
  if *scan_keyword*(`"true"`) **then** ⟨ Adjust for the magnification ratio 460 ⟩;
  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 461 ⟩;
attach_fraction: **if** $cur\_val \geq {}'40000$ **then** *arith_error* ← *true*
  else *cur_val* ← *cur_val* ∗ *unity* + f;
done:

This code is used in section 451.

457. 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 457 ⟩ ≡
  if *scan_keyword*(`"fil"`) **then**
    begin *cur_order* ← *fil*;
    while *scan_keyword*(`"l"`) **do**
      begin if *cur_order* = *filll* **then**
        begin *print_err*(`"Illegal␣unit␣of␣measure␣("`); *print*(`"replaced␣by␣filll)"`);
        help1(`"I␣dddon´t␣go␣any␣higher␣than␣filll."`); *error*;
        end
      else *incr*(*cur_order*);
      end;
    goto *attach_fraction*;
    end

This code is used in section 456.

458. ⟨Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 458⟩ ≡

```
save_cur_val ← cur_val; ⟨Get the next non-blank non-call token 409⟩;
if (cur_cmd < min_internal) ∨ (cur_cmd > max_internal) then back_input
else begin if mu then
    begin scan_something_internal(mu_val, false); ⟨Coerce glue to a dimension 454⟩;
    if cur_val_level ≠ mu_val then mu_error;
    end
  else scan_something_internal(dimen_val, false);
  v ← cur_val; goto found;
  end;
if mu then goto not_found;
if scan_keyword("em") then v ← (⟨The em width for cur_font 561⟩)
else if scan_keyword("ex") then v ← (⟨The x-height for cur_font 562⟩)
  else goto not_found;
⟨Scan an optional space 446⟩;
found: cur_val ← nx_plus_y(save_cur_val, v, xn_over_d(v, f, '200000)); goto attach_sign;
not_found:
```

This code is used in section 456.

459. ⟨Scan for mu units and **goto** *attach_fraction* 459⟩ ≡

```
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 456.

460. ⟨Adjust for the magnification ratio 460⟩ ≡

```
begin prepare_mag;
if mag ≠ 1000 then
  begin cur_val ← xn_over_d(cur_val, 1000, mag); f ← (1000 ∗ f + '200000 ∗ remainder) div mag;
  cur_val ← cur_val + (f div '200000); f ← f mod '200000;
  end;
end
```

This code is used in section 456.

461. 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 dd ≈ 1000.33297 mm; this agrees well with the value 1000.333 mm cited by Bosshard in *Technische Grundlagen zur Satzherstellung* (Bern, 1980).

```
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 461⟩ ≡

```
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("sp") then goto done
              else ⟨Complain about unknown unit and goto done2 462⟩;
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 456.

462. ⟨Complain about unknown unit and **goto** *done2* 462⟩ ≡

```
begin print_err("Illegal␣unit␣of␣measure␣("); print("pt␣inserted)");
help6("Dimensions␣can␣be␣in␣units␣of␣em,␣ex,␣in,␣pt,␣pc,")
("cm,␣mm,␣dd,␣cc,␣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,␣it´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 461.

463. ⟨Report that this dimension is out of range 463⟩ ≡

```
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 451.

464. 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 444 ⟩;
  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 465 ⟩;
exit: **end**;

465. ⟨ Create a new glue specification whose width is *cur_val*; scan for its stretch and shrink components 465 ⟩ ≡
  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 464.

466. 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).

TCW: not intend to modify the function here; just append declarations of scanning routines for PUTeX.

define *default_rule* = 26214 { 0.4 pt }

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; ⟨ PUTeX basic scanning routines 1418 ⟩

467. 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.)

function *str_toks*(b : *pool_pointer*): *pointer*; { changes the string *str_pool*[b .. *pool_ptr*] to a token list }
  var p: *pointer*; { tail of the token list }
    q: *pointer*; { new node being added to the token list via *store_new_token* }
    t: *halfword*; { token being appended }
    k: *pool_pointer*; { index into *str_pool* }
  begin *str_room*(1); $p \leftarrow$ *temp_head*; *link*(p) $\leftarrow$ *null*; $k \leftarrow b$;
  while $k <$ *pool_ptr* **do**
    begin $t \leftarrow$ *so*(*str_pool*[k]);
    if $t > 128$ **then**
      begin $t \leftarrow t * 256 +$ *so*(*str_pool*[$k+1$]); *incr*(k);
      end;
    if $t =$ "␣" **then** $t \leftarrow$ *space_token*
    else $t \leftarrow$ *other_token* $+ t$;
    fast_store_new_token(t); *incr*(k);
    end;
  pool_ptr $\leftarrow b$; *str_toks* $\leftarrow p$;
  end;

468. 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`'.

TCW: make the function able to print CJK characters stored in local names table.

```
function the_toks: pointer;
  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 }
  begin get_x_token; char_val_flag ← false; scan_something_internal(tok_val, false);
  if cur_val_level ≥ ident_val then ⟨Copy the token list 469⟩
  else begin old_setting ← selector; selector ← new_string; b ← pool_ptr;
    case cur_val_level of
    int_val: if char_val_flag then
          if cur_val > 255 then print_wchar(cur_val)
          else { an empty slot }
        begin print_char("?"); print_char("?");
        end
    else 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;
  end;
```

469. ⟨Copy the token list 469⟩ ≡

```
  begin p ← temp_head; link(p) ← null;
  if cur_val_level = ident_val then store_new_token(cs_token_flag + cur_val)
  else if cur_val ≠ null then
      begin r ← link(cur_val); { do not copy the reference count }
      while r ≠ null do
        begin fast_store_new_token(info(r)); r ← link(r);
        end;
      end;
  the_toks ← p;
  end
```

This code is used in section 468.

470. Here's part of the *expand* subroutine that we are now ready to complete:

```
procedure ins_the_toks;
  begin link(garbage) ← the_toks; ins_list(link(temp_head));
  end;
```

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

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 *cnumber_code* = 5 { command code for `\PUXcnumber` }
define *scnumber_code* = 6 { command code for `\PUXscnumber` }
define *ucnumber_code* = 7 { command code for `\PUXucnumber` }
define *fcnumber_code* = 8 { command code for `\PUXfcnumber` }
define *acnumber_code* = 9 { command code for `\PUXacnumber` }
define *cjknumber_code* = 10 { command code for `\PUXcjknumber` }
define *nameseq_code* = 11 { command code for `\PUXnameseq` }
define *job_name_code* = 12 { command code for `\jobname` }
define *lower_cdigit_base* = 10 { lowercase style Chinese number }
define *upper_cdigit_base* = 25 { uppercase style Chinese number }

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("number", *convert*, *number_code*);
primitive("romannumeral", *convert*, *roman_numeral_code*);
primitive("string", *convert*, *string_code*);
primitive("meaning", *convert*, *meaning_code*);
primitive("fontname", *convert*, *font_name_code*);
primitive("jobname", *convert*, *job_name_code*);
primitive("PUXcnumber", *convert*, *cnumber_code*);
primitive("PUXscnumber", *convert*, *scnumber_code*);
primitive("PUXucnumber", *convert*, *ucnumber_code*);
primitive("PUXfcnumber", *convert*, *fcnumber_code*);
primitive("PUXacnumber", *convert*, *acnumber_code*);
primitive("PUXcjknumber", *convert*, *cjknumber_code*);
primitive("PUXnameseq", *convert*, *nameseq_code*);

472. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
convert: **case** *chr_code* **of**
number_code: *print_esc*("number");
roman_numeral_code: *print_esc*("romannumeral");
string_code: *print_esc*("string");
meaning_code: *print_esc*("meaning");
font_name_code: *print_esc*("fontname");
cnumber_code: *print_esc*("PUXcnumber");
scnumber_code: *print_esc*("PUXscnumber");
ucnumber_code: *print_esc*("PUXucnumber");
fcnumber_code: *print_esc*("PUXfcnumber");
acnumber_code: *print_esc*("PUXfanumber");
cjknumber_code: *print_esc*("PUXcjknumber");
nameseq_code: *print_esc*("PUXnameseq");
othercases *print_esc*("jobname")
endcases;

473. 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`'.

procedure *conv_toks*;
  var *old_setting*: 0 . . *max_selector*; { holds *selector* setting }
    c: *number_code* . . *job_name_code*; { desired type of conversion }
    save_scanner_status: *small_number*; { *scanner_status* upon entry }
    b: *pool_pointer*; { base of temporary string }
    dsize: *integer*; *saved_val*, *digit_base*, *sign*: *integer*; *min_val*, *max_val*, *offset*: *integer*;
  begin $c \leftarrow$ *cur_chr*; ⟨Scan the argument for command *c* 474⟩;
  old_setting ← *selector*; *selector* ← *new_string*; *b* ← *pool_ptr*; ⟨Print the result of command *c* 475⟩;
  selector ← *old_setting*; *link*(*garbage*) ← *str_toks*(*b*); *ins_list*(*link*(*temp_head*));
  end;

474. ⟨Scan the argument for command *c* 474⟩ ≡
  case *c* **of**
  number_code, *roman_numeral_code*, *cnumber_code*, *scnumber_code*, *ucnumber_code*, *fcnumber_code*:
      scan_int;
  acnumber_code: ⟨scan and split the number 1432⟩;
  cjknumber_code: ⟨scan a CJK number with a possible selector and then split it 1433⟩;
  nameseq_code: ⟨scan a CJK name sequence number 1437⟩;
  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*;
  job_name_code: **if** *job_name* = 0 **then** *open_log_file*;
  end { there are no other cases }

This code is used in section 473.

475. ⟨Print the result of command c 475⟩ ≡

case c **of**
number_code: *print_int*(*cur_val*);
roman_numeral_code: *print_roman_int*(*cur_val*);
cnumber_code: *print_chinese_int*(*cur_val*, *lower_cdigit_base*, *false*, *false*);
scnumber_code: *print_chinese_int*(*cur_val*, *lower_cdigit_base*, *true*, *false*);
ucnumber_code: *print_chinese_int*(*cur_val*, *upper_cdigit_base*, *false*, *false*);
fcnumber_code: *print_chinese_int*(*cur_val*, *upper_cdigit_base*, *false*, *true*);
acnumber_code: ⟨using full-width arabic characters to print a CJK number 1435⟩;
cjknumber_code: ⟨print a CJK number with specified format 1436⟩;
nameseq_code: ⟨print a CJK name sequence member 1438⟩;
string_code: **if** $cur\_cs \neq 0$ **then** *sprint_cs*(*cur_cs*)
  else if *is_wchar*(*cur_chr*) **then** *print_wchar*(*cur_chr*)
    else *print_char*(*cur_chr*);
meaning_code: *print_meaning*;
font_name_code: **begin if** $cur\_val \leq font\_max$ **then**
    begin *print*(*font_name*[*cur_val*]);
    if $font\_size[cur\_val] \neq font\_dsize[cur\_val]$ **then**
      begin *print*("␣at␣"); *print_scaled*(*font_size*[*cur_val*]); *print*("pt");
      end;
    end
  else begin *print*("CFONT"); *print*(*cface*[*cfont_face*[*cur_val*]]); $dsize \leftarrow cfont\_dsize[cur\_val]$ **div** ″10000;
    print_int(*dsize*);
    if $cfont\_size[cur\_val] \neq cfont\_dsize[cur\_val]$ **then**
      begin *print*("␣at␣"); *print_scaled*(*cfont_size*[*cur_val*]); *print*("pt");
      end;
    end;
  end;
job_name_code: *print*(*job_name*);
end {there are no other cases}

This code is used in section 473.

476. 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*, *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 477⟩
  else *scan_left_brace*; { remove the compulsory left brace }
  ⟨Scan and build the body of the token list; **goto** *found* when finished 480⟩;
found: *scanner_status* ← *normal*;
  if *hash_brace* ≠ 0 **then** *store_new_token*(*hash_brace*);
  scan_toks ← *p*;
  end;

477. ⟨Scan and build the parameter part of the macro definition 477⟩ ≡
  begin loop
    begin *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* 479⟩;
    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* 478⟩;
done: **end**

This code is used in section 476.

478. ⟨Express shock at the missing left brace; **goto** *found* 478⟩ ≡
  begin *print_err*("`Missing␣{␣inserted`"); *incr*(*align_state*);
  help2("`Where␣was␣the␣left␣brace?␣You␣said␣something␣like␣`\def\a}´,`")
  ("`which␣I´m␣going␣to␣interpret␣as␣`\def\a{}´.`"); *error*; **goto** *found*;
  end

This code is used in section 477.

479. ⟨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* 479⟩ ≡

```
begin s ← match_token + cur_chr; get_token;
if cur_cmd = left_brace 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");
  help1("I´m␣going␣to␣ignore␣the␣#␣sign␣you␣just␣used."); error;
  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´␣to␣delete␣what␣you␣did␣use."); back_error;
    end;
  cur_tok ← s;
  end;
end
```

This code is used in section 477.

480. ⟨Scan and build the body of the token list; **goto** *found* when finished 480⟩ ≡

```
unbalance ← 1;
loop begin if xpand then ⟨Expand the next part of the input 481⟩
  else 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
  else if cur_cmd = mac_param then
      if macro_def then ⟨Look for parameter number or ## 482⟩;
  store_new_token(cur_tok);
  end
```

This code is used in section 476.

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

⟨Expand the next part of the input 481⟩ ≡

```
begin loop
  begin get_next;
  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 480.

482. ⟨ Look for parameter number or `##` 482 ⟩ ≡
begin $s \leftarrow cur\_tok$;
if $xpand$ **then** get_x_token
else get_token;
if $cur\_cmd \neq mac\_param$ **then**
  if $(cur\_tok \le zero\_token) \lor (cur\_tok > t)$ **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*; $cur\_tok \leftarrow s$;
    end
  else $cur\_tok \leftarrow out\_param\_token - \text{"0"} + cur\_chr$;
end

This code is used in section 480.

483. 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 $read\_file[n]$ }

484. ⟨ Set initial values of key variables 21 ⟩ +≡
for $k \leftarrow 0$ **to** 16 **do** $read\_open[k] \leftarrow closed$;

485. 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*);
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 $scanner\_status \leftarrow defining$; $warning\_index \leftarrow r$; $def\_ref \leftarrow get\_avail$;
$token\_ref\_count(def\_ref) \leftarrow null$; $p \leftarrow def\_ref$; { the reference count }
store_new_token(*end_match_token*);
if $(n < 0) \lor (n > 15)$ **then** $m \leftarrow 16$ **else** $m \leftarrow n$;
$s \leftarrow align\_state$; $align\_state \leftarrow 1000000$; { disable tab marks, etc. }
repeat ⟨ Input and store tokens from the next line of the file 486 ⟩;
until $align\_state = 1000000$;
$cur\_val \leftarrow def\_ref$; $scanner\_status \leftarrow normal$; $align\_state \leftarrow s$;
end;

486. ⟨Input and store tokens from the next line of the file 486⟩ ≡

begin_file_reading; *name* ← $m + 1$;
if *read_open*[m] = *closed* **then** ⟨Input for `\read` from the terminal 487⟩
else if *read_open*[m] = *just_open* **then** ⟨Input the first line of *read_file*[m] 488⟩
  else ⟨Input the next line of *read_file*[m] 489⟩;
limit ← *last*;
if *end_line_char_inactive* **then** *decr*(*limit*)
else *buffer*[*limit*] ← *end_line_char*;
first ← *limit* + 1; *loc* ← *start*; *state* ← *new_line*;
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 485.

487. 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 487⟩ ≡

if *interaction* > *nonstop_mode* **then**
  if $n < 0$ **then** *prompt_input*("")
  else begin *wake_up_terminal*; *print_ln*; *sprint_cs*(*r*); *prompt_input*("="); $n \leftarrow -1$;
    end
else *fatal_error*("***␣(cannot␣\read␣from␣terminal␣in␣nonstop␣modes)")

This code is used in section 486.

488. 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] 488⟩ ≡

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 486.

489. An empty line is appended at the end of a *read_file*.

⟨Input the next line of *read_file*[m] 489⟩ ≡

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; *error*;
    end;
  end;
end

This code is used in section 486.

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

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

⟨Put each of TEX's primitives into the hash table 226⟩ +≡
primitive(`"if"`, *if_test*, *if_char_code*); *primitive*(`"ifcat"`, *if_test*, *if_cat_code*);
primitive(`"ifnum"`, *if_test*, *if_int_code*); *primitive*(`"ifdim"`, *if_test*, *if_dim_code*);
primitive(`"ifodd"`, *if_test*, *if_odd_code*); *primitive*(`"ifvmode"`, *if_test*, *if_vmode_code*);
primitive(`"ifhmode"`, *if_test*, *if_hmode_code*); *primitive*(`"ifmmode"`, *if_test*, *if_mmode_code*);
primitive(`"ifinner"`, *if_test*, *if_inner_code*); *primitive*(`"ifvoid"`, *if_test*, *if_void_code*);
primitive(`"ifhbox"`, *if_test*, *if_hbox_code*); *primitive*(`"ifvbox"`, *if_test*, *if_vbox_code*);
primitive(`"ifx"`, *if_test*, *ifx_code*); *primitive*(`"ifeof"`, *if_test*, *if_eof_code*);
primitive(`"iftrue"`, *if_test*, *if_true_code*); *primitive*(`"iffalse"`, *if_test*, *if_false_code*);
primitive(`"ifcase"`, *if_test*, *if_case_code*);

491. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
if_test: **case** *chr_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"`);
othercases *print_esc*(`"if"`)
endcases;

492. 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 }

493. ⟨ Set initial values of key variables 21 ⟩ +≡
cond_ptr ← *null*; *if_limit* ← *normal*; *cur_if* ← 0; *if_line* ← 0;

494. ⟨ Put each of TeX's primitives into the hash table 226 ⟩ +≡
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*);

495. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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");

496. 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 }

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

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

498. 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 498 ⟩ ≡

```
  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 501.

499. ⟨ Pop the condition stack 499 ⟩ ≡

```
  begin 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 501, 503, 512, and 513.

500. 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;
```

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

```
procedure conditional;
  label exit, common_ending;
  var b: boolean;  { is the condition true? }
    r: "<" .. ">";  { relation to be evaluated }
    m, n: integer;  { to be tested against the second operand }
    p, q: pointer;  { for traversing token lists in \ifx tests }
    save_scanner_status: small_number;  { scanner_status upon entry }
    save_cond_ptr: pointer;  { cond_ptr corresponding to this conditional }
    this_if: small_number;  { type of this conditional }
  begin ⟨Push the condition stack 498⟩; save_cond_ptr ← cond_ptr; this_if ← cur_chr;
  ⟨Either process \ifcase or set b to the value of a boolean condition 504⟩;
  if tracing_commands > 1 then ⟨Display the value of b 505⟩;
  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 503⟩;
common_ending: if cur_chr = fi_code then ⟨Pop the condition stack 499⟩
  else if_limit ← fi_code;  { wait for \fi }
exit: end;
```

502. 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.

503. ⟨Skip to `\else` or `\fi`, then **goto** *common_ending* 503⟩ ≡

```
  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 499⟩;
    end
```

This code is used in section 501.

504. ⟨ Either process `\ifcase` or set *b* to the value of a boolean condition 504 ⟩ ≡

```
case this_if of
if_char_code, if_cat_code: ⟨ Test if two characters match 509 ⟩;
if_int_code, if_dim_code: ⟨ Test relation between integers or dimensions 506 ⟩;
if_odd_code: ⟨ Test if an integer is odd 507 ⟩;
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 508 ⟩;
ifx_code: ⟨ Test if two tokens match 510 ⟩;
if_eof_code: begin scan_four_bit_int_or_18;
  if cur_val = 18 then b ← ¬shellenabledp
  else b ← (read_open[cur_val] = closed);
  end;
if_true_code: b ← true;
if_false_code: b ← false;
if_case_code: ⟨ Select the appropriate case and return or goto common_ending 512 ⟩;
end  { there are no other cases }
```

This code is used in section 501.

505. ⟨ Display the value of *b* 505 ⟩ ≡

```
begin begin_diagnostic;
if b then print("{true}") else print("{false}");
end_diagnostic(false);
end
```

This code is used in section 501.

506. Here we use the fact that `"<"`, `"="`, and `">"` are consecutive ASCII codes.

⟨ Test relation between integers or dimensions 506 ⟩ ≡

```
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 409 ⟩;
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_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 504.

507. ⟨ Test if an integer is odd 507 ⟩ ≡

```
begin scan_int; b ← odd(cur_val);
end
```

This code is used in section 504.

508. ⟨ Test box register status 508 ⟩ ≡
  begin *scan_eight_bit_int*; $p \leftarrow$ *box*(*cur_val*);
  if *this_if* = *if_void_code* **then** $b \leftarrow (p = null)$
  else if $p = null$ **then** $b \leftarrow$ *false*
    else if *this_if* = *if_hbox_code* **then** $b \leftarrow$ (*type*(p) = *hlist_node*)
      else $b \leftarrow$ (*type*(p) = *vlist_node*);
  end

This code is used in section 504.

509. 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

⟨ Test if two characters match 509 ⟩ ≡
  begin *get_x_token_or_active_char*;
  if (*cur_cmd* > *active_char*) ∨ (*cur_chr* > 65535) **then** { not a character }
    begin $m \leftarrow$ *relax*; $n \leftarrow 256$; { values other than 256 will break latex.fmt }
    end
  else begin $m \leftarrow$ *cur_cmd*; $n \leftarrow$ *cur_chr*;
    end;
  get_x_token_or_active_char;
  if (*cur_cmd* > *active_char*) ∨ (*cur_chr* > 65535) **then**
    begin *cur_cmd* ← *relax*; *cur_chr* ← 256; { values other than 256 will break latex.fmt }
    end;
  if *this_if* = *if_char_code* **then** $b \leftarrow$ (n = *cur_chr*) **else** $b \leftarrow$ (m = *cur_cmd*);
  end

This code is used in section 504.

510. 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 510 ⟩ ≡
  begin *save_scanner_status* ← *scanner_status*; *scanner_status* ← *normal*; *get_next*; $n \leftarrow$ *cur_cs*;
  $p \leftarrow$ *cur_cmd*; $q \leftarrow$ *cur_chr*; *get_next*;
  if *cur_cmd* ≠ p **then** $b \leftarrow$ *false*
  else if *cur_cmd* < *call* **then** $b \leftarrow$ (*cur_chr* = q)
    else ⟨ Test if two macro texts match 511 ⟩;
  scanner_status ← *save_scanner_status*;
  end

This code is used in section 504.

511. Note also that '`\ifx`' decides that macros `\a` and `\b` are different in examples like this:

```
\def\a{\c}    \def\c{}
\def\b{\d}    \def\d{}
```

⟨ Test if two macro texts match 511 ⟩ ≡

```
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 510.

512. ⟨ Select the appropriate case and **return** or **goto** *common_ending* 512 ⟩ ≡

```
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 499 ⟩;
  end;
change_if_limit(or_code, save_cond_ptr); return; { wait for \or, \else, or \fi }
end
```

This code is used in section 504.

513. 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` 513 ⟩ ≡

```
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 499 ⟩;
  end
```

This code is used in section 370.

514. 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.

515. 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 $c_1 \ldots c_n$, the system-independent driver program does the operations

$$\mathit{begin\_name};\ \mathit{more\_name}(c_1);\ \ldots\ ;\ \mathit{more\_name}(c_n);\ \mathit{end\_name}.$$

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 $\mathit{more\_name}(c_1)$, ..., $\mathit{more\_name}(c_{n-1})$. The final call $\mathit{more\_name}(c_n)$ 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 $\mathit{more\_name}(c_n)$ 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 `""` }

516. The file names we shall deal with have the following structure: If the name contains '/' or ':' (for Amiga only), 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 last '.' 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 '/', if any }
ext_delimiter: *pool_pointer*; { the most recent '.', if any }

517. 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.

In C, the default paths are specified separately.

518. Here now is the first of the system-dependent routines for file name scanning.

procedure *begin_name*;
  begin *area_delimiter* ← 0; *ext_delimiter* ← 0; *quoted_filename* ← *false*;
  end;

519. 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* = "␣") ∧ *stop_at_space* ∧ (¬*quoted_filename*) **then** *more_name* ← *false*
  else if *c* = """" **then**
      begin *quoted_filename* ← ¬*quoted_filename*; *more_name* ← *true*;
      end
    else begin *str_room*(1); *append_char*(*c*); { contribute *c* to the current string }
      if *IS_DIR_SEP*(*c*) **then**
        begin *area_delimiter* ← *cur_length*; *ext_delimiter* ← 0;
        end
      else if *c* = "." **then** *ext_delimiter* ← *cur_length*;
      more_name ← *true*;
      end;
  end;

520. The third. If a string is already in the string pool, the function *slow_make_string* does not create a new string but returns this string number, thus saving string space. Because of this new property of the returned string number it is not possible to apply *flush_string* to these strings.

```
procedure end_name;
  var temp_str: str_number; { result of file name cache lookups }
    j, s, t: pool_pointer; { running indices }
    must_quote: boolean; { whether we need to quote a string }
  begin if str_ptr + 3 > max_strings then overflow("number␣of␣strings", max_strings − init_str_ptr);
  str_room(6); { Room for quotes, if needed. }
    { add quotes if needed }
  if area_delimiter ≠ 0 then
    begin { maybe quote cur_area }
    must_quote ← false; s ← str_start[str_ptr]; t ← str_start[str_ptr] + area_delimiter; j ← s;
    while (¬must_quote) ∧ (j < t) do
      begin must_quote ← str_pool[j] = "␣"; incr(j);
      end;
    if must_quote then
      begin for j ← pool_ptr − 1 downto t do str_pool[j + 2] ← str_pool[j];
      str_pool[t + 1] ← """";
      for j ← t − 1 downto s do str_pool[j + 1] ← str_pool[j];
      str_pool[s] ← """";
      if ext_delimiter ≠ 0 then ext_delimiter ← ext_delimiter + 2;
      area_delimiter ← area_delimiter + 2; pool_ptr ← pool_ptr + 2;
      end;
    end; { maybe quote cur_name }
  s ← str_start[str_ptr] + area_delimiter;
  if ext_delimiter = 0 then t ← pool_ptr
  else t ← str_start[str_ptr] + ext_delimiter − 1;
  must_quote ← false; j ← s;
  while (¬must_quote) ∧ (j < t) do
    begin must_quote ← str_pool[j] = "␣"; incr(j);
    end;
  if must_quote then
    begin for j ← pool_ptr − 1 downto t do str_pool[j + 2] ← str_pool[j];
    str_pool[t + 1] ← """";
    for j ← t − 1 downto s do str_pool[j + 1] ← str_pool[j];
    str_pool[s] ← """";
    if ext_delimiter ≠ 0 then ext_delimiter ← ext_delimiter + 2;
    pool_ptr ← pool_ptr + 2;
    end;
  if ext_delimiter ≠ 0 then
    begin { maybe quote cur_ext }
    s ← str_start[str_ptr] + ext_delimiter − 1; t ← pool_ptr; must_quote ← false; j ← s;
    while (¬must_quote) ∧ (j < t) do
      begin must_quote ← str_pool[j] = "␣"; incr(j);
      end;
    if must_quote then
      begin str_pool[t + 1] ← """";
      for j ← t − 1 downto s do str_pool[j + 1] ← str_pool[j];
      str_pool[s] ← """"; pool_ptr ← pool_ptr + 2;
      end;
    end;
```

if $area\_delimiter = 0$ **then** $cur\_area \leftarrow$ ""
else begin $cur\_area \leftarrow str\_ptr$; $str\_start[str\_ptr + 1] \leftarrow str\_start[str\_ptr] + area\_delimiter$; $incr(str\_ptr)$;
  $temp\_str \leftarrow search\_string(cur\_area)$;
  if $temp\_str > 0$ **then**
    begin $cur\_area \leftarrow temp\_str$; $decr(str\_ptr)$; { no *flush_string*, *pool_ptr* will be wrong! }
    for $j \leftarrow str\_start[str\_ptr + 1]$ **to** $pool\_ptr - 1$ **do**
      begin $str\_pool[j - area\_delimiter] \leftarrow str\_pool[j]$;
      end;
    $pool\_ptr \leftarrow pool\_ptr - area\_delimiter$; { update *pool_ptr* }
    end;
  end;
if $ext\_delimiter = 0$ **then**
  begin $cur\_ext \leftarrow$ ""; $cur\_name \leftarrow slow\_make\_string$;
  end
else begin $cur\_name \leftarrow str\_ptr$;
  $str\_start[str\_ptr + 1] \leftarrow str\_start[str\_ptr] + ext\_delimiter - area\_delimiter - 1$; $incr(str\_ptr)$;
  $cur\_ext \leftarrow make\_string$; $decr(str\_ptr)$; { undo extension string to look at name part }
  $temp\_str \leftarrow search\_string(cur\_name)$;
  if $temp\_str > 0$ **then**
    begin $cur\_name \leftarrow temp\_str$; $decr(str\_ptr)$; { no *flush_string*, *pool_ptr* will be wrong! }
    for $j \leftarrow str\_start[str\_ptr + 1]$ **to** $pool\_ptr - 1$ **do**
      begin $str\_pool[j - ext\_delimiter + area\_delimiter + 1] \leftarrow str\_pool[j]$;
      end;
    $pool\_ptr \leftarrow pool\_ptr - ext\_delimiter + area\_delimiter + 1$; { update *pool_ptr* }
    end;
  $cur\_ext \leftarrow slow\_make\_string$; { remake extension string }
  end;
end;

521. 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.)

```
define check_quoted(#) ≡ {check if string # needs quoting}
        if # ≠ 0 then
          begin j ← str_start[#];
          while (¬must_quote) ∧ (j < str_start[# + 1]) do
            begin must_quote ← str_pool[j] = "␣"; incr(j);
            end;
          end
define print_quoted(#) ≡ {print string #, omitting quotes}
        if # ≠ 0 then
          for j ← str_start[#] to str_start[# + 1] − 1 do
            if so(str_pool[j]) ≠ """" then print(so(str_pool[j]))
```

⟨Basic printing procedures 57⟩ +≡

```
procedure print_file_name(n, a, e : integer);
  var must_quote: boolean; {whether to quote the filename}
    j: pool_pointer; {index into str_pool}
  begin must_quote ← false; check_quoted(a); check_quoted(n);
  check_quoted(e); {FIXME: Alternative is to assume that any filename that has to be quoted has at least
      one quoted component...if we pick this, a number of insertions of print_file_name should go away.
      must_quote:=((a¡¿0)and(str_pool[str_start[a]]=””””))or ((n¡¿0)and(str_pool[str_start[n]]=””””))or
      ((e¡¿0)and(str_pool[str_start[e]]=””””)); }
  if must_quote then print_char("""");
  print_quoted(a); print_quoted(n); print_quoted(e);
  if must_quote then print_char("""");
  end;
```

522. 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 ← #;
          if ¬(c = """") then
            begin incr(k);
            if k ≤ file_name_size then name_of_file[k] ← xchr[c];
            end
          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;
  if name_of_file then libc_free(name_of_file);
  name_of_file ← xmalloc_array(ASCII_code, length(a) + length(n) + length(e) + 1);
  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;
  name_of_file[name_length + 1] ← 0;
  end;
```

523. 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.

Under UNIX we don't give the area part, instead depending on the path searching that will happen during file opening. Also, the length will be set in the main program.

define *format_area_length* $= 0$ { 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 ⟩ +≡
format_default_length: *integer*;
TEX_format_default: *cstring*;

524. We set the name of the default format file and the length of that name in C, instead of Pascal, since we want them to depend on the name of the program.

525. ⟨ Check the "constant" values for consistency 14 ⟩ +≡
if *format_default_length* > *file_name_size* **then** *bad* ← 31;

526. 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*$[a \mathrel{..} b]$, 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**
    $b \leftarrow a +$ *file_name_size* $- n - 1 -$ *format_ext_length*;
  $k \leftarrow 0$;
  if *name_of_file* **then** *libc_free*(*name_of_file*);
  name_of_file ← *xmalloc_array*(*ASCII_code*, $n + (b - a + 1) +$ *format_ext_length* $+ 1$);
  for $j \leftarrow 1$ **to** n **do** *append_to_name*(*xord*[*ucharcast*(*TEX_format_default*[j])]);
  for $j \leftarrow a$ **to** b **do** *append_to_name*(*buffer*[j]);
  for $j \leftarrow$ *format_default_length* $-$ *format_ext_length* $+ 1$ **to** *format_default_length* **do**
    append_to_name(*xord*[*ucharcast*(*TEX_format_default*[j])]);
  if $k \le$ *file_name_size* **then** *name_length* ← k **else** *name_length* ← *file_name_size*;
  name_of_file[*name_length* + 1] ← 0;
  end;

527. 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* 527 ⟩ ≡
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); { Kpathsea does everything }
    if *w_open_in*(*fmt_file*) **then goto** *found*;
    wake_up_terminal; *wterm*(´Sorry,␣I␣can´´t␣find␣the␣format␣`´);
    fputs(*stringcast*(*name_of_file* + 1), *stdout*); *wterm*(´´´;␣will␣try␣`´);
    fputs(*TEX_format_default* + 1, *stdout*); *wterm_ln*(´´´.´); *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*(´I␣can´´t␣find␣the␣format␣file␣`´);
    fputs(*TEX_format_default* + 1, *stdout*); *wterm_ln*(´´´!´); *open_fmt_file* ← *false*; **return**;
    end;
found: *loc* ← *j*; *open_fmt_file* ← *true*;
exit: **end**;

This code is used in section 1306.

528. 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 }
    save_area_delimiter, save_ext_delimiter: pool_pointer;
    save_name_in_progress, save_stop_at_space: boolean;
  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;  { At this point we also set cur_name, cur_ext, and cur_area to
        match the contents of name_of_file. }
    save_area_delimiter ← area_delimiter; save_ext_delimiter ← ext_delimiter;
    save_name_in_progress ← name_in_progress; save_stop_at_space ← stop_at_space;
    name_in_progress ← true; begin_name; stop_at_space ← false; k ← 1;
    while (k ≤ name_length) ∧ (more_name(name_of_file[k])) do incr(k);
    stop_at_space ← save_stop_at_space; end_name; name_in_progress ← save_name_in_progress;
    area_delimiter ← save_area_delimiter; ext_delimiter ← save_ext_delimiter;
    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;
```

529. 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 409⟩;
  loop begin if (cur_cmd > other_char) ∨ (cur_chr > 255) then  { not a character }
      begin back_input; goto done;
      end;  { If cur_chr is a space and we're not scanning a token list, check whether we're at the end of
          the buffer. Otherwise we end up adding spurious spaces to file names in some cases. }
    if (cur_chr = "␣") ∧ (state ≠ token_list) ∧ (loc > limit) then goto done;
    if ¬more_name(cur_chr) then goto done;
    get_x_token;
    end;
done: end_name; name_in_progress ← false;
  end;
```

530. 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? }

531. 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*;

532. Here is a routine that manufactures the output file names, assuming that $job\_name \neq 0$. 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;

533. 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 }
    saved_cur_name: str_number; { to catch empty terminal input }
    saved_cur_ext: str_number; { to catch empty terminal input }
    saved_cur_area: str_number; { to catch empty terminal input }
  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") ∨ (e = "") then show_context;
  print_ln; print_c_string(prompt_file_name_help_msg);
  if (e ≠ "") then
    begin print(";␣default␣file␣extension␣is␣`"); print(e); print("´");
    end;
  print(")"); print_ln; print_nl("Please␣type␣another␣"); print(s);
  if interaction < scroll_mode then fatal_error("***␣(job␣aborted,␣file␣error␣in␣nonstop␣mode)");
  saved_cur_name ← cur_name; saved_cur_ext ← cur_ext; saved_cur_area ← cur_area; clear_terminal;
  prompt_input(":␣"); ⟨Scan file name in the buffer 534⟩;
  if (length(cur_name) = 0) ∧ (cur_ext = "") ∧ (cur_area = "") then
    begin cur_name ← saved_cur_name; cur_ext ← saved_cur_ext; cur_area ← saved_cur_area;
    end
  else if cur_ext = "" then cur_ext ← e;
  pack_cur_name;
  end;
```

534. ⟨Scan file name in the buffer 534⟩ ≡

```
  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 533.

535. 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 *log_name* ≡ *texmf_log_name*
define *ensure_dvi_open* ≡
  if *output_file_name* = 0 **then**
    begin if *job_name* = 0 **then** *open_log_file*;
    pack_job_name(".cdi");
    while ¬*b_open_out*(*dvi_file*) **do** *prompt_file_name*("file␣name␣for␣output", ".cdi");
    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 }

536. ⟨Initialize the output routines 55⟩ +≡
output_file_name ← 0;

537. 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: *const_cstring*;
begin *old_setting* ← *selector*;
if *job_name* = 0 **then** *job_name* ← *get_job_name*("texput");
pack_job_name(".fls"); *recorder_change_filename*(*stringcast*(*name_of_file* + 1)); *pack_job_name*(".log");
while ¬*a_open_out*(*log_file*) **do** ⟨Try to get a different log file name 538⟩;
log_name ← *a_make_name_string*(*log_file*); *selector* ← *log_only*; *log_opened* ← *true*;
⟨Print the banner line, including the date and time 539⟩;
if *mltex_enabled_p* **then**
  begin *wlog_cr*; *wlog*('MLTeX␣v2.2␣enabled');
  end;
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;

538. 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 538 ⟩ ≡

```
begin selector ← term_only; prompt_file_name("transcript␣file␣name", ".log");
end
```

This code is used in section 537.

539. ⟨ Print the banner line, including the date and time 539 ⟩ ≡

```
begin if src_specials_p ∨ file_line_error_style_p ∨ parse_first_line_p then wlog(banner_k)
else wlog(banner);
wlog(version_string); slow_print(format_ident); print("␣␣"); print_int(day); print_char("␣");
months ← ´␣JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
for k ← 3 ∗ month − 2 to 3 ∗ month do wlog(months[k]);
print_char("␣"); print_int(year); print_char("␣"); print_two(time div 60); print_char(":");
print_two(time mod 60);
if shellenabledp then
  begin wlog_cr; wlog(´␣´);
  if restrictedshell then
    begin wlog(´restricted␣´);
    end;
  wlog(´\write18␣enabled.´)
  end;
if src_specials_p then
  begin wlog_cr; wlog(´␣Source␣specials␣enabled.´)
  end;
if file_line_error_style_p then
  begin wlog_cr; wlog(´␣file:line:error␣style␣messages␣enabled.´)
  end;
if parse_first_line_p then
  begin wlog_cr; wlog(´␣%&-line␣parsing␣enabled.´);
  end;
if translate_filename then
  begin wlog_cr; wlog(´␣(´); fputs(translate_filename, log_file); wlog(´)´);
  end;
end
```

This code is used in section 537.

540. Let's turn now to the procedure that is used to initiate file reading when an '`\input`' command is being processed.

procedure *start_input*; { TeX will `\input` something }
  label *done*;
  var *temp_str*: *str_number*;
  begin *scan_file_name*; { set *cur_name* to desired file name }
  pack_cur_name;
  loop begin *begin_file_reading*; { set up *cur_file* and new level of input }
    tex_input_type ← 1; { Tell *open_input* we are `\input`. }
      { Kpathsea tries all the various ways to get the file. }
    if *kpse_in_name_ok*(*stringcast*(*name_of_file* + 1)) ∧ *a_open_in*(*cur_file*, *kpse_tex_format*) **then**
      goto *done*;
    end_file_reading; { remove the level that didn't work }
    prompt_file_name(`"input␣file␣name"`, `""`);
    end;
done: *name* ← *a_make_name_string*(*cur_file*); *source_filename_stack*[*in_open*] ← *name*;
  full_source_filename_stack[*in_open*] ← *make_full_name_string*;
  if *name* = *str_ptr* − 1 **then** { we can try to conserve string pool space now }
    begin *temp_str* ← *search_string*(*name*);
    if *temp_str* > 0 **then**
      begin *name* ← *temp_str*; *flush_string*;
      end;
    end;
  if *job_name* = 0 **then**
    begin *job_name* ← *get_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*(*full_source_filename_stack*[*in_open*]) > *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*(*full_source_filename_stack*[*in_open*]); *update_terminal*;
  state ← *new_line*; ⟨ Read the first line of the new file 541 ⟩;
  end;

541. 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 541 ⟩ ≡
  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 540.

542. 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*;

543. 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:

$$\begin{aligned}
lf &= \text{length of the entire file, in words;}\\
lh &= \text{length of the header data, in words;}\\
bc &= \text{smallest character code in the font;}\\
ec &= \text{largest character code in the font;}\\
nw &= \text{number of words in the width table;}\\
nh &= \text{number of words in the height table;}\\
nd &= \text{number of words in the depth table;}\\
ni &= \text{number of words in the italic correction table;}\\
nl &= \text{number of words in the lig/kern table;}\\
nk &= \text{number of words in the kern table;}\\
ne &= \text{number of words in the extensible character table;}\\
np &= \text{number of font parameter words.}
\end{aligned}$$

They are all nonnegative and less than 2^{15}. We must have $bc - 1 \le ec \le 255$, 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.

544. The rest of the `TFM` file may be regarded as a sequence of ten data arrays having the informal specification

$$\begin{aligned}
header &: \mathbf{array}\ [0 \mathrel{..} lh - 1]\ \mathbf{of}\ stuff\\
char_info &: \mathbf{array}\ [bc \mathrel{..} ec]\ \mathbf{of}\ char_info_word\\
width &: \mathbf{array}\ [0 \mathrel{..} nw - 1]\ \mathbf{of}\ fix_word\\
height &: \mathbf{array}\ [0 \mathrel{..} nh - 1]\ \mathbf{of}\ fix_word\\
depth &: \mathbf{array}\ [0 \mathrel{..} nd - 1]\ \mathbf{of}\ fix_word\\
italic &: \mathbf{array}\ [0 \mathrel{..} ni - 1]\ \mathbf{of}\ fix_word\\
lig_kern &: \mathbf{array}\ [0 \mathrel{..} nl - 1]\ \mathbf{of}\ lig_kern_command\\
kern &: \mathbf{array}\ [0 \mathrel{..} nk - 1]\ \mathbf{of}\ fix_word\\
exten &: \mathbf{array}\ [0 \mathrel{..} ne - 1]\ \mathbf{of}\ extensible_recipe\\
param &: \mathbf{array}\ [1 \mathrel{..} np]\ \mathbf{of}\ fix_word
\end{aligned}$$

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 $+16$.

545. 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.

546. 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 '`\/`' 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*.

547. 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 }

548. 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 $0 \le a \le b+c$ and $0 \le b, c \le 1$. 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 right 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 left boundary character, 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 $stop\_flag \equiv qi(128)$ { value indicating '`STOP`' in a lig/kern program }
define $kern\_flag \equiv qi(128)$ { op code for a kern step }
define $skip\_byte(\#) \equiv \#.b0$
define $next\_char(\#) \equiv \#.b1$
define $op\_byte(\#) \equiv \#.b2$
define $rem\_byte(\#) \equiv \#.b3$

549. 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^kMR^kB from top to bottom, for some $k \ge 0$, unless M is absent; in the latter case we can have TR^kB 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 $ext\_top(\#) \equiv \#.b0$ { *top* piece in a recipe }
define $ext\_mid(\#) \equiv \#.b1$ { *mid* piece in a recipe }
define $ext\_bot(\#) \equiv \#.b2$ { *bot* piece in a recipe }
define $ext\_rep(\#) \equiv \#.b3$ { *rep* piece in a recipe }

550. 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 `"␣"` 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

551. 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 = *integer*; { *font* in a *char_node* }
font_index = *integer*; { index into *font_info* }
nine_bits = *min_quarterword* . . *non_char*;

552. Here now is the (rather formidable) array of font arrays.

define *non_char* $\equiv$ *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: ↑*fmemory_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: ↑*four_quarters*; { check sum }
font_size: ↑*scaled*; { "at" size }
font_dsize: ↑*scaled*; { "design" size }
font_params: ↑*font_index*; { how many font parameters are present }
font_name: ↑*str_number*; { name of the font }
font_area: ↑*str_number*; { area of the font }
font_bc: ↑*eight_bits*; { beginning (smallest) character code }
font_ec: ↑*eight_bits*; { ending (largest) character code }
font_glue: ↑*pointer*; { glue specification for interword space, *null* if not allocated }
font_used: ↑*boolean*; { has a character from this font actually appeared in the output? }
hyphen_char: ↑*integer*; { current `\hyphenchar` values }
skew_char: ↑*integer*; { current `\skewchar` values }
bchar_label: ↑*font_index*;
{ start of *lig_kern* program for left boundary character, *non_address* if there is none }
font_bchar: ↑*nine_bits*; { right boundary character, *non_char* if there is none }
font_false_bchar: ↑*nine_bits*; { *font_bchar* if it doesn't exist in the font, otherwise *non_char* }

553. 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: ↑*integer*; { base addresses for *char_info* }
width_base: ↑*integer*; { base addresses for widths }
height_base: ↑*integer*; { base addresses for heights }
depth_base: ↑*integer*; { base addresses for depths }
italic_base: ↑*integer*; { base addresses for italic corrections }
lig_kern_base: ↑*integer*; { base addresses for ligature/kerning programs }
kern_base: ↑*integer*; { base addresses for kerns }
exten_base: ↑*integer*; { base addresses for extensible recipes }
param_base: ↑*integer*; { base addresses for font parameters }

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

555. $\TeX$ always knows at least one font, namely the null font. It has no characters, and its seven parameters are all equal to zero.

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

556. ⟨ Put each of $\TeX$'s primitives into the hash table 226 ⟩ +≡
primitive(`"nullfont"`, *set_font*, *null_font*); *text*(*frozen_null_font*) ← `"nullfont"`;
eqtb[*frozen_null_font*] ← *eqtb*[*cur_val*];

557. 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

$$font\_info[width\_base[f] + font\_info[char\_base[f] + c].qqqq.b0].sc$$

too often. The WEB definitions here make $char\_info(f)(c)$ the *four_quarters* word of font information corresponding to character c of font f. If q is such a word, $char\_width(f)(q)$ will be the character's width; hence the long formula above is at least abbreviated to

$$char\_width(f)(char\_info(f)(c)).$$

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 $char\_italic(f)(q)$, 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 $height\_depth(q)$ will be the 8-bit quantity

$$b = height\_index \times 16 + depth\_index,$$

and if b is such a byte we will write $char\_height(f)(b)$ and $char\_depth(f)(b)$ for the height and depth of the character c for which $q = char\_info(f)(c)$. Got that?

The tag field will be called $char\_tag(q)$; the remainder byte will be called $rem\_byte(q)$, 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.

MLTEX will assume that a character c exists iff either exists in the current font or a character substitution definition for this character was defined using `\charsubdef`. To avoid the distinction between these two cases, MLTEX introduces the notion "effective character" of an input character c. If c exists in the current font, the effective character of c is the character c itself. If it doesn't exist but a character substitution is defined, the effective character of c is the base character defined in the character substitution. If there is an effective character for a non-existing character c, the "virtual character" c will get appended to the horizontal lists.

The effective character is used within *char_info* to access appropriate character descriptions in the font. For example, when calculating the width of a box, MLTEX will use the metrics of the effective characters. For the case of a substitution, MLTEX uses the metrics of the base character, ignoring the metrics of the accent character.

If character substitutions are changed, it will be possible that a character c neither exists in a font nor there is a valid character substitution for c. To handle these cases *effective_char* should be called with its first argument set to *true* to ensure that it will still return an existing character in the font. If neither c nor the substituted base character in the current character substitution exists, *effective_char* will output a warning and return the character $font\_bc[f]$ (which is incorrect, but can not be changed within the current framework).

Sometimes character substitutions are unwanted, therefore the original definition of *char_info* can be used using the macro *orig_char_info*. Operations in which character substitutions should be avoided are, for example, loading a new font and checking the font metric information in this font, and character accesses in math mode.

define $char\_list\_exists(\#) \equiv (char\_sub\_code(\#) > hi(0))$
define $char\_list\_accent(\#) \equiv (ho(char\_sub\_code(\#))\ \mathbf{div}\ 256)$
define $char\_list\_char(\#) \equiv (ho(char\_sub\_code(\#))\ \mathbf{mod}\ 256)$
define $char\_info\_end(\#) \equiv \#\boxed{)}\ ].qqqq$
define $char\_info(\#) \equiv font\_info\ [\ char\_base[\#] + effective\_char\ \boxed{(}\ true, \#, char\_info\_end$
define $orig\_char\_info\_end(\#) \equiv \#\ ].qqqq$
define $orig\_char\_info(\#) \equiv font\_info\ [\ char\_base[\#] + orig\_char\_info\_end$

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)

558. 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 }

559. ⟨ 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*;

560. Here are some macros that help process ligatures and kerns. We write *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 $i =$ *font_info*[*lig_kern_start*(f)(j)] or *font_info*[*lig_kern_restart*(f)(i)], depending on whether or not *skip_byte*(i) ≤ *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*

561. Font parameters are referred to as *slant*(f), *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 }

⟨ The em width for *cur_font* 561 ⟩ ≡
quad(*cur_font*)

This code is used in section 458.

562. ⟨ The x-height for *cur_font* 562 ⟩ ≡
x_height(*cur_font*)

This code is used in section 458.

563. 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 2^{27} 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 }

⟨ Declare additional functions for MLTEX 1396 ⟩
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* }
name_too_long: *boolean*; { *nom* or *aire* exceeds 255 bytes? }
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; { right 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* 565 ⟩;
bad_tfm: ⟨ Report that the font won't be loaded 564 ⟩;
done: **if** *file_opened* **then** *b_close*(*tfm_file*);
read_font_info ← *g*;
end;

564. 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 564⟩ ≡

```
  start_font_error_message;
  if file_opened then print("␣not␣loadable:␣Bad␣metric␣(TFM)␣file")
  else if name_too_long then print("␣not␣loadable:␣Metric␣(TFM)␣file␣name␣too␣long")
    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 563.

565. ⟨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* 565⟩ ≡

⟨Open *tfm_file* for input 566⟩;
⟨Read the `TFM` size fields 568⟩;
⟨Use size fields to allocate font information 569⟩;
⟨Read the `TFM` header 571⟩;
⟨Read character data 572⟩;
⟨Read box dimensions 574⟩;
⟨Read ligature/kern program 576⟩;
⟨Read extensible character recipes 577⟩;
⟨Read font parameters 578⟩;
⟨Make final adjustments and **goto** *done* 579⟩

This code is used in section 563.

566. ⟨Open *tfm_file* for input 566⟩ ≡

```
  file_opened ← false; name_too_long ← (length(nom) > 255) ∨ (length(aire) > 255);
  if name_too_long then abort;  { kpse_find_file will append the ".tfm", and avoid searching the disk
        before the font alias files as well. }
  pack_file_name(nom, aire, "");
  if ¬b_open_in(tfm_file) then abort;
  file_opened ← true
```

This code is used in section 565.

567. 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 ≡ tfm_temp ← getc(tfm_file)
define fbyte ≡ tfm_temp
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
```

568. ⟨Read the `TFM` size fields 568⟩ ≡

```
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 565.

569. 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 569⟩ ≡

```
lf ← lf − 6 − 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 570⟩;
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 565.

570. ⟨Apologize for not loading the font, **goto** *done* 570⟩ ≡

```
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␣`I\font<same␣font␣id>=<name␣of␣loaded␣font>´."); error; goto done;
end
```

This code is used in section 569.

571. Only the first two words of the header are needed by $\TeX$82.

⟨Read the TFM header 571⟩ ≡
begin if $lh < 2$ **then** *abort*;
store_four_quarters(*font_check*[*f*]); *fget*; *read_sixteen*(*z*); { this rejects a negative design size }
fget; $z \leftarrow z * {'}400 + fbyte$; *fget*; $z \leftarrow (z * {'}20) + (fbyte$ **div** ${'}20)$;
if $z < unity$ **then** *abort*;
while $lh > 2$ **do**
  begin *fget*; *fget*; *fget*; *fget*; *decr*(*lh*); { ignore the rest of the header }
  end;
font_dsize[*f*] ← *z*;
if $s \neq -1000$ **then**
  if $s \geq 0$ **then** $z \leftarrow s$
  else $z \leftarrow xn\_over\_d(z, -s, 1000)$;
font_size[*f*] ← *z*;
end

This code is used in section 565.

572. ⟨Read character data 572⟩ ≡
for $k \leftarrow fmem\_ptr$ **to** $width\_base[f] - 1$ **do**
  begin *store_four_quarters*(*font_info*[*k*].*qqqq*);
  if $(a \geq nw) \lor (b$ **div** ${'}20 \geq nh) \lor (b$ **mod** ${'}20 \geq nd) \lor (c$ **div** $4 \geq ni)$ **then** *abort*;
  case *c* **mod** 4 **of**
  lig_tag: **if** $d \geq nl$ **then** *abort*;
  ext_tag: **if** $d \geq ne$ **then** *abort*;
  list_tag: ⟨Check for charlist cycle 573⟩;
  othercases *do_nothing* { *no_tag* }
  endcases;
  end

This code is used in section 565.

573. We want to make sure that there is no cycle of characters linked together by *list_tag* entries, since such a cycle would get $\TeX$ 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 $(\# < bc) \lor (\# > ec)$ **then** *abort*
    end
define *current_character_being_worked_on* ≡ $k + bc - fmem\_ptr$

⟨Check for charlist cycle 573⟩ ≡
begin *check_byte_range*(*d*);
while $d < current\_character\_being\_worked\_on$ **do**
  begin $qw \leftarrow orig\_char\_info(f)(d)$; { N.B.: not *qi*(*d*), since *char_base*[*f*] hasn't been adjusted yet }
  if $char\_tag(qw) \neq list\_tag$ **then goto** *not_found*;
  $d \leftarrow qo(rem\_byte(qw))$; { 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 572.

574. 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
```

```
⟨Read box dimensions 574⟩ ≡
  begin ⟨Replace z by z′ and compute α, β 575⟩;
  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 565.

575. ⟨Replace z by z' and compute α,β 575⟩ ≡

```
  begin alpha ← 16;
  while z ≥ ´40000000 do
    begin z ← z div 2; alpha ← alpha + alpha;
    end;
  beta ← 256 div alpha; alpha ← alpha * z;
  end
```

This code is used in sections 574 and 1441.

576. **define** *check_existence*(#) ≡
begin *check_byte_range*(#); $qw \gets orig\_char\_info(f)(\#)$; { N.B.: not *qi*(#) }
if $\neg char\_exists(qw)$ **then** *abort*;
end

⟨ Read ligature/kern program 576 ⟩ ≡
$bch\_label \gets '77777$; $bchar \gets 256$;
if $nl > 0$ **then**
begin for $k \gets lig\_kern\_base[f]$ **to** $kern\_base[f] + kern\_base\_offset - 1$ **do**
begin *store_four_quarters*(*font_info*[*k*].*qqqq*);
if $a > 128$ **then**
begin if $256 * c + d \ge nl$ **then** *abort*;
if $a = 255$ **then**
if $k = lig\_kern\_base[f]$ **then** $bchar \gets b$;
end
else begin if $b \ne bchar$ **then** *check_existence*(*b*);
if $c < 128$ **then** *check_existence*(*d*) { check ligature }
else if $256 * (c - 128) + d \ge nk$ **then** *abort*; { check kern }
if $a < 128$ **then**
if $k - lig\_kern\_base[f] + a + 1 \ge nl$ **then** *abort*;
end;
end;
if $a = 255$ **then** $bch\_label \gets 256 * c + d$;
end;
for $k \gets kern\_base[f] + kern\_base\_offset$ **to** $exten\_base[f] - 1$ **do** *store_scaled*(*font_info*[*k*].*sc*);

This code is used in section 565.

577. ⟨ Read extensible character recipes 577 ⟩ ≡
for $k \gets exten\_base[f]$ **to** $param\_base[f] - 1$ **do**
begin *store_four_quarters*(*font_info*[*k*].*qqqq*);
if $a \ne 0$ **then** *check_existence*(*a*);
if $b \ne 0$ **then** *check_existence*(*b*);
if $c \ne 0$ **then** *check_existence*(*c*);
check_existence(*d*);
end

This code is used in section 565.

578. 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 578 ⟩ ≡
begin for $k \gets 1$ **to** *np* **do**
if $k = 1$ **then** { the *slant* parameter is a pure number }
begin *fget*; $sw \gets fbyte$;
if $sw > 127$ **then** $sw \gets sw - 256$;
fget; $sw \gets sw * '400 + fbyte$; *fget*; $sw \gets sw * '400 + fbyte$; *fget*;
$font\_info[param\_base[f]].sc \gets (sw * '20) + (fbyte$ **div** $'20)$;
end
else *store_scaled*(*font_info*[*param_base*[*f*] + *k* − 1].*sc*);
if *feof*(*tfm_file*) **then** *abort*;
for $k \gets np + 1$ **to** 7 **do** $font\_info[param\_base[f] + k - 1].sc \gets 0$;
end

This code is used in section 565.

579. 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* 579 ⟩ ≡
  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* ← *orig_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 565.

580. Before we forget about the format of these tables, let's deal with two of T$_E$X's basic scanning routines related to font information.

TCW: handle the commands *def_cfont* and *set_cfont*.

⟨ Declare procedures that scan font-related stuff 580 ⟩ ≡
procedure *scan_font_ident*;
  var *f*: *integer*; *m*: *halfword*;
  begin ⟨ Get the next non-blank non-call token 409 ⟩;
  if *cur_cmd* = *def_font* **then** *f* ← *cur_font*
  else if *cur_cmd* = *set_font* ∨ *cur_cmd* = *set_cfont* **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 581.

This code is used in section 412.

581. 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 580⟩ +≡
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*; $n \leftarrow cur\_val$; *scan_font_ident*; $f \leftarrow cur\_val$;
  if $n \le 0$ **then** $cur\_val \leftarrow fmem\_ptr$
  else begin if $writing \land (n \le space\_shrink\_code) \land (n \ge space\_code) \land (font\_glue[f] \ne null)$ **then**
      begin *delete_glue_ref*(*font_glue*[*f*]); $font\_glue[f] \leftarrow null$;
      end;
    if $n > font\_params[f]$ **then**
      if $f < font\_ptr$ **then** $cur\_val \leftarrow fmem\_ptr$
      else ⟨Increase the number of parameters in the last font 583⟩
    else $cur\_val \leftarrow n + param\_base[f]$;
    end;
  ⟨Issue an error message if $cur\_val = fmem\_ptr$ 582⟩;
  end;

582. ⟨Issue an error message if $cur\_val = fmem\_ptr$ 582⟩ ≡
  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 581.

583. ⟨Increase the number of parameters in the last font 583⟩ ≡
  begin repeat if $fmem\_ptr = font\_mem\_size$ **then** *overflow*("font␣memory", *font_mem_size*);
    $font\_info[fmem\_ptr].sc \leftarrow 0$; *incr*(*fmem_ptr*); *incr*(*font_params*[*f*]);
  until $n = font\_params[f]$;
  $cur\_val \leftarrow fmem\_ptr - 1$; { this equals $param\_base[f] + font\_params[f]$ }
  end

This code is used in section 581.

584. 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*);
  begin if $tracing\_lost\_chars > 0$ **then**
    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;
  end;

585. 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.

This allows a character node to be used if there is an equivalent in the *char_sub_code* list.

function *new_character*(*f* : *internal_font_number*; *c* : *eight_bits*): *pointer*;
  label *exit*;
  var *p*: *pointer*; { newly allocated node }
    ec: *quarterword*; { effective character of *c* }
  begin *ec* ← *effective_char*(*false*, *f*, *qi*(*c*));
  if *font_bc*[*f*] ≤ *qo*(*ec*) **then**
    if *font_ec*[*f*] ≥ *qo*(*ec*) **then**
      if *char_exists*(*orig_char_info*(*f*)(*ec*)) **then** { N.B.: not *char_info* }
        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**;

586. 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.)

587. 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.

588. 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. TEX82 uses this command for characters in the range $128 \le c < 256$.

set2 129 $c[2]$. Same as *set1*, except that c is two bytes long, so it is in the range $0 \le c < 65536$. PUTEX uses this to typeset a CJK two-byte character.

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 $a \le 0$ or $b \le 0$, 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 $c_0[4]$ $c_1[4]$ $\ldots$ $c_9[4]$ $p[4]$. 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 TEX 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, $-128 \le b < 128$; 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 $-32768 \le b < 32768$.

right3 145 $b[3]$. Same as *right1*, except that b is a three-byte quantity in the range $-2^{23} \le b < 2^{23}$.

right4 146 $b[4]$. Same as *right1*, except that b is a four-byte quantity in the range $-2^{31} \le b < 2^{31}$.

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, $-128 \le b < 128$. 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, $-32768 \le b < 32768$.

w3 150 $b[3]$. Same as *w1*, but b is three bytes long, $-2^{23} \le b < 2^{23}$.

w4 151 $b[4]$. Same as *w1*, but b is four bytes long, $-2^{31} \le b < 2^{31}$.

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, $-128 \le b < 128$. 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, $-32768 \le b < 32768$.

x3 155 $b[3]$. Same as *x1*, but b is three bytes long, $-2^{23} \le b < 2^{23}$.

x4 156 $b[4]$. Same as *x1*, but b is four bytes long, $-2^{31} \le b < 2^{31}$.

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, $-128 \le a < 128$; 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 $-32768 \le a < 32768$.

down3 159 $a[3]$. Same as *down1*, except that a is a three-byte quantity in the range $-2^{23} \le a < 2^{23}$.

down4 160 $a[4]$. Same as *down1*, except that a is a four-byte quantity in the range $-2^{31} \le a < 2^{31}$.

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, $-128 \le a < 128$. 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, $-32768 \le a < 32768$.

y3 164 $a[3]$. Same as *y1*, but a is three bytes long, $-2^{23} \le a < 2^{23}$.

y4 165 $a[4]$. Same as *y1*, but a is four bytes long, $-2^{31} \le a < 2^{31}$.

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, $-128 \le a < 128$. 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, $-32768 \le a < 32768$.

z3 169 $a[3]$. Same as *z1*, but a is three bytes long, $-2^{23} \le a < 2^{23}$.

z4 170 $a[4]$. Same as *z1*, but a is four bytes long, $-2^{31} \le a < 2^{31}$.

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 $f \leftarrow 1$, ..., $f \leftarrow 63$, respectively.

fnt1 235 $k[1]$. Set $f \leftarrow k$. TEX82 uses this command for font numbers in the range $64 \le k < 256$.

fnt2 236 $k[2]$. Same as *fnt1*, except that k is two bytes long, so it is in the range $0 \le k < 65536$. 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 $0 \le k < 65536$.

xxx3 241 $k[3]$ $x[k]$. Like *xxx1*, but $0 \le k < 2^{24}$.

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 $0 \le k < 256$; 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 $0 \le k < 65536$.

fnt_def3 245 $k[3]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $0 \le k < 2^{24}$.

fnt_def4 246 $k[4]$ $c[4]$ $s[4]$ $d[4]$ $a[1]$ $l[1]$ $n[a+l]$. Define font k, where $-2^{31} \le k < 2^{31}$.

pre 247 $i[1]$ $c[1]$ $num[4]$ $den[4]$ $mag[4]$ $k[1]$ $x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameters i, c, 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.

cfnt 250 $k[2]$. Set $cf \leftarrow k$. PUTEX uses this command for CJK font numbers in the range $0 \le k < 65535$.

cfnt_def 251 $k[2]$ $l[1]$ $n[l]$ $c[1]$ $s[4]$ $ds[4]$ $wt[2]$ $y[1]$ $w[4]$ $h[4]$ $d[4]$ $fw[4]$ $fh[4]$ $fd[4]$. Define CJK font k, where $0 \le k < 65536$, see below.

Commands 252–255 are undefined at the present time.

589. **define** *set_char_0* = 0 { typeset character 0 and move right }
define *set1* = 128 { typeset a character and move right }
define *set2* = 129 { typeset a two-byte CJK character and move right }
define *set4* = 131 { typeset a four-byte CJK 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 }
define *cfnt* = 250 { set current chinese font }
define *cfnt_def* = 251 { define the meaning of a chinese font }

590. The preamble contains basic information about the file as a whole. As stated above, there are six parameters:

$$i[1]\ c[1]\ num[4]\ den[4]\ mag[4]\ k[1]\ x[k].$$

The i byte identifies `CDI` format; currently this byte is always set to 100. (Some day we will set $i = 101$, when `CDI` format makes another incompatible change—perhaps in the year 2048.)

The c byte identifies the default character code set of document. Currently, the following code value is defined:

0: USC2 (Unicode, not supported yet)
1: Big5 (Traditional Chinese used in Taiwan and Hong Kong)
2: GBK (Simplified Chinese used in PRC and Singapore)

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 7227pt = 254cm, and since TEX works with scaled points where there are 2^{16} sp in a point, TEX sets $num/den = (254 \cdot 10^5)/(7227 \cdot 2^{16}) = 25400000/473628672$.

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 $mag \cdot num/1000den$. 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 $0 \le k < 256$.

define *id_byte* $= 100$ { identifies the kind of `DVI` files described here }

591. 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 $mag \cdot s/1000d$ 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*.

592. 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 $0 \le x < \alpha b$ and $0 \le y < \alpha a$, where α is the number of pixels per DVI unit.

593. 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

$$\begin{array}{l} \mathit{post}\ p[4]\ \mathit{num}[4]\ \mathit{den}[4]\ \mathit{mag}[4]\ l[4]\ u[4]\ s[2]\ t[2] \\ \langle\,\text{font definitions}\,\rangle \\ \mathit{post\_post}\ q[4]\ i[1]\ \text{223's}[{\ge}4] \end{array}$$

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.

594. 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.)

595. 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? }
{ character and font in current *char_node* }
c: *quarterword*;
f: *internal_font_number*;
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 }

596. ⟨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;

597. 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 }

598. 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: ↑*eight_bits*; { buffer for `DVI` output }
half_buf: *integer*; { half of *dvi_buf_size* }
dvi_limit: *integer*; { end of the current half buffer }
dvi_ptr: *integer*; { 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* }

599. 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;

600. The actual output of *dvi_buf* $[a \mathrel{..} b]$ 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.

In C, we use a macro to call *fwrite* or *write* directly, writing all the bytes in one shot. Much better even than writing four bytes at a time.

601. To put a byte in the buffer without paying the cost of invoking a procedure each time, we use the macro *dvi_out*.

The length of *dvi_file* should not exceed `"7FFFFFFF`; we set *cur_s* ← −2 to prevent further `DVI` output causing infinite recursion.

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_ptr* > (`"7FFFFFFF` − *dvi_offset*) **then**
    begin *cur_s* ← −2; *fatal_error*(`"dvi␣length␣exceeds␣""7FFFFFFF"`);
    end;
  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;

602. 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* 602 ⟩ ≡
  if *dvi_limit* = *half_buf* **then** *write_dvi*(*half_buf*, *dvi_buf_size* − 1);
  if *dvi_ptr* > (`"7FFFFFFF` − *dvi_offset*) **then**
    begin *cur_s* ← −2; *fatal_error*(`"dvi␣length␣exceeds␣""7FFFFFFF"`);
    end;
  if *dvi_ptr* > 0 **then** *write_dvi*(0, *dvi_ptr* − 1)

This code is used in section 645.

603. The *dvi_four* procedure outputs four bytes in two's complement notation, without risking arithmetic overflow.

procedure *dvi_four*(*x* : *integer*);
  begin if $x \geq 0$ **then** *dvi_out*(*x* **div** ´*100000000*)
  else begin $x \leftarrow x +$ ´*10000000000*; $x \leftarrow x +$ ´*10000000000*; *dvi_out*((*x* **div** ´*100000000*) + 128);
    end;
  $x \leftarrow x$ **mod** ´*100000000*; *dvi_out*(*x* **div** ´*200000*); $x \leftarrow x$ **mod** ´*200000*; *dvi_out*(*x* **div** ´*400*);
  dvi_out(*x* **mod** ´*400*);
  end;

604. 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 ($l = \mathit{dvi\_offset} + \mathit{dvi\_ptr}$) $\wedge$ ($\mathit{dvi\_ptr} > 0$) **then** *decr*(*dvi_ptr*)
  else *dvi_out*(*pop*);
  end;

605. 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.

TCW: the procedure *dvi_cfont_def* outputs a chinese font definition.

procedure *dvi_font_def*(*f* : *internal_font_number*);
  var *k*: *pool_pointer*; { index into *str_pool* }
  begin if $f \leq 256 + \mathit{font\_base}$ **then**
    begin *dvi_out*(*fnt_def1*); *dvi_out*($f - \mathit{font\_base} - 1$);
    end
  else begin *dvi_out*(*fnt_def1* + 1); *dvi_out*(($f - \mathit{font\_base} - 1$) **div** ´*400*);
    dvi_out(($f - \mathit{font\_base} - 1$) **mod** ´*400*);
    end;
  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* 606 ⟩;
  end;
procedure *dvi_cfont_def*(*f* : *internal_cfont_number*);
  var *k*: *pool_pointer*; *j*: *integer*;
  begin $j \leftarrow$ *cfont_face*[*f*]; *dvi_out*(*cfnt_def*); *dvi_out*(($f - \mathit{cfont\_base} - 1$) **div** 256);
  dvi_out(($f - \mathit{cfont\_base} - 1$) **mod** 256); { Output the CJK font face name }
  dvi_out(*length*(*cface_name*[*j*]));
  for $k \leftarrow$ *str_start*[*cface_name*[*j*]] **to** *str_start*[*cface_name*[*j*] + 1] − 1 **do** *dvi_out*(*str_pool*[*k*]);
  dvi_out(*cface_charset*[*j*]); *dvi_four*(*cfont_size*[*f*]); *dvi_four*(*cfont_dsize*[*f*]);
  dvi_out(*cface_weight*[*j*] **div** 256); *dvi_out*(*cface_weight*[*j*] **mod** 256); *dvi_out*(*cface_style*[*j*]);
  dvi_four(*cfont_width*[*f*]); *dvi_four*(*cfont_height*[*f*]); *dvi_four*(*cfont_depth*[*f*]); *dvi_four*(*cface_fw_width*[*j*]);
  dvi_four(*cface_fw_height*[*j*]); *dvi_four*(*cface_fw_depth*[*j*]);
  end;

606. ⟨ Output the font name whose internal number is *f* 606 ⟩ ≡
  for $k \leftarrow$ *str_start*[*font_area*[*f*]] **to** *str_start*[*font_area*[*f*] + 1] − 1 **do** *dvi_out*(*so*(*str_pool*[*k*]));
  for $k \leftarrow$ *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 605.

607. 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 3 1 4 1 5 9 2 6 5 3 5 8 9, 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 ($1_y \ldots 1_y$ and $5_y \ldots 5_y \ldots 5_y$) and one z-hit ($3_z \ldots 3_z$); 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 $3_z\,1_y\,4_d\ldots$ in the example above were, in fact, produced by this procedure, as the reader can verify. (Go ahead and try it.)

608. 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 }

609. ⟨ Set initial values of key variables 21 ⟩ +≡
down_ptr ← *null*; *right_ptr* ← *null*;

610. 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: $y1 - down1 = w1 - right1$ and $z1 - down1 = x1 - right1$.

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 $q \leftarrow$ *get_node*(*movement_node_size*); { new node for the top of the stack }
  width(q) $\leftarrow w$; *location*(q) $\leftarrow$ *dvi_offset* + *dvi_ptr*;
  if $o =$ *down1* **then**
    begin *link*(q) $\leftarrow$ *down_ptr*; *down_ptr* $\leftarrow q$;
    end
  else begin *link*(q) $\leftarrow$ *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" 614 ⟩;
  ⟨ Generate a *down* or *right* command for w and **return** 613 ⟩;
found: ⟨ Generate a *y0* or *z0* command in order to reuse a previous appearance of w 612 ⟩;
exit: **end**;

611. 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 $\delta_y \ldots \delta_y$ 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 $2_y \ldots 2_y$ 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 }

612. 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 612⟩ ≡

```
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
```

This code is used in section 610.

613. ⟨Generate a *down* or *right* command for w and **return** 613⟩ ≡

```
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 610.

614. 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" 614 ⟩ ≡
$p \leftarrow link(q)$; $mstate \leftarrow none\_seen$;
while $p \neq null$ **do**
  begin if $width(p) = w$ **then** ⟨ Consider a node with matching width; **goto** *found* if it's a hit 615 ⟩
  else case $mstate + info(p)$ **of**
    $none\_seen + y\_here$: $mstate \leftarrow y\_seen$;
    $none\_seen + z\_here$: $mstate \leftarrow z\_seen$;
    $y\_seen + z\_here, z\_seen + y\_here$: **goto** *not_found*;
    othercases *do_nothing*
    endcases;
  $p \leftarrow link(p)$;
  end;
not_found:

This code is used in section 610.

615. 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 615 ⟩ ≡
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* 616 ⟩;
$none\_seen + z\_OK, y\_seen + yz\_OK, y\_seen + z\_OK$:
  if $location(p) < dvi\_gone$ **then goto** *not_found*
  else ⟨ Change buffered instruction to z or x and **goto** *found* 617 ⟩;
$none\_seen + y\_here, none\_seen + z\_here, y\_seen + z\_here, z\_seen + y\_here$: **goto** *found*;
othercases *do_nothing*
endcases

This code is used in section 614.

616. ⟨ Change buffered instruction to y or w and **goto** *found* 616 ⟩ ≡
begin $k \leftarrow location(p) - dvi\_offset$;
if $k < 0$ **then** $k \leftarrow k + dvi\_buf\_size$;
$dvi\_buf[k] \leftarrow dvi\_buf[k] + y1 - down1$; $info(p) \leftarrow y\_here$; **goto** *found*;
end

This code is used in section 615.

617. ⟨ Change buffered instruction to z or x and **goto** *found* 617 ⟩ ≡
begin $k \leftarrow location(p) - dvi\_offset$;
if $k < 0$ **then** $k \leftarrow k + dvi\_buf\_size$;
$dvi\_buf[k] \leftarrow dvi\_buf[k] + z1 - down1$; $info(p) \leftarrow z\_here$; **goto** *found*;
end

This code is used in section 615.

618. 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**;

619. 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.

  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 ⟩ +≡
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 }

620. ⟨Initialize variables as *ship_out* begins 620⟩ ≡

```
dvi_h ← 0; dvi_v ← 0; cur_h ← h_offset; dvi_f ← null_font; dvi_cf ← null_cfont; ensure_dvi_open;
if total_pages = 0 then
  begin dvi_out(pre); dvi_out(id_byte); doc_charset ← pux_charset; dvi_out(doc_charset);
      { output the preamble }
  dvi_four(25400000); dvi_four(473628672);   { conversion ratio for sp }
  prepare_mag; dvi_four(mag);   { magnification factor is frozen }
  if output_comment then
    begin l ← strlen(output_comment); dvi_out(l);
    for s ← 0 to l − 1 do dvi_out(output_comment[s]);
    end
  else begin   { the default code is unchanged }
    old_setting ← selector; selector ← new_string; print("␣PUTeX␣output␣"); print_int(year);
    print_char("."); print_two(month); print_char("."); print_two(day); print_char(":");
    print_two(time div 60); print_two(time mod 60); selector ← old_setting; dvi_out(cur_length);
    for s ← str_start[str_ptr] to pool_ptr − 1 do dvi_out(so(str_pool[s]));
    pool_ptr ← str_start[str_ptr];   { flush the current string }
    end;
  end
```

This code is used in section 643.

621. 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 }

622. 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* 1371 ⟩
procedure *hlist_out*; { output an *hlist_node* box }
  label *reswitch*, *move_past*, *fin_rule*, *next_p*, *continue*, *found*;
  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*; { left edge of sub-box, or right edge 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*; *base_line* ← *cur_v*; *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* 623 ⟩;
  prune_movements(*save_loc*);
  if *cur_s* > 0 **then** *dvi_pop*(*save_loc*);
  decr(*cur_s*);
  end;

623. 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 $set\_char\_0 = 0$.

In MLTEX this part looks for the existence of a substitution definition for a character c, if c does not exist in the font, and create appropriate `DVI` commands. Former versions of MLTEX have spliced appropriate character, kern, and box nodes into the horizontal list. Because the user can change character substitions or `\charsubdefmax` on the fly, we have to test a again for valid substitutions. (Additional it is necessary to be careful—if leaders are used the current hlist is normally traversed more than once!)

⟨Output node p for *hlist_out* and move to the next node, maintaining the condition $cur\_v = base\_line$ 623⟩ ≡

```
reswitch: if is_char_node(p) then
    begin synch_h; synch_v;
    repeat f ← font(p); c ← character(p);
      if (is_wchar(c)) then
        begin if f ≠ dvi_cf then ⟨Change font dvi_cf to f 1574⟩;
        dvi_out(set2); dvi_out(c div 256); dvi_out(c mod 256); cur_h ← cur_h + cfont_width[f];
        end
      else begin if f ≠ dvi_f then ⟨Change font dvi_f to f 624⟩;
        if font_ec[f] ≥ qo(c) then
          if font_bc[f] ≤ qo(c) then
            if char_exists(orig_char_info(f)(c)) then  { N.B.: not char_info }
              if c ≥ qi(128) then dvi_out(set1);
        dvi_out(qo(c));
        cur_h ← cur_h + char_width(f)(orig_char_info(f)(c)); goto continue;
        end;
      if mltex_enabled_p then ⟨Output a substitution, goto continue if not possible 1398⟩;
    continue: 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 625⟩
```

This code is used in section 622.

624. ⟨Change font *dvi_f* to f 624⟩ ≡

```
  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 if f ≤ 256 + font_base then
      begin dvi_out(fnt1); dvi_out(f − font_base − 1);
      end
    else begin dvi_out(fnt1 + 1); dvi_out((f − font_base − 1) div '400);
      dvi_out((f − font_base − 1) mod '400);
      end;
  dvi_f ← f;
  end
```

This code is used in section 623.

625. ⟨Output the non-*char_node* p for *hlist_out* and move to the next node 625⟩ ≡

begin case *type*(p) **of**
hlist_node, *vlist_node*: ⟨Output a box in an hlist 626⟩;
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 1370⟩;
glue_node: ⟨Move right or output leaders 628⟩;
kern_node, *math_node*: *cur_h* ← *cur_h* + *width*(p);
ligature_node: ⟨Make node p look like a *char_node* and **goto** *reswitch* 655⟩;
othercases *do_nothing*
endcases;
goto *next_p*;
fin_rule: ⟨Output a rule in an hlist 627⟩;
move_past: *cur_h* ← *cur_h* + *rule_wd*;
next_p: p ← *link*(p);
end

This code is used in section 623.

626. ⟨Output a box in an hlist 626⟩ ≡

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*;
  if *type*(p) = *vlist_node* **then** *vlist_out* **else** *hlist_out*;
  dvi_h ← *save_h*; *dvi_v* ← *save_v*; *cur_h* ← *edge* + *width*(p); *cur_v* ← *base_line*;
  end

This code is used in section 625.

627. ⟨Output a rule in an hlist 627⟩ ≡

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 625.

628. **define** *billion* $\equiv$ *float_constant*(1000000000)
define *vet_glue*(#) $\equiv$ *glue_temp* $\leftarrow$ #;
  if *glue_temp* $>$ *billion* **then** *glue_temp* $\leftarrow$ *billion*
  else if *glue_temp* $<$ $-$*billion* **then** *glue_temp* $\leftarrow$ $-$*billion*

⟨Move right or output leaders 628⟩ $\equiv$
```
begin g ← glue_ptr(p); rule_wd ← width(g) − cur_g;
if g_sign ≠ normal then
  begin if g_sign = stretching then
    begin if stretch_order(g) = g_order then
      begin cur_glue ← cur_glue + stretch(g); vet_glue(float(glue_set(this_box)) ∗ cur_glue);
      cur_g ← round(glue_temp);
      end;
    end
  else if shrink_order(g) = g_order then
      begin cur_glue ← cur_glue − shrink(g); vet_glue(float(glue_set(this_box)) ∗ cur_glue);
      cur_g ← round(glue_temp);
      end;
  end;
rule_wd ← rule_wd + cur_g;
if subtype(p) ≥ a_leaders then
  ⟨Output leaders in an hlist, goto fin_rule if a rule or to next_p if done 629⟩;
goto move_past;
end
```
This code is used in section 625.

629. ⟨Output leaders in an hlist, **goto** *fin_rule* if a rule or to *next_p* if done 629⟩ $\equiv$
```
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}
  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 630⟩;
  while cur_h + leader_wd ≤ edge do
    ⟨Output a leader box at cur_h, then advance cur_h by leader_wd + lx 631⟩;
  cur_h ← edge − 10; goto next_p;
  end;
end
```
This code is used in section 628.

630. 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 $left\_edge + leader\_wd \times \lceil (cur\_h - left\_edge)/leader\_wd \rceil$. 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 630⟩ ≡

```
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 section 629.

631. 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* 631⟩ ≡

```
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; 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 629.

632. 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 $cur\_g \leftarrow 0$; $cur\_glue \leftarrow float\_constant(0)$; $this\_box \leftarrow temp\_ptr$; $g\_order \leftarrow glue\_order(this\_box)$;
  $g\_sign \leftarrow glue\_sign(this\_box)$; $p \leftarrow list\_ptr(this\_box)$; *incr*(*cur_s*);
  if $cur\_s > 0$ **then** *dvi_out*(*push*);
  if $cur\_s > max\_push$ **then** $max\_push \leftarrow cur\_s$;
  $save\_loc \leftarrow dvi\_offset + dvi\_ptr$; $left\_edge \leftarrow cur\_h$; $cur\_v \leftarrow cur\_v - height(this\_box)$; $top\_edge \leftarrow cur\_v$;
  while $p \neq null$ **do** ⟨ Output node *p* for *vlist_out* and move to the next node, maintaining the condition
      $cur\_h = left\_edge$ 633 ⟩;
  prune_movements(*save_loc*);
  if $cur\_s > 0$ **then** *dvi_pop*(*save_loc*);
  decr(*cur_s*);
  end;

633. ⟨ Output node *p* for *vlist_out* and move to the next node, maintaining the condition
    $cur\_h = left\_edge$ 633 ⟩ ≡
  begin if *is_char_node*(*p*) **then** *confusion*(`"vlistout"`)
  else ⟨ Output the non-*char_node* *p* for *vlist_out* 634 ⟩;
next_p: $p \leftarrow link(p)$;
  end

This code is used in section 632.

634. $\langle$ Output the non-*char_node* p for *vlist_out* 634 $\rangle \equiv$

begin case $type(p)$ **of**
$hlist\_node, vlist\_node$: $\langle$ Output a box in a vlist 635 $\rangle$;
$rule\_node$: **begin** $rule\_ht \leftarrow height(p)$; $rule\_dp \leftarrow depth(p)$; $rule\_wd \leftarrow width(p)$; **goto** fin_rule;
end;
$whatsit\_node$: $\langle$ Output the whatsit node p in a vlist 1369 $\rangle$;
$glue\_node$: $\langle$ Move down or output leaders 637 $\rangle$;
$kern\_node$: $cur\_v \leftarrow cur\_v + width(p)$;
othercases $do\_nothing$
endcases;
goto $next\_p$;
fin_rule: $\langle$ Output a rule in a vlist, **goto** $next\_p$ 636 $\rangle$;
$move\_past$: $cur\_v \leftarrow cur\_v + rule\_ht$;
end

This code is used in section 633.

635. 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.

$\langle$ Output a box in a vlist 635 $\rangle \equiv$
if $list\_ptr(p) = null$ **then** $cur\_v \leftarrow cur\_v + height(p) + depth(p)$
else begin $cur\_v \leftarrow cur\_v + height(p)$; $synch\_v$; $save\_h \leftarrow dvi\_h$; $save\_v \leftarrow dvi\_v$;
$cur\_h \leftarrow left\_edge + shift\_amount(p)$; { shift the box right }
$temp\_ptr \leftarrow p$;
if $type(p) = vlist\_node$ **then** $vlist\_out$ **else** $hlist\_out$;
$dvi\_h \leftarrow save\_h$; $dvi\_v \leftarrow save\_v$; $cur\_v \leftarrow save\_v + depth(p)$; $cur\_h \leftarrow left\_edge$;
end

This code is used in section 634.

636. $\langle$ Output a rule in a vlist, **goto** $next\_p$ 636 $\rangle \equiv$

if $is\_running(rule\_wd)$ **then** $rule\_wd \leftarrow width(this\_box)$;
$rule\_ht \leftarrow rule\_ht + rule\_dp$; { this is the rule thickness }
$cur\_v \leftarrow cur\_v + rule\_ht$;
if $(rule\_ht > 0) \land (rule\_wd > 0)$ **then** { we don't output empty rules }
begin $synch\_h$; $synch\_v$; $dvi\_out(put\_rule)$; $dvi\_four(rule\_ht)$; $dvi\_four(rule\_wd)$;
end;
goto $next\_p$

This code is used in section 634.

637. ⟨ Move down or output leaders 637 ⟩ ≡

begin $g \gets glue\_ptr(p)$; $rule\_ht \gets width(g) - cur\_g$;
if $g\_sign \neq normal$ **then**
  begin if $g\_sign = stretching$ **then**
    begin if $stretch\_order(g) = g\_order$ **then**
      begin $cur\_glue \gets cur\_glue + stretch(g)$; $vet\_glue(float(glue\_set(this\_box)) * cur\_glue)$;
      $cur\_g \gets round(glue\_temp)$;
      end;
    end
  else if $shrink\_order(g) = g\_order$ **then**
      begin $cur\_glue \gets cur\_glue - shrink(g)$; $vet\_glue(float(glue\_set(this\_box)) * cur\_glue)$;
      $cur\_g \gets round(glue\_temp)$;
      end;
  end;
$rule\_ht \gets rule\_ht + cur\_g$;
if $subtype(p) \geq a\_leaders$ **then**
  ⟨ Output leaders in a vlist, **goto** *fin_rule* if a rule or to *next_p* if done 638 ⟩;
goto *move_past*;
end

This code is used in section 634.

638. ⟨ Output leaders in a vlist, **goto** *fin_rule* if a rule or to *next_p* if done 638 ⟩ ≡

begin $leader\_box \gets leader\_ptr(p)$;
if $type(leader\_box) = rule\_node$ **then**
  begin $rule\_wd \gets width(leader\_box)$; $rule\_dp \gets 0$; **goto** *fin_rule*;
  end;
$leader\_ht \gets height(leader\_box) + depth(leader\_box)$;
if $(leader\_ht > 0) \wedge (rule\_ht > 0)$ **then**
  begin $rule\_ht \gets rule\_ht + 10$; { compensate for floating-point rounding }
  $edge \gets cur\_v + rule\_ht$; $lx \gets 0$; ⟨ Let *cur_v* be the position of the first box, and set $leader\_ht + lx$ to the spacing between corresponding parts of boxes 639 ⟩;
  while $cur\_v + leader\_ht \leq edge$ **do**
    ⟨ Output a leader box at *cur_v*, then advance *cur_v* by $leader\_ht + lx$ 640 ⟩;
  $cur\_v \gets edge - 10$; **goto** *next_p*;
  end;
end

This code is used in section 637.

639. ⟨ Let *cur_v* be the position of the first box, and set $leader\_ht + lx$ to the spacing between corresponding parts of boxes 639 ⟩ ≡

if $subtype(p) = a\_leaders$ **then**
  begin $save\_v \gets cur\_v$; $cur\_v \gets top\_edge + leader\_ht * ((cur\_v - top\_edge)$ **div** $leader\_ht)$;
  if $cur\_v < save\_v$ **then** $cur\_v \gets cur\_v + leader\_ht$;
  end
else begin $lq \gets rule\_ht$ **div** $leader\_ht$; { the number of box copies }
  $lr \gets rule\_ht$ **mod** $leader\_ht$; { the remaining space }
  if $subtype(p) = c\_leaders$ **then** $cur\_v \gets cur\_v + (lr$ **div** $2)$
  else begin $lx \gets lr$ **div** $(lq + 1)$; $cur\_v \gets cur\_v + ((lr - (lq - 1) * lx)$ **div** $2)$;
    end;
  end

This code is used in section 638.

640. 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* 640 ⟩ ≡
begin *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 638.

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

procedure *ship_out*(*p* : *pointer*); { output the box *p* }
label *done*;
var *page_loc*: *integer*; { location of the current *bop* }
j, *k*: 0 . . 9; { indices to first ten count registers }
s: *pool_pointer*; { index into *str_pool* }
old_setting: 0 . . *max_selector*; { saved *selector* setting }
begin if *tracing_output* > 0 **then**
begin *print_nl*(""); *print_ln*; *print*("Completed␣box␣being␣shipped␣out");
end;
if *term_offset* > *max_print_line* − 9 **then** *print_ln*
else if (*term_offset* > 0) ∨ (*file_offset* > 0) **then** *print_char*("␣");
print_char("["); *j* ← 9;
while (*count*(*j*) = 0) ∧ (*j* > 0) **do** *decr*(*j*);
for *k* ← 0 **to** *j* **do**
begin *print_int*(*count*(*k*));
if *k* < *j* **then** *print_char*(".");
end;
update_terminal;
if *tracing_output* > 0 **then**
begin *print_char*("]"); *begin_diagnostic*; *show_box*(*p*); *end_diagnostic*(*true*);
end;
⟨ Ship box *p* out 643 ⟩;
if *tracing_output* ≤ 0 **then** *print_char*("]");
dead_cycles ← 0; *update_terminal*; { progress report }
⟨ Flush the box from memory, showing statistics if requested 642 ⟩;
end;

642. ⟨Flush the box from memory, showing statistics if requested 642⟩ ≡

```
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␣untouched:␣"); print_int(hi_mem_min − lo_mem_max − 1); print_ln;
  end;
tats
```

This code is used in section 641.

643. ⟨Ship box p out 643⟩ ≡

```
⟨Update the values of max_h and max_v; but if the page is too large, goto done 644⟩;
⟨Initialize variables as ship_out begins 620⟩;
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; ifdef(´IPC´)
  if ipc_on > 0 then
    begin if dvi_limit = half_buf then
      begin write_dvi(half_buf, dvi_buf_size − 1); flush_dvi; dvi_gone ← dvi_gone + half_buf;
      end;
    if dvi_ptr > (″7FFFFFFF − dvi_offset) then
      begin cur_s ← −2; fatal_error("dvi␣length␣exceeds␣""7FFFFFFF");
      end;
    if dvi_ptr > 0 then
      begin write_dvi(0, dvi_ptr − 1); flush_dvi; dvi_offset ← dvi_offset + dvi_ptr;
      dvi_gone ← dvi_gone + dvi_ptr;
      end;
    dvi_ptr ← 0; dvi_limit ← dvi_buf_size; ipc_page(dvi_gone);
    end;
endif(´IPC´);
done:
```

This code is used in section 641.

644. 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.

⟨Update the values of *max_h* and *max_v*; but if the page is too large, **goto** *done* 644⟩ ≡
if $(height(p) > max\_dimen) \lor (depth(p) > max\_dimen) \lor$
$(height(p) + depth(p) + v\_offset > max\_dimen) \lor (width(p) + h\_offset > max\_dimen)$ **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 $tracing\_output \le 0$ **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** $max\_v \gets height(p) + depth(p) + v\_offset$;
if $width(p) + h\_offset > max\_h$ **then** $max\_h \gets width(p) + h\_offset$

This code is used in section 643.

645. 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* ≥ 65536, the `DVI` file will lie. And if *max_push* ≥ 65536, the user deserves whatever chaos might ensue.

An integer variable k will be declared for use by this routine.

```
⟨Finish the DVI file 645⟩ ≡
  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␣pages␣of␣output.")
  else if cur_s ≠ −2 then
      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 646⟩;
      ⟨Output the CJK font definitions for all fonts that were used 1573⟩;
      dvi_out(post_post); dvi_four(last_bop); dvi_out(doc_charset); dvi_out(id_byte);
      ifdef(´IPC´)k ← 7 − ((3 + dvi_offset + dvi_ptr) mod 4); { the number of 223’s }
      endif(´IPC´)ifndef(´IPC´)k ← 4 + ((dvi_buf_size − dvi_ptr) mod 4); { the number of 223’s }
      endifn(´IPC´)
        while k > 0 do
          begin dvi_out(223); decr(k);
          end;
      ⟨Empty the last bytes out of dvi_buf 602⟩;
      print_nl("Output␣written␣on␣"); print_file_name(0, output_file_name, 0); print("␣(");
      print_int(total_pages);
      if total_pages ≠ 1 then print("␣pages")
      else print("␣page");
      print(",␣"); print_int(dvi_offset + dvi_ptr); print("␣bytes)."); b_close(dvi_file);
      end
```

This code is used in section 1336.

646. ⟨Output the font definitions for all fonts that were used 646⟩ ≡

```
  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 645.

647. 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 $natural \equiv 0, additional$ { shorthand for parameters to *hpack* and *vpack* }

648. 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:

$$save\_ptr \leftarrow save\_ptr - 2;$$
$$hpack(p, saved(1), saved(0)).$$

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** $s \leftarrow saved(0)$;
  if *scan_keyword*("to") **then** $spec\_code \leftarrow exactly$
  else if *scan_keyword*("spread") **then** $spec\_code \leftarrow additional$
    else begin $spec\_code \leftarrow additional$; $cur\_val \leftarrow 0$; **goto** *found*;
      end;
  scan_normal_dimen;
found: **if** *three_codes* **then**
    begin $saved(0) \leftarrow s$; $incr(save\_ptr)$;
    end;
  $saved(0) \leftarrow spec\_code$; $saved(1) \leftarrow cur\_val$; $save\_ptr \leftarrow save\_ptr + 2$; $new\_save\_level(c)$; *scan_left_brace*;
  end;

649. 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 }

650. 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 }

651. ⟨ Set initial values of key variables 21 ⟩ +≡
adjust_tail ← *null*; *last_badness* ← 0;

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

function *hpack*(*p* : *pointer*; *w* : *scaled*; *m* : *small_number*): *pointer*;
  label *reswitch*, *common_ending*, *exit*;
  var *r*: *pointer*; { the box node that will be returned }
    q: *pointer*; { trails behind *p* }
    h, *d*, *x*: *scaled*; { height, depth, and natural width }
    s: *scaled*; { shift amount }
    g: *pointer*; { points to a glue specification }
    o: *glue_ord*; { order of infinity }
    f: *internal_font_number*; { the font in a *char_node* }
    i: *four_quarters*; { font information about a *char_node* }
    hd: *eight_bits*; { height and depth indices for a character }
  begin *last_badness* ← 0; *r* ← *get_node*(*box_node_size*); *type*(*r*) ← *hlist_node*;
  subtype(*r*) ← *min_quarterword*; *shift_amount*(*r*) ← 0; *q* ← *r* + *list_offset*; *link*(*q*) ← *p*;
  h ← 0; ⟨ Clear dimensions to zero 653 ⟩;
  while *p* ≠ *null* **do** ⟨ Examine node *p* in the hlist, taking account of its effect on the dimensions of the new box, or moving it to the adjustment list; then advance *p* to the next node 654 ⟩;
  if *adjust_tail* ≠ *null* **then** *link*(*adjust_tail*) ← *null*;
  height(*r*) ← *h*; *depth*(*r*) ← *d*;
  ⟨ Determine the value of *width*(*r*) and the appropriate glue setting; then **return** or **goto** *common_ending* 660 ⟩;
common_ending: ⟨ Finish issuing a diagnostic message for an overfull or underfull hbox 666 ⟩;
exit: *hpack* ← *r*;
  end;

653. ⟨Clear dimensions to zero 653⟩ ≡
$d \leftarrow 0$; $x \leftarrow 0$; $total\_stretch[normal] \leftarrow 0$; $total\_shrink[normal] \leftarrow 0$; $total\_stretch[fil] \leftarrow 0$; $total\_shrink[fil] \leftarrow 0$; $total\_stretch[fill] \leftarrow 0$; $total\_shrink[fill] \leftarrow 0$; $total\_stretch[filll] \leftarrow 0$; $total\_shrink[filll] \leftarrow 0$

This code is used in sections 652 and 671.

654. ⟨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 654⟩ ≡
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 657⟩;
if $p \neq null$ **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 656⟩;
  ins_node, *mark_node*, *adjust_node*: **if** $adjust\_tail \neq null$ **then**
    ⟨Transfer node p to the adjustment list 658⟩;
  whatsit_node: ⟨Incorporate a whatsit node into an hbox 1363⟩;
  glue_node: ⟨Incorporate glue into the horizontal totals 659⟩;
  kern_node, *math_node*: $x \leftarrow x + width(p)$;
  ligature_node: ⟨Make node p look like a *char_node* and **goto** *reswitch* 655⟩;
  othercases *do_nothing*
  endcases;
  $p \leftarrow link(p)$;
  end;
end

This code is used in section 652.

655. ⟨Make node p look like a *char_node* and **goto** *reswitch* 655⟩ ≡
begin $mem[lig\_trick] \leftarrow mem[lig\_char(p)]$; $link(lig\_trick) \leftarrow link(p)$; $p \leftarrow lig\_trick$; **goto** *reswitch*;
end

This code is used in sections 625, 654, and 1150.

656. 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 656⟩ ≡
begin $x \leftarrow x + width(p)$;
if $type(p) \geq rule\_node$ **then** $s \leftarrow 0$ **else** $s \leftarrow shift\_amount(p)$;
if $height(p) - s > h$ **then** $h \leftarrow height(p) - s$;
if $depth(p) + s > d$ **then** $d \leftarrow depth(p) + s$;
end

This code is used in section 654.

657. 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 657⟩ ≡
begin $f \leftarrow font(p)$; $c \leftarrow character(p)$;
if $(is\_wchar(c))$ **then**
  begin $x \leftarrow x + cfont\_width[f]$;
  $s \leftarrow cfont\_height[f]$; **if** $s > h$ **then** $h \leftarrow s$;
  $s \leftarrow cfont\_depth[f]$; **if** $s > d$ **then** $d \leftarrow s$;
  end
else begin $i \leftarrow char\_info(f)(c)$; $hd \leftarrow height\_depth(i)$; $x \leftarrow x + char\_width(f)(i)$;
  $s \leftarrow char\_height(f)(hd)$; **if** $s > h$ **then** $h \leftarrow s$;
  $s \leftarrow char\_depth(f)(hd)$; **if** $s > d$ **then** $d \leftarrow s$;
  end;
$p \leftarrow link(p)$;
end

This code is used in section 654.

658. 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.

⟨Transfer node p to the adjustment list 658⟩ ≡
begin while $link(q) \neq p$ **do** $q \leftarrow link(q)$;
if $type(p) = adjust\_node$ **then**
  begin $link(adjust\_tail) \leftarrow adjust\_ptr(p)$;
  while $link(adjust\_tail) \neq null$ **do** $adjust\_tail \leftarrow link(adjust\_tail)$;
  $p \leftarrow link(p)$; $free\_node(link(q), small\_node\_size)$;
  end
else begin $link(adjust\_tail) \leftarrow p$; $adjust\_tail \leftarrow p$; $p \leftarrow link(p)$;
  end;
$link(q) \leftarrow p$; $p \leftarrow q$;
end

This code is used in section 654.

659. ⟨Incorporate glue into the horizontal totals 659⟩ ≡
begin $g \leftarrow glue\_ptr(p)$; $x \leftarrow x + width(g)$;
$o \leftarrow stretch\_order(g)$; $total\_stretch[o] \leftarrow total\_stretch[o] + stretch(g)$; $o \leftarrow shrink\_order(g)$;
$total\_shrink[o] \leftarrow total\_shrink[o] + shrink(g)$;
if $subtype(p) \geq a\_leaders$ **then**
  begin $g \leftarrow leader\_ptr(p)$;
  if $height(g) > h$ **then** $h \leftarrow height(g)$;
  if $depth(g) > d$ **then** $d \leftarrow depth(g)$;
  end;
end

This code is used in section 654.

660. 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* 660 ⟩ ≡
if $m = additional$ **then** $w \leftarrow x + w$;
width(r) ← w; $x \leftarrow w - x$; { 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* 661 ⟩
  else ⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 667 ⟩

This code is used in section 652.

661. ⟨ Determine horizontal glue stretch setting, then **return** or **goto** *common_ending* 661 ⟩ ≡
begin ⟨ Determine the stretch order 662 ⟩;
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 663 ⟩;
return;
end

This code is used in section 660.

662. ⟨ Determine the stretch order 662 ⟩ ≡
if *total_stretch*[*filll*] ≠ 0 **then** $o \leftarrow$ *filll*
else if *total_stretch*[*fill*] ≠ 0 **then** $o \leftarrow$ *fill*
  else if *total_stretch*[*fil*] ≠ 0 **then** $o \leftarrow$ *fil*
    else $o \leftarrow$ *normal*

This code is used in sections 661, 676, and 799.

663. ⟨ Report an underfull hbox and **goto** *common_ending*, if this box is sufficiently bad 663 ⟩ ≡
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 661.

664. 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 }

665. ⟨ Set initial values of key variables 21 ⟩ +≡
pack_begin_line ← 0;

666. ⟨Finish issuing a diagnostic message for an overfull or underfull hbox 666⟩ ≡

```
if output_active then print(") has occurred while \output is active")
else begin if pack_begin_line ≠ 0 then
    begin if pack_begin_line > 0 then print(") in paragraph at lines ")
    else print(") in alignment at lines ");
    print_int(abs(pack_begin_line)); print("--");
    end
  else print(") detected at line ");
  print_int(line);
  end;
print_ln;
font_in_short_display ← null_font; cfont_in_short_display ← null_cfont;
short_display(list_ptr(r)); print_ln;
begin_diagnostic; show_box(r); end_diagnostic(true)
```

This code is used in section 652.

667. ⟨Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 667⟩ ≡

```
begin ⟨Determine the shrink order 668⟩;
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 669⟩;
  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 670⟩;
return;
end
```

This code is used in section 660.

668. ⟨Determine the shrink order 668⟩ ≡

```
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 667, 679, and 799.

669. ⟨Report an overfull hbox and **goto** *common_ending*, if this box is sufficiently bad 669⟩ ≡

```
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 \hbox ("); print_scaled(−x − total_shrink[normal]);
  print("pt too wide"); goto common_ending;
  end
```

This code is used in section 667.

670. ⟨Report a tight hbox and **goto** *common_ending*, if this box is sufficiently bad 670⟩ ≡
  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 667.

671. 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 653⟩;
  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 672⟩;
  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* 675⟩;
common_ending: ⟨Finish issuing a diagnostic message for an overfull or underfull vbox 678⟩;
exit: *vpackage* ← *r*;
  end;

672. ⟨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 672⟩ ≡
  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 673⟩;
    whatsit_node: ⟨Incorporate a whatsit node into a vbox 1362⟩;
    glue_node: ⟨Incorporate glue into the vertical totals 674⟩;
    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 671.

673. ⟨Incorporate box dimensions into the dimensions of the vbox that will contain it 673⟩ ≡
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 672.

674. ⟨Incorporate glue into the vertical totals 674⟩ ≡
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 672.

675. When we get to the present part of the program, x is the natural height of the box being packaged.

⟨Determine the value of $height(r)$ and the appropriate glue setting; then **return** or **goto** *common_ending* 675⟩ ≡
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** ⟨Determine vertical glue stretch setting, then **return** or **goto** *common_ending* 676⟩
  else ⟨Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 679⟩

This code is used in section 671.

676. ⟨Determine vertical glue stretch setting, then **return** or **goto** *common_ending* 676⟩ ≡
begin ⟨Determine the stretch order 662⟩;
$glue\_order(r) \leftarrow o$; $glue\_sign(r) \leftarrow stretching$;
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**
    ⟨Report an underfull vbox and **goto** *common_ending*, if this box is sufficiently bad 677⟩;
return;
end

This code is used in section 675.

677. ⟨Report an underfull vbox and **goto** *common_ending*, if this box is sufficiently bad 677⟩ ≡
begin $last\_badness \leftarrow badness(x, total\_stretch[normal])$;
if $last\_badness > vbadness$ **then**
begin *print_ln*;
if $last\_badness > 100$ **then** *print_nl*("`Underfull`") **else** *print_nl*("`Loose`");
print("`␣\vbox␣(badness␣`"); *print_int*(*last_badness*); **goto** *common_ending*;
end;
end

This code is used in section 676.

678. ⟨Finish issuing a diagnostic message for an overfull or underfull vbox 678⟩ ≡
if *output_active* **then** *print*("`)␣has␣occurred␣while␣\output␣is␣active`")
else begin if $pack\_begin\_line \neq 0$ **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 671.

679. ⟨Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 679⟩ ≡
begin ⟨Determine the shrink order 668⟩;
$glue\_order(r) \leftarrow o$; $glue\_sign(r) \leftarrow shrinking$;
if $total\_shrink[o] \neq 0$ **then** $glue\_set(r) \leftarrow unfloat((-x)/total\_shrink[o])$
else begin $glue\_sign(r) \leftarrow normal$; *set_glue_ratio_zero*(*glue_set*(*r*)); { there's nothing to shrink }
end;
if $(total\_shrink[o] < -x) \land (o = normal) \land (list\_ptr(r) \neq null)$ **then**
begin $last\_badness \leftarrow 1000000$; *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 680⟩;
end
else if $o = normal$ **then**
if $list\_ptr(r) \neq null$ **then**
⟨Report a tight vbox and **goto** *common_ending*, if this box is sufficiently bad 681⟩;
return;
end

This code is used in section 675.

680. ⟨Report an overfull vbox and **goto** *common_ending*, if this box is sufficiently bad 680⟩ ≡
if $(-x - total\_shrink[normal] > vfuzz) \lor (vbadness < 100)$ **then**
begin *print_ln*; *print_nl*("`Overfull␣\vbox␣(`"); $print\_scaled(-x - total\_shrink[normal])$;
print("`pt␣too␣high`"); **goto** *common_ending*;
end

This code is used in section 679.

681. ⟨Report a tight vbox and **goto** *common_ending*, if this box is sufficiently bad 681⟩ ≡
begin $last\_badness \leftarrow badness(-x, total\_shrink[normal])$;
if $last\_badness > vbadness$ **then**
begin *print_ln*; *print_nl*("`Tight␣\vbox␣(badness␣`"); *print_int*(*last_badness*); **goto** *common_ending*;
end;
end

This code is used in section 679.

682. 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* $>$ *ignore_depth* **then**
    begin $d \leftarrow$ *width*(*baseline_skip*) $-$ *prev_depth* $-$ *height*(b);
    if $d <$ *line_skip_limit* **then** $p \leftarrow$ *new_param_glue*(*line_skip_code*)
    else begin $p \leftarrow$ *new_skip_param*(*baseline_skip_code*); *width*(*temp_ptr*) $\leftarrow d$; { *temp_ptr* = *glue_ptr*(p) }
      end;
    link(*tail*) $\leftarrow p$; *tail* $\leftarrow p$;
    end;
  link(*tail*) $\leftarrow b$; *tail* $\leftarrow b$; *prev_depth* $\leftarrow$ *depth*(b);
  end;

683. 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.

684. 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 }

685. Each portion of a formula is classified as Ord, Op, Bin, Rel, Ope, Clo, Pun, or Inn, 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 \mathrel 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 Ope }
define $close\_noad = ord\_noad + 5$ { *type* of a noad classified Clo }
define $punct\_noad = ord\_noad + 6$ { *type* of a noad classified Pun }
define $inner\_noad = ord\_noad + 7$ { *type* of a noad classified Inn }
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 }

686. 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

$$math\_type(numerator) = math\_type(denominator) = sub\_mlist.$$

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 }

687. 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*;

688. ⟨ 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*;

689. 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;

690. 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`. 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 *scripts_allowed*(#) ≡ (*type*(#) ≥ *ord_noad*) ∧ (*type*(#) < *left_noad*)

691. 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;

692. 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;

693. 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 693 ⟩ ≡
style_node: *print_style*(*subtype*(*p*));
choice_node: ⟨ Display choice node *p* 698 ⟩;
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* 699 ⟩;
fraction_noad: ⟨ Display fraction noad *p* 700 ⟩;

This code is used in section 183.

694. Here are some simple routines used in the display of noads.

⟨ Declare procedures needed for displaying the elements of mlists 694 ⟩ ≡
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 695 and 697.

This code is used in section 179.

695. 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 *math_type*(*p*) = *empty*, because these are not equivalent (as explained above).

```
⟨Declare procedures needed for displaying the elements of mlists 694⟩ +≡
procedure show_info; forward; { show_node_list(info(temp_ptr)) }
procedure print_subsidiary_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;
```

696. 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 stoopidity 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;
```

697. ⟨Declare procedures needed for displaying the elements of mlists 694⟩ +≡

```
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␣style!")
  endcases;
  end;
```

698. ⟨ Display choice node p 698 ⟩ ≡
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 693.

699. ⟨ Display normal noad p 699 ⟩ ≡
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** *print_esc*("right"); *print_delimiter*(*delimiter*(p));
  end;
end;
if *subtype*(p) $\neq$ *normal* **then**
  if *subtype*(p) $=$ *limits* **then** *print_esc*("limits")
  else *print_esc*("nolimits");
if *type*(p) $<$ *left_noad* **then** *print_subsidiary_data*(*nucleus*(p), ".");
print_subsidiary_data(*supscr*(p), "^"); *print_subsidiary_data*(*subscr*(p), "_");
end

This code is used in section 693.

700. ⟨ Display fraction noad p 700 ⟩ ≡
  begin *print_esc*("fraction,␣thickness␣");
  if *thickness*(p) = *default_code* **then** *print*("=␣default")
  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*(",␣left-delimiter␣"); *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*(",␣right-delimiter␣"); *print_delimiter*(*right_delimiter*(p));
    end;
  print_subsidiary_data(*numerator*(p), "\"); *print_subsidiary_data*(*denominator*(p), "/");
  end

This code is used in section 693.

701. That which can be displayed can also be destroyed.

⟨ Cases of *flush_node_list* that arise in mlists only 701 ⟩ ≡
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 202.

702. 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 *script_script_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;

703. 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

704. 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

705. 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*(#) $\equiv 2 * (\# \ \mathbf{div}\ 2) + cramped$ { cramp the style }
define *sub_style*(#) $\equiv 2 * (\# \ \mathbf{div}\ 4) + script\_style + cramped$ { smaller and cramped }
define *sup_style*(#) $\equiv 2 * (\# \ \mathbf{div}\ 4) + script\_style + (\# \ \mathbf{mod}\ 2)$ { smaller }
define *num_style*(#) $\equiv \# + 2 - 2 * (\# \ \mathbf{div}\ 6)$ { smaller unless already script-script }
define *denom_style*(#) $\equiv 2 * (\# \ \mathbf{div}\ 2) + cramped + 2 - 2 * (\# \ \mathbf{div}\ 6)$ { smaller, cramped }

706. 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* 706⟩ ≡
begin if $cur\_style < script\_style$ **then** $cur\_size \leftarrow text\_size$
else $cur\_size \leftarrow 16 * ((cur\_style - text\_style)\ \mathbf{div}\ 2)$;
$cur\_mu \leftarrow x\_over\_n(math\_quad(cur\_size), 18)$;
end

This code is used in sections 723, 729, 733, 757, 763, and 766.

707. 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 $p \leftarrow new\_rule$; $height(p) \leftarrow t$; $depth(p) \leftarrow 0$; $fraction\_rule \leftarrow p$;
end;

708. 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 $p \leftarrow new\_kern(k)$; $link(p) \leftarrow b$; $q \leftarrow fraction\_rule(t)$; $link(q) \leftarrow p$; $p \leftarrow new\_kern(t)$; $link(p) \leftarrow q$;
$overbar \leftarrow vpack(p, natural)$;
end;

709. 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 noad, 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 712⟩
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 710⟩;
    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) 713⟩
  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;
```

710. 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 710⟩ ≡
  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 711⟩;
    until z < 16;
    end
```

This code is used in section 709.

711. ⟨ 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 711 ⟩ ≡

```
begin y ← x;
if (qo(y) ≥ font_bc[g]) ∧ (qo(y) ≤ font_ec[g]) then
  begin continue: q ← orig_char_info(g)(y);
  if char_exists(q) then
    begin if char_tag(q) = ext_tag then
      begin f ← g; c ← y; goto found;
      end;
    hd ← height_depth(q); u ← char_height(g)(hd) + char_depth(g)(hd);
    if u > w then
      begin f ← g; c ← y; w ← u;
      if u ≥ v then goto found;
      end;
    if char_tag(q) = list_tag then
      begin y ← rem_byte(q); goto continue;
      end;
    end;
  end;
end
```

This code is used in section 710.

712. 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* 712 ⟩ ≡

```
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 ← char_info(f)(c); hd ← height_depth(q); b ← new_null_box;
  width(b) ← char_width(f)(q) + char_italic(f)(q); height(b) ← char_height(f)(hd);
  depth(b) ← char_depth(f)(hd); p ← get_avail; character(p) ← c; font(p) ← f; list_ptr(b) ← p;
  char_box ← b;
  end;
```

See also sections 714 and 715.

This code is used in section 709.

713. 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) 713 ⟩ ≡

```
if char_tag(q) = ext_tag then
  ⟨ Construct an extensible character in a new box b, using recipe rem_byte(q) and font f 716 ⟩
else b ← char_box(f, c)
```

This code is used in section 709.

714. 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* 712 ⟩ +≡
procedure *stack_into_box*(b : *pointer*; f : *internal_font_number*; c : *quarterword*);
  var p: *pointer*; { new node placed into b }
  begin $p \leftarrow$ *char_box*(f, c); *link*(p) ← *list_ptr*(b); *list_ptr*(b) ← p; *height*(b) ← *height*(p);
  end;

715. Another handy subroutine computes the height plus depth of a given character:

⟨ Declare subprocedures for *var_delimiter* 712 ⟩ +≡
function *height_plus_depth*(f : *internal_font_number*; c : *quarterword*): *scaled*;
  var q: *four_quarters*; *hd*: *eight_bits*; { *height_depth* byte }
  begin $q \leftarrow$ *char_info*(f)(c); *hd* ← *height_depth*(q);
  height_plus_depth ← *char_height*(f)(*hd*) + *char_depth*(f)(*hd*);
  end;

716. ⟨ Construct an extensible character in a new box b, using recipe *rem_byte*(q) and font f 716 ⟩ ≡
  begin $b \leftarrow$ *new_null_box*; *type*(b) ← *vlist_node*; $r \leftarrow$ *font_info*[*exten_base*[f] + *rem_byte*(q)].*qqqq*;
  ⟨ Compute the minimum suitable height, w, and the corresponding number of extension steps, n; also set *width*(b) 717 ⟩;
  $c \leftarrow$ *ext_bot*(r);
  if $c \neq$ *min_quarterword* **then** *stack_into_box*(b, f, c);
  $c \leftarrow$ *ext_rep*(r);
  for $m \leftarrow 1$ **to** n **do** *stack_into_box*(b, f, c);
  $c \leftarrow$ *ext_mid*(r);
  if $c \neq$ *min_quarterword* **then**
    begin *stack_into_box*(b, f, c); $c \leftarrow$ *ext_rep*(r);
    for $m \leftarrow 1$ **to** n **do** *stack_into_box*(b, f, c);
    end;
  $c \leftarrow$ *ext_top*(r);
  if $c \neq$ *min_quarterword* **then** *stack_into_box*(b, f, c);
  depth(b) ← w − *height*(b);
  end

This code is used in section 713.

717. 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) 717⟩ ≡

$c \leftarrow ext\_rep(r)$; $u \leftarrow height\_plus\_depth(f,c)$; $w \leftarrow 0$; $q \leftarrow char\_info(f)(c)$;
$width(b) \leftarrow char\_width(f)(q) + char\_italic(f)(q)$;
$c \leftarrow ext\_bot(r)$; **if** $c \neq min\_quarterword$ **then** $w \leftarrow w + height\_plus\_depth(f,c)$;
$c \leftarrow ext\_mid(r)$; **if** $c \neq min\_quarterword$ **then** $w \leftarrow w + height\_plus\_depth(f,c)$;
$c \leftarrow ext\_top(r)$; **if** $c \neq min\_quarterword$ **then** $w \leftarrow w + height\_plus\_depth(f,c)$;
$n \leftarrow 0$;
if $u > 0$ **then**
  while $w < v$ **do**
    begin $w \leftarrow w + u$; $incr(n)$;
    if $ext\_mid(r) \neq min\_quarterword$ **then** $w \leftarrow w + u$;
    end

This code is used in section 716.

718. 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 $(width(b) \neq w) \land (list\_ptr(b) \neq null)$ **then**
    begin if $type(b) = vlist\_node$ **then** $b \leftarrow hpack(b, natural)$;
    $p \leftarrow list\_ptr(b)$;
    if $(is\_char\_node(p)) \land (link(p) = null)$ **then**
      begin $f \leftarrow font(p)$;
      if $is\_wchar\_node(p)$ **then** $v \leftarrow cfont\_width[f]$
      else $v \leftarrow char\_width(f)(char\_info(f)(character(p)))$;
      if $v \neq width(b)$ **then** $link(p) \leftarrow new\_kern(width(b) - v)$;
      end;
    $free\_node(b, box\_node\_size)$; $b \leftarrow new\_glue(ss\_glue)$; $link(b) \leftarrow p$;
    while $link(p) \neq null$ **do** $p \leftarrow link(p)$;
    $link(p) \leftarrow new\_glue(ss\_glue)$; $rebox \leftarrow hpack(b, w, exactly)$;
    end
  else begin $width(b) \leftarrow w$; $rebox \leftarrow b$;
    end;
  end;

719. 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;
```

720. 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;
```

721. 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;
```

722. 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? }

723. 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** $cur\_mlist \gets new\_noad$; $mem[nucleus(cur\_mlist)] \gets mem[p]$;
    end;
  sub_box: **begin** $q \gets info(p)$; **goto** *found*;
    end;
  sub_mlist: $cur\_mlist \gets info(p)$;
  othercases begin $q \gets new\_null\_box$; **goto** *found*;
    end
  endcases;
  $save\_style \gets cur\_style$; $cur\_style \gets s$; $mlist\_penalties \gets false$;
  mlist_to_hlist; $q \gets link(temp\_head)$; { recursive call }
  $cur\_style \gets save\_style$; { restore the style }
  ⟨Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 706⟩;
found: **if** $is\_char\_node(q) \lor (q = null)$ **then** $x \gets hpack(q, natural)$
  else if $(link(q) = null) \land (type(q) \le vlist\_node) \land (shift\_amount(q) = 0)$ **then** $x \gets q$
      { it's already clean }
    else $x \gets hpack(q, natural)$;
  ⟨Simplify a trivial box 724⟩;
  $clean\_box \gets x$;
  end;

724. Here we save memory space in a common case.

⟨ Simplify a trivial box 724 ⟩ ≡
```
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
```
This code is used in section 723.

725. 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 726 ⟩
  else begin if (qo(cur_c) ≥ font_bc[cur_f]) ∧ (qo(cur_c) ≤ font_ec[cur_f]) then
      cur_i ← orig_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;
      end;
    end;
  end;
```

726. ⟨ Complain about an undefined family and set *cur_i* null 726 ⟩ ≡
```
  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␣math␣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 725.

727. 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 }
```

728. 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 }

729. 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 737 ⟩
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* 706 ⟩;
  while $q \neq$ *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 730 ⟩;
  ⟨ Convert a final *bin_noad* to an *ord_noad* 732 ⟩;
  ⟨ Make a second pass over the mlist, removing all noads and inserting the proper spacing and penalties 763 ⟩;
  end;

730. 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 730⟩ ≡
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 731⟩;
check_dimensions: $z \gets$ *hpack*(*new_hlist*(q), *natural*);
if *height*(z) > *max_h* **then** *max_h* ← *height*(z);
if *depth*(z) > *max_d* **then** *max_d* ← *depth*(z);
free_node(z, *box_node_size*);
done_with_noad: $r \gets q$; *r_type* ← *type*(r);
done_with_node: $q \gets$ *link*(q);
end

This code is used in section 729.

731. 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 731⟩ ≡
reswitch: *delta* ← 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) ← *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* 732⟩;
if *type*(q) = *right_noad* **then goto** *done_with_noad*;
end;
⟨Cases for noads that can follow a *bin_noad* 736⟩
⟨Cases for nodes that can appear in an mlist, after which we **goto** *done_with_node* 733⟩
othercases *confusion*("mlist1")
endcases;
⟨Convert *nucleus*(q) to an hlist and attach the sub/superscripts 757⟩

This code is used in section 730.

732. ⟨Convert a final *bin_noad* to an *ord_noad* 732⟩ ≡
if *r_type* = *bin_noad* **then** *type*(r) ← *ord_noad*

This code is used in sections 729 and 731.

733. ⟨ Cases for nodes that can appear in an mlist, after which we **goto** *done_with_node* 733 ⟩ ≡
style_node: **begin** *cur_style* ← *subtype*(*q*);
  ⟨ Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 706 ⟩;
  goto *done_with_node*;
  end;
choice_node: ⟨ Change this node to a style node followed by the correct choice, then **goto** *done_with_node* 734 ⟩;
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 735 ⟩;
  goto *done_with_node*;
  end;
kern_node: **begin** *math_kern*(*q*, *cur_mu*); **goto** *done_with_node*;
  end;

This code is used in section 731.

734. **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* 734 ⟩ ≡
  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 733.

735. 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 735⟩ ≡
  if $subtype(q) = mu\_glue$ **then**
    begin $x \leftarrow glue\_ptr(q)$; $y \leftarrow math\_glue(x, cur\_mu)$; $delete\_glue\_ref(x)$; $glue\_ptr(q) \leftarrow y$;
    $subtype(q) \leftarrow normal$;
    end
  else if $(cur\_size \neq text\_size) \land (subtype(q) = cond\_math\_glue)$ **then**
      begin $p \leftarrow link(q)$;
      if $p \neq null$ **then**
        if $(type(p) = glue\_node) \lor (type(p) = kern\_node)$ **then**
          begin $link(q) \leftarrow link(p)$; $link(p) \leftarrow null$; $flush\_node\_list(p)$;
          end;
      end

This code is used in section 733.

736. ⟨Cases for noads that can follow a *bin_noad* 736⟩ ≡
left_noad: **goto** *done_with_noad*;
fraction_noad: **begin** $make\_fraction(q)$; **goto** *check_dimensions*;
  end;
op_noad: **begin** $delta \leftarrow make\_op(q)$;
  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 731.

737. 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 737⟩ ≡
procedure $make\_over(q : pointer)$;
  begin $info(nucleus(q)) \leftarrow overbar(clean\_box(nucleus(q), cramped\_style(cur\_style)),$
    $3 * default\_rule\_thickness, default\_rule\_thickness)$; $math\_type(nucleus(q)) \leftarrow sub\_box$;
  end;

See also sections 738, 739, 740, 741, 746, 752, 755, 759, and 765.

This code is used in section 729.

738. ⟨Declare math construction procedures 737⟩ +≡
procedure *make_under*(*q* : *pointer*);
  var *p*, *x*, *y*: *pointer*; { temporary registers for box construction }
    delta: *scaled*; { overall height plus depth }
  begin $x \leftarrow$ *clean_box*(*nucleus*(*q*), *cur_style*); $p \leftarrow$ *new_kern*($3 *$ *default_rule_thickness*); *link*(*x*) $\leftarrow p$;
  link(*p*) $\leftarrow$ *fraction_rule*(*default_rule_thickness*); $y \leftarrow$ *vpack*(*x*, *natural*);
  delta $\leftarrow$ *height*(*y*) + *depth*(*y*) + *default_rule_thickness*; *height*(*y*) $\leftarrow$ *height*(*x*);
  depth(*y*) $\leftarrow$ *delta* − *height*(*y*); *info*(*nucleus*(*q*)) $\leftarrow y$; *math_type*(*nucleus*(*q*)) $\leftarrow$ *sub_box*;
  end;

739. ⟨Declare math construction procedures 737⟩ +≡
procedure *make_vcenter*(*q* : *pointer*);
  var *v*: *pointer*; { the box that should be centered vertically }
    delta: *scaled*; { its height plus depth }
  begin $v \leftarrow$ *info*(*nucleus*(*q*));
  if *type*(*v*) $\neq$ *vlist_node* **then** *confusion*("vcenter");
  delta $\leftarrow$ *height*(*v*) + *depth*(*v*); *height*(*v*) $\leftarrow$ *axis_height*(*cur_size*) + *half*(*delta*);
  depth(*v*) $\leftarrow$ *delta* − *height*(*v*);
  end;

740. 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.

⟨Declare math construction procedures 737⟩ +≡
procedure *make_radical*(*q* : *pointer*);
  var *x*, *y*: *pointer*; { temporary registers for box construction }
    delta, *clr*: *scaled*; { dimensions involved in the calculation }
  begin $x \leftarrow$ *clean_box*(*nucleus*(*q*), *cramped_style*(*cur_style*));
  if *cur_style* < *text_style* **then** { display style }
    clr $\leftarrow$ *default_rule_thickness* + (*abs*(*math_x_height*(*cur_size*)) **div** 4)
  else begin *clr* $\leftarrow$ *default_rule_thickness*; *clr* $\leftarrow$ *clr* + (*abs*(*clr*) **div** 4);
    end;
  $y \leftarrow$ *var_delimiter*(*left_delimiter*(*q*), *cur_size*, *height*(*x*) + *depth*(*x*) + *clr* + *default_rule_thickness*);
  delta $\leftarrow$ *depth*(*y*) − (*height*(*x*) + *depth*(*x*) + *clr*);
  if *delta* > 0 **then** *clr* $\leftarrow$ *clr* + *half*(*delta*); { increase the actual clearance }
  shift_amount(*y*) $\leftarrow$ −(*height*(*x*) + *clr*); *link*(*y*) $\leftarrow$ *overbar*(*x*, *clr*, *height*(*y*));
  info(*nucleus*(*q*)) $\leftarrow$ *hpack*(*y*, *natural*); *math_type*(*nucleus*(*q*)) $\leftarrow$ *sub_box*;
  end;

741. 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 737 ⟩ +≡
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 $i \gets$ *cur_i*; $c \gets$ *cur_c*; $f \gets$ *cur_f*;
    ⟨ Compute the amount of skew 744 ⟩;
    $x \gets$ *clean_box*(*nucleus*(q), *cramped_style*(*cur_style*)); $w \gets$ *width*(x); $h \gets$ *height*(x);
    ⟨ Switch to a larger accent if available and appropriate 743 ⟩;
    if $h <$ *x_height*(f) **then** *delta* $\gets h$ **else** *delta* $\gets$ *x_height*(f);
    if (*math_type*(*supscr*(q)) $\neq$ *empty*) $\lor$ (*math_type*(*subscr*(q)) $\neq$ *empty*) **then**
      if *math_type*(*nucleus*(q)) = *math_char* **then** ⟨ Swap the subscript and superscript into box x 745 ⟩;
    $y \gets$ *char_box*(f, c); *shift_amount*(y) $\gets s +$ *half*($w -$ *width*(y)); *width*(y) $\gets 0$; $p \gets$ *new_kern*($-$*delta*);
    link(p) $\gets x$; *link*(y) $\gets p$; $y \gets$ *vpack*(y, *natural*); *width*(y) $\gets$ *width*(x);
    if *height*(y) $< h$ **then** ⟨ Make the height of box y equal to h 742 ⟩;
    info(*nucleus*(q)) $\gets y$; *math_type*(*nucleus*(q)) $\gets$ *sub_box*;
    end;
  end;

742. ⟨ Make the height of box y equal to h 742 ⟩ ≡
  begin $p \gets$ *new_kern*($h -$ *height*(y)); *link*(p) $\gets$ *list_ptr*(y); *list_ptr*(y) $\gets p$; *height*(y) $\gets h$;
  end

This code is used in section 741.

743. ⟨ Switch to a larger accent if available and appropriate 743 ⟩ ≡
  loop begin if *char_tag*(i) $\neq$ *list_tag* **then goto** *done*;
    $y \gets$ *rem_byte*(i); $i \gets$ *orig_char_info*(f)(y);
    if $\lnot$*char_exists*(i) **then goto** *done*;
    if *char_width*(f)(i) $> w$ **then goto** *done*;
    $c \gets y$;
    end;
done:

This code is used in section 741.

744. ⟨ Compute the amount of skew 744 ⟩ ≡

```
s ← 0;
if math_type(nucleus(q)) = math_char then
  begin fetch(nucleus(q));
  if char_tag(cur_i) = lig_tag then
    begin a ← lig_kern_start(cur_f)(cur_i); 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 qo(next_char(cur_i)) = skew_char[cur_f] then
        begin if op_byte(cur_i) ≥ kern_flag then
          if skip_byte(cur_i) ≤ stop_flag then s ← char_kern(cur_f)(cur_i);
        goto done1;
        end;
      if skip_byte(cur_i) ≥ stop_flag then goto done1;
      a ← a + qo(skip_byte(cur_i)) + 1; cur_i ← font_info[a].qqqq;
      end;
    end;
  end;
done1:
```

This code is used in section 741.

745. ⟨ Swap the subscript and superscript into box x 745 ⟩ ≡

```
begin flush_node_list(x); x ← new_noad; mem[nucleus(x)] ← mem[nucleus(q)];
mem[supscr(x)] ← mem[supscr(q)]; mem[subscr(x)] ← mem[subscr(q)];
mem[supscr(q)].hh ← empty_field; mem[subscr(q)].hh ← empty_field;
math_type(nucleus(q)) ← sub_mlist; info(nucleus(q)) ← x; x ← clean_box(nucleus(q), cur_style);
delta ← delta + height(x) − h; h ← height(x);
end
```

This code is used in section 741.

746. 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 737 ⟩ +≡

```
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 747 ⟩;
  if thickness(q) = 0 then ⟨ Adjust shift_up and shift_down for the case of no fraction line 748 ⟩
  else ⟨ Adjust shift_up and shift_down for the case of a fraction line 749 ⟩;
  ⟨ Construct a vlist box for the fraction, according to shift_up and shift_down 750 ⟩;
  ⟨ Put the fraction into a box with its delimiters, and make new_hlist(q) point to it 751 ⟩;
  end;
```

747. ⟨ 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 747 ⟩ ≡

$x \leftarrow clean\_box(numerator(q), num\_style(cur\_style));$
$z \leftarrow clean\_box(denominator(q), denom\_style(cur\_style));$
if $width(x) < width(z)$ **then** $x \leftarrow rebox(x, width(z))$
else $z \leftarrow rebox(z, width(x));$
if $cur\_style < text\_style$ **then** { display style }
  begin $shift\_up \leftarrow num1(cur\_size);$ $shift\_down \leftarrow denom1(cur\_size);$
  end
else begin $shift\_down \leftarrow denom2(cur\_size);$
  if $thickness(q) \neq 0$ **then** $shift\_up \leftarrow num2(cur\_size)$
  else $shift\_up \leftarrow num3(cur\_size);$
  end

This code is used in section 746.

748. 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 $2\,delta$.

⟨ Adjust *shift_up* and *shift_down* for the case of no fraction line 748 ⟩ ≡

begin if $cur\_style < text\_style$ **then** $clr \leftarrow 7 * default\_rule\_thickness$
else $clr \leftarrow 3 * default\_rule\_thickness;$
$delta \leftarrow half(clr - ((shift\_up - depth(x)) - (height(z) - shift\_down)));$
if $delta > 0$ **then**
  begin $shift\_up \leftarrow shift\_up + delta;$ $shift\_down \leftarrow shift\_down + delta;$
  end;
end

This code is used in section 746.

749. 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 749 ⟩ ≡

begin if $cur\_style < text\_style$ **then** $clr \leftarrow 3 * thickness(q)$
else $clr \leftarrow thickness(q);$
$delta \leftarrow half(thickness(q));$ $delta1 \leftarrow clr - ((shift\_up - depth(x)) - (axis\_height(cur\_size) + delta));$
$delta2 \leftarrow clr - ((axis\_height(cur\_size) - delta) - (height(z) - shift\_down));$
if $delta1 > 0$ **then** $shift\_up \leftarrow shift\_up + delta1;$
if $delta2 > 0$ **then** $shift\_down \leftarrow shift\_down + delta2;$
end

This code is used in section 746.

750. ⟨ Construct a vlist box for the fraction, according to *shift_up* and *shift_down* 750 ⟩ ≡

$v \leftarrow new\_null\_box;$ $type(v) \leftarrow vlist\_node;$ $height(v) \leftarrow shift\_up + height(x);$
$depth(v) \leftarrow depth(z) + shift\_down;$ $width(v) \leftarrow width(x);$ { this also equals $width(z)$ }
if $thickness(q) = 0$ **then**
  begin $p \leftarrow new\_kern((shift\_up - depth(x)) - (height(z) - shift\_down));$ $link(p) \leftarrow z;$
  end
else begin $y \leftarrow fraction\_rule(thickness(q));$
  $p \leftarrow new\_kern((axis\_height(cur\_size) - delta) - (height(z) - shift\_down));$
  $link(y) \leftarrow p;$ $link(p) \leftarrow z;$
  $p \leftarrow new\_kern((shift\_up - depth(x)) - (axis\_height(cur\_size) + delta));$ $link(p) \leftarrow y;$
  end;
$link(x) \leftarrow p;$ $list\_ptr(v) \leftarrow x$

This code is used in section 746.

751. ⟨Put the fraction into a box with its delimiters, and make *new_hlist*(q) point to it 751⟩ ≡
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 746.

752. 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 737⟩ +≡
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 ← *orig_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* 753⟩;
  make_op ← *delta*;
  end;

753. 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* 753⟩ ≡
begin $x \leftarrow$ *clean_box*(*supscr*(q), *sup_style*(*cur_style*)); $y \leftarrow$ *clean_box*(*nucleus*(q), *cur_style*);
$z \leftarrow$ *clean_box*(*subscr*(q), *sub_style*(*cur_style*)); $v \leftarrow$ *new_null_box*; *type*(v) $\leftarrow$ *vlist_node*;
width(v) $\leftarrow$ *width*(y);
if *width*(x) $>$ *width*(v) **then** *width*(v) $\leftarrow$ *width*(x);
if *width*(z) $>$ *width*(v) **then** *width*(v) $\leftarrow$ *width*(z);
$x \leftarrow$ *rebox*(x, *width*(v)); $y \leftarrow$ *rebox*(y, *width*(v)); $z \leftarrow$ *rebox*(z, *width*(v));
shift_amount(x) $\leftarrow$ *half*(*delta*); *shift_amount*(z) $\leftarrow -$*shift_amount*(x); *height*(v) $\leftarrow$ *height*(y);
depth(v) $\leftarrow$ *depth*(y);
⟨Attach the limits to y and adjust *height*(v), *depth*(v) to account for their presence 754⟩;
new_hlist(q) $\leftarrow v$;
end

This code is used in section 752.

754. 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 754⟩ ≡
if *math_type*(*supscr*(q)) $=$ *empty* **then**
begin *free_node*(x, *box_node_size*); *list_ptr*(v) $\leftarrow y$;
end
else begin *shift_up* $\leftarrow$ *big_op_spacing3* $-$ *depth*(x);
if *shift_up* $<$ *big_op_spacing1* **then** *shift_up* $\leftarrow$ *big_op_spacing1*;
$p \leftarrow$ *new_kern*(*shift_up*); *link*(p) $\leftarrow y$; *link*(x) $\leftarrow p$;
$p \leftarrow$ *new_kern*(*big_op_spacing5*); *link*(p) $\leftarrow x$; *list_ptr*(v) $\leftarrow p$;
height(v) $\leftarrow$ *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* $\leftarrow$ *big_op_spacing4* $-$ *height*(z);
if *shift_down* $<$ *big_op_spacing2* **then** *shift_down* $\leftarrow$ *big_op_spacing2*;
$p \leftarrow$ *new_kern*(*shift_down*); *link*(y) $\leftarrow p$; *link*(p) $\leftarrow z$;
$p \leftarrow$ *new_kern*(*big_op_spacing5*); *link*(z) $\leftarrow p$;
depth(v) $\leftarrow$ *depth*(v) $+$ *big_op_spacing5* $+$ *height*(z) $+$ *depth*(z) $+$ *shift_down*;
end

This code is used in section 753.

755. 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 737⟩ +≡
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 756⟩;
                    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;
exit: end;
```

756. 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* 756⟩ ≡

```
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 755.

757. 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 757⟩ ≡
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 758⟩;
empty: $p \leftarrow$ *null*;
sub_box: $p \leftarrow$ *info*(*nucleus*(q));
sub_mlist: **begin** *cur_mlist* $\leftarrow$ *info*(*nucleus*(q)); *save_style* $\leftarrow$ *cur_style*; *mlist_penalties* $\leftarrow$ *false*;
mlist_to_hlist; { recursive call }
cur_style $\leftarrow$ *save_style*; ⟨Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 706⟩;
$p \leftarrow$ *hpack*(*link*(*temp_head*), *natural*);
end;
othercases *confusion*("mlist2")
endcases;
new_hlist(q) $\leftarrow p$;
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 731.

758. ⟨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 758⟩ ≡
begin *fetch*(*nucleus*(q));
if *char_exists*(*cur_i*) **then**
begin *delta* $\leftarrow$ *char_italic*(*cur_f*)(*cur_i*); $p \leftarrow$ *new_character*(*cur_f*, *qo*(*cur_c*));
if (*math_type*(*nucleus*(q)) = *math_text_char*) ∧ (*space*(*cur_f*) $\neq 0$) **then** *delta* $\leftarrow 0$;
{ no italic correction in mid-word of text font }
if (*math_type*(*subscr*(q)) = *empty*) ∧ (*delta* $\neq 0$) **then**
begin *link*(p) $\leftarrow$ *new_kern*(*delta*); *delta* $\leftarrow 0$;
end;
end
else $p \leftarrow$ *null*;
end

This code is used in section 757.

759. The purpose of *make_scripts*$(q, delta)$ is to attach the subscript and/or superscript of noad q to the list that starts at *new_hlist*(q), given that 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 737 ⟩ +≡
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 760 ⟩
  else begin ⟨ Construct a superscript box x 761 ⟩;
    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 762 ⟩;
    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;
```

760. 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 760 ⟩ ≡
  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 759.

761. The bottom of a superscript should never descend below the baseline plus one-fourth of the x-height.

⟨ Construct a superscript box x 761 ⟩ ≡
begin $x \gets$ *clean_box*(*supscr*(q), *sup_style*(*cur_style*)); *width*(x) $\gets$ *width*(x) + *script_space*;
if *odd*(*cur_style*) **then** *clr* $\gets$ *sup3*(*cur_size*)
else if *cur_style* < *text_style* **then** *clr* $\gets$ *sup1*(*cur_size*)
else *clr* $\gets$ *sup2*(*cur_size*);
if *shift_up* < *clr* **then** *shift_up* $\gets$ *clr*;
clr $\gets$ *depth*(x) + (*abs*(*math_x_height*(*cur_size*)) **div** 4);
if *shift_up* < *clr* **then** *shift_up* $\gets$ *clr*;
end

This code is used in section 759.

762. When both subscript and superscript are present, the subscript must be separated from the superscript by at least four times *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.

⟨ Construct a sub/superscript combination box x, with the superscript offset by *delta* 762 ⟩ ≡
begin $y \gets$ *clean_box*(*subscr*(q), *sub_style*(*cur_style*)); *width*(y) $\gets$ *width*(y) + *script_space*;
if *shift_down* < *sub2*(*cur_size*) **then** *shift_down* $\gets$ *sub2*(*cur_size*);
clr $\gets 4 *$ *default_rule_thickness* − ((*shift_up* − *depth*(x)) − (*height*(y) − *shift_down*));
if *clr* > 0 **then**
begin *shift_down* $\gets$ *shift_down* + *clr*;
clr $\gets$ (*abs*(*math_x_height*(*cur_size*) ∗ 4) **div** 5) − (*shift_up* − *depth*(x));
if *clr* > 0 **then**
begin *shift_up* $\gets$ *shift_up* + *clr*; *shift_down* $\gets$ *shift_down* − *clr*;
end;
end;
shift_amount(x) $\gets$ *delta*; { superscript is *delta* to the right of the subscript }
$p \gets$ *new_kern*((*shift_up* − *depth*(x)) − (*height*(y) − *shift_down*)); *link*(x) $\gets p$; *link*(p) $\gets y$;
$x \gets$ *vpack*(x, *natural*); *shift_amount*(x) $\gets$ *shift_down*;
end

This code is used in section 759.

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

⟨ Make a second pass over the mlist, removing all noads and inserting the proper spacing and penalties 763 ⟩ ≡
$p \gets$ *temp_head*; *link*(p) $\gets$ *null*; $q \gets$ *mlist*; *r_type* $\gets 0$; *cur_style* $\gets$ *style*;
⟨ Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 706 ⟩;
while $q \neq$ *null* **do**
begin ⟨ If node q is a style node, change the style and **goto** *delete_q*; otherwise if it is not a noad, put it into the hlist, advance q, and **goto** *done*; otherwise set s to the size of noad q, set t to the associated type (*ord_noad* .. *inner_noad*), and set *pen* to the associated penalty 764 ⟩;
⟨ Append inter-element spacing based on *r_type* and t 769 ⟩;
⟨ Append any *new_hlist* entries for q, and any appropriate penalties 770 ⟩;
r_type $\gets t$;
delete_q: $r \gets q$; $q \gets$ *link*(q); *free_node*(r, s);
done: **end**

This code is used in section 729.

764. 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 764⟩ ≡

$t \leftarrow ord\_noad$; $s \leftarrow noad\_size$; $pen \leftarrow inf\_penalty$;
case *type*(q) **of**
op_noad, *open_noad*, *close_noad*, *punct_noad*, *inner_noad*: $t \leftarrow type(q)$;
bin_noad: **begin** $t \leftarrow bin\_noad$; $pen \leftarrow bin\_op\_penalty$;
  end;
rel_noad: **begin** $t \leftarrow rel\_noad$; $pen \leftarrow rel\_penalty$;
  end;
ord_noad, *vcenter_noad*, *over_noad*, *under_noad*: *do_nothing*;
radical_noad: $s \leftarrow radical\_noad\_size$;
accent_noad: $s \leftarrow accent\_noad\_size$;
fraction_noad: **begin** $t \leftarrow inner\_noad$; $s \leftarrow fraction\_noad\_size$;
  end;
left_noad, *right_noad*: $t \leftarrow make\_left\_right(q, style, max\_d, max\_h)$;
style_node: ⟨Change the current style and **goto** *delete_q* 766⟩;
whatsit_node, *penalty_node*, *rule_node*, *disc_node*, *adjust_node*, *ins_node*, *mark_node*, *glue_node*, *kern_node*:
  begin $link(p) \leftarrow q$; $p \leftarrow q$; $q \leftarrow link(q)$; $link(p) \leftarrow null$; **goto** *done*;
  end;
othercases *confusion*("mlist3")
endcases

This code is used in section 763.

765. 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 737⟩ +≡

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 if $style < script\_style$ **then** $cur\_size \leftarrow text\_size$
else $cur\_size \leftarrow 16 * ((style - text\_style)$ **div** $2)$;
$delta2 \leftarrow max\_d + axis\_height(cur\_size)$; $delta1 \leftarrow max\_h + max\_d - delta2$;
if $delta2 > delta1$ **then** $delta1 \leftarrow delta2$; { *delta1* is max distance from axis }
$delta \leftarrow (delta1$ **div** $500) * delimiter\_factor$; $delta2 \leftarrow delta1 + delta1 - delimiter\_shortfall$;
if $delta < delta2$ **then** $delta \leftarrow delta2$;
$new\_hlist(q) \leftarrow var\_delimiter(delimiter(q), cur\_size, delta)$;
$make\_left\_right \leftarrow type(q) - (left\_noad - open\_noad)$; { *open_noad* or *close_noad* }
end;

766. ⟨Change the current style and **goto** *delete_q* 766⟩ ≡

begin $cur\_style \leftarrow subtype(q)$; $s \leftarrow style\_node\_size$;
⟨Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 706⟩;
goto *delete_q*;
end

This code is used in section 764.

767. The inter-element spacing in math formulas depends on a 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 }

768. ⟨ Compute the magic offset 768 ⟩ ≡
magic_offset ← *str_start*[*math_spacing*] − 9 ∗ *ord_noad*

This code is used in section 1340.

769. ⟨ Append inter-element spacing based on *r_type* and *t* 769 ⟩ ≡
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 763.

770. 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.

⟨Append any *new_hlist* entries for q, and any appropriate penalties 770⟩ ≡

```
if new_hlist(q) ≠ null then
  begin link(p) ← new_hlist(q);
  repeat p ← link(p);
  until link(p) = null;
  end;
if penalties then
  if link(q) ≠ null then
    if pen < inf_penalty then
      begin r_type ← type(link(q));
      if r_type ≠ penalty_node then
        if r_type ≠ rel_noad then
          begin z ← new_penalty(pen); link(p) ← z; p ← z;
          end;
      end
```

This code is used in section 763.

771. 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:

`\glue 2pt plus 3pt`	(the tabskip preceding column 1)
`\alignrecord, width` $-\infty$	(preamble info for column 1)
`\glue 2pt plus 3pt`	(the tabskip between columns 1 and 2)
`\alignrecord, width` $-\infty$	(preamble info for column 2)
`\glue 1pt plus 1fil`	(the tabskip between columns 2 and 3)
`\alignrecord, width` $-\infty$	(preamble info for column 3)
`\glue 1pt plus 1fil`	(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.

772. 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*(#) $\equiv$ *mem*[# + *height_offset*].*int* { pointer to $\langle u_j \rangle$ token list }
define *v_part*(#) $\equiv$ *mem*[# + *depth_offset*].*int* { pointer to $\langle v_j \rangle$ token list }
define *extra_info*(#) $\equiv$ *info*(# + *list_offset*) { info to remember during template }

773. 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* $\equiv$ *link*(*align_head*) { the current preamble list }
define *align_stack_node_size* = 5 { 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 }

774. 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*;

775. 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*; *align_ptr* ← *p*; *cur_head* ← *get_avail*;
end;

procedure *pop_alignment*;
var *p*: *pointer*; { the top alignment stack node }
begin *free_avail*(*cur_head*); *p* ← *align_ptr*; *cur_tail* ← *link*(*p* + 4); *cur_head* ← *info*(*p* + 4);
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;

776. 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*.

777. 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* 785⟩
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 779⟩;
  push_nest; { enter a new semantic level }
  ⟨Change current mode to −*vmode* for `\halign`, −*hmode* for `\valign` 778⟩;
  scan_spec(*align_group*, *false*);
  ⟨Scan the preamble and record it in the *preamble* list 780⟩;
  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;

778. 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` 778⟩ ≡
  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 777.

779. When `\halign` is used as a displayed formula, there should be no other pieces of mlists present.

⟨Check for improper alignment in displayed math 779⟩ ≡
  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 777.

780. ⟨ Scan the preamble and record it in the *preamble* list 780 ⟩ ≡
preamble ← *null*; *cur_align* ← *align_head*; *cur_loop* ← *null*; *scanner_status* ← *aligning*;
warning_index ← *save_cs_ptr*; *align_state* ← −1000000; { at this point, *cur_cmd* = *left_brace* }
loop begin ⟨ Append the current tabskip glue to the preamble list 781 ⟩;
if *cur_cmd* = *car_ret* **then goto** *done*; { `\cr` ends the preamble }
⟨ 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 782 ⟩;
end;
done: *scanner_status* ← *normal*

This code is used in section 777.

781. ⟨ Append the current tabskip glue to the preamble list 781 ⟩ ≡
link(*cur_align*) ← *new_param_glue*(*tab_skip_code*); *cur_align* ← *link*(*cur_align*)

This code is used in section 780.

782. ⟨ 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 782 ⟩ ≡
⟨ Scan the template $\langle u_j\rangle$, putting the resulting token list in *hold_head* 786 ⟩;
link(*cur_align*) ← *new_null_box*; *cur_align* ← *link*(*cur_align*); { a new alignrecord }
info(*cur_align*) ← *end_span*; *width*(*cur_align*) ← *null_flag*; *u_part*(*cur_align*) ← *link*(*hold_head*);
⟨ Scan the template $\langle v_j\rangle$, putting the resulting token list in *hold_head* 787 ⟩;
v_part(*cur_align*) ← *link*(*hold_head*)

This code is used in section 780.

783. 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 '`\crcr`'.

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 `\crcr` from `\cr` }
define *end_template_token* ≡ *cs_token_flag* + *frozen_end_template*

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("`span`", *tab_mark*, *span_code*);
primitive("`cr`", *car_ret*, *cr_code*); *text*(*frozen_cr*) ← "`cr`"; *eqtb*[*frozen_cr*] ← *eqtb*[*cur_val*];
primitive("`crcr`", *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*;

784. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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*("`crcr`");

785. 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* 785⟩ ≡

```
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 777.

786. Spaces are eliminated from the beginning of a template.

⟨Scan the template ⟨u_j⟩, putting the resulting token list in *hold_head* 786⟩ ≡

```
  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 782.

787. ⟨Scan the template $\langle v_j\rangle$, putting the resulting token list in *hold_head* 787⟩ ≡
$p \leftarrow$ *hold_head*; *link*(p) $\leftarrow$ *null*;
loop begin *continue*: *get_preamble_token*;
  if (*cur_cmd* $\le$ *car_ret*) $\wedge$ (*cur_cmd* $\ge$ *tab_mark*) $\wedge$ (*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) $\leftarrow$ *get_avail*; $p \leftarrow$ *link*(p); *info*(p) $\leftarrow$ *cur_tok*;
  end;
done2: *link*(p) $\leftarrow$ *get_avail*; $p \leftarrow$ *link*(p); *info*(p) $\leftarrow$ *end_template_token* { put `\endtemplate` at the end }

This code is used in section 782.

788. 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* 788⟩ ≡
procedure *align_peek*;
label *restart*;
begin *restart*: *align_state* $\leftarrow 1000000$; ⟨Get the next non-blank non-call token 409⟩;
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*) $\wedge$ (*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 803.

789. 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* 790⟩
procedure *init_row*;
begin *push_nest*; *mode* $\leftarrow$ ($-$*hmode* $-$ *vmode*) $-$ *mode*;
if *mode* $= -$*hmode* **then** *space_factor* $\leftarrow 0$ **else** *prev_depth* $\leftarrow 0$;
tail_append(*new_glue*(*glue_ptr*(*preamble*))); *subtype*(*tail*) $\leftarrow$ *tab_skip_code* $+ 1$;
cur_align $\leftarrow$ *link*(*preamble*); *cur_tail* $\leftarrow$ *cur_head*; *init_span*(*cur_align*);
end;

790. 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* 790 ⟩ ≡
procedure *init_span*(*p* : *pointer*);
  begin *push_nest*;
  if *mode* = −*hmode* **then** *space_factor* ← 1000
  else begin *prev_depth* ← *ignore_depth*; *normal_paragraph*;
    end;
  cur_span ← *p*;
  end;

This code is used in section 789.

791. 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;

792. 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* 792 ⟩ ≡
  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 342.

793. 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 793 ⟩ ≡
  info(*omit_template*) ← *end_template_token*; { *link*(*omit_template*) = *null* }

See also sections 800, 823, 984, and 991.

This code is used in section 164.

794. 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 795 ⟩;
  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 799 ⟩;
    ⟨ Copy the tabskip glue between columns 798 ⟩;
    if *extra_info*(*cur_align*) ≥ *cr_code* **then**
      begin *fin_col* ← *true*; **return**;
      end;
    init_span(*p*);
    end;
  align_state ← 1000000; ⟨ Get the next non-blank non-call token 409 ⟩;
  cur_align ← *p*; *init_col*; *fin_col* ← *false*;
exit: **end**;

795. ⟨ If the preamble list has been traversed, check that the row has ended 795 ⟩ ≡
  if (*p* = *null*) ∧ (*extra_info*(*cur_align*) < *cr_code*) **then**
    if *cur_loop* ≠ *null* **then** ⟨ Lengthen the preamble periodically 796 ⟩
    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 794.

796. ⟨ Lengthen the preamble periodically 796 ⟩ ≡
  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* 797 ⟩;
  cur_loop ← *link*(*cur_loop*); *link*(*p*) ← *new_glue*(*glue_ptr*(*cur_loop*));
  end

This code is used in section 795.

797. ⟨Copy the templates from node *cur_loop* into node p 797⟩ ≡
$q \leftarrow hold\_head$; $r \leftarrow u\_part(cur\_loop)$;
while $r \neq null$ **do**
begin $link(q) \leftarrow get\_avail$; $q \leftarrow link(q)$; $info(q) \leftarrow info(r)$; $r \leftarrow link(r)$;
end;
$link(q) \leftarrow null$; $u\_part(p) \leftarrow link(hold\_head)$; $q \leftarrow hold\_head$; $r \leftarrow v\_part(cur\_loop)$;
while $r \neq null$ **do**
begin $link(q) \leftarrow get\_avail$; $q \leftarrow link(q)$; $info(q) \leftarrow info(r)$; $r \leftarrow link(r)$;
end;
$link(q) \leftarrow null$; $v\_part(p) \leftarrow link(hold\_head)$

This code is used in section 796.

798. ⟨Copy the tabskip glue between columns 798⟩ ≡
$tail\_append(new\_glue(glue\_ptr(link(cur\_align))))$; $subtype(tail) \leftarrow tab\_skip\_code + 1$

This code is used in section 794.

799. ⟨Package an unset box for the current column and record its width 799⟩ ≡
begin if $mode = -hmode$ **then**
begin $adjust\_tail \leftarrow cur\_tail$; $u \leftarrow hpack(link(head), natural)$; $w \leftarrow width(u)$; $cur\_tail \leftarrow adjust\_tail$;
$adjust\_tail \leftarrow null$;
end
else begin $u \leftarrow vpackage(link(head), natural, 0)$; $w \leftarrow height(u)$;
end;
$n \leftarrow min\_quarterword$; { this represents a span count of 1 }
if $cur\_span \neq cur\_align$ **then** ⟨Update width entry for spanned columns 801⟩
else if $w > width(cur\_align)$ **then** $width(cur\_align) \leftarrow w$;
$type(u) \leftarrow unset\_node$; $span\_count(u) \leftarrow n$;
⟨Determine the stretch order 662⟩;
$glue\_order(u) \leftarrow o$; $glue\_stretch(u) \leftarrow total\_stretch[o]$;
⟨Determine the shrink order 668⟩;
$glue\_sign(u) \leftarrow o$; $glue\_shrink(u) \leftarrow total\_shrink[o]$;
pop_nest; $link(tail) \leftarrow u$; $tail \leftarrow u$;
end

This code is used in section 794.

800. 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 793⟩ +≡
$link(end\_span) \leftarrow max\_quarterword + 1$; $info(end\_span) \leftarrow null$;

801. ⟨Update width entry for spanned columns 801⟩ ≡

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 799.

802. 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; $append\_to\_vlist(p)$;
    if $cur\_head \neq cur\_tail$ **then**
      begin $link(tail) \leftarrow link(cur\_head)$; $tail \leftarrow cur\_tail$;
      end;
    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 $glue\_shrink \equiv shift\_amount$ }

803. 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* $\neq$ *align_group* **then** *confusion*(`"align1"`);
  unsave; { that *align_group* was for individual entries }
  if *cur_group* $\neq$ *align_group* **then** *confusion*(`"align0"`);
  unsave; { that *align_group* was for the whole alignment }
  if *nest*[*nest_ptr* − 1].*mode_field* = *mmode* **then** $o \leftarrow$ *display_indent*
  else $o \leftarrow 0$;
  ⟨ Go through the preamble list, determining the column widths and changing the alignrecords to dummy unset boxes 804 ⟩;
  ⟨ Package the preamble list, to determine the actual tabskip glue amounts, and let p point to this prototype box 807 ⟩;
  ⟨ Set the glue in all the unset boxes of the current list 808 ⟩;
  flush_node_list(p); *pop_alignment*; ⟨ Insert the current list into its environment 815 ⟩;
  end;
⟨ Declare the procedure called *align_peek* 788 ⟩

804. 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

$$w_j = \max_{1\le i\le j}\biggl(w_{ij} - \sum_{i\le k<j}(t_k + w_k)\biggr),$$

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 $1\le i\le j$ (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 804 ⟩ ≡
  $q \leftarrow link(preamble)$;
  repeat $flush\_list(u\_part(q))$; $flush\_list(v\_part(q))$; $p \leftarrow link(link(q))$;
    if $width(q) = null\_flag$ **then** ⟨ Nullify $width(q)$ and the tabskip glue following this column 805 ⟩;
    if $info(q) \ne end\_span$ **then**
      ⟨ Merge the widths in the span nodes of q with those of p, destroying the span nodes of q 806 ⟩;
    $type(q) \leftarrow unset\_node$; $span\_count(q) \leftarrow min\_quarterword$; $height(q) \leftarrow 0$; $depth(q) \leftarrow 0$;
    $glue\_order(q) \leftarrow normal$; $glue\_sign(q) \leftarrow normal$; $glue\_stretch(q) \leftarrow 0$; $glue\_shrink(q) \leftarrow 0$; $q \leftarrow p$;
  until $q = null$

This code is used in section 803.

805. ⟨ Nullify $width(q)$ and the tabskip glue following this column 805 ⟩ ≡
  begin $width(q) \leftarrow 0$; $r \leftarrow link(q)$; $s \leftarrow glue\_ptr(r)$;
  if $s \ne zero\_glue$ **then**
    begin $add\_glue\_ref(zero\_glue)$; $delete\_glue\_ref(s)$; $glue\_ptr(r) \leftarrow zero\_glue$;
    end;
  end

This code is used in section 804.

806. 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 $link(r) = n$ or $link(r) > n$ or $link(r) < n$.

⟨ Merge the widths in the span nodes of q with those of p, destroying the span nodes of q 806 ⟩ ≡

begin $t \leftarrow width(q) + width(glue\_ptr(link(q)))$; $r \leftarrow info(q)$; $s \leftarrow end\_span$; $info(s) \leftarrow p$;
$n \leftarrow min\_quarterword + 1$;
repeat $width(r) \leftarrow width(r) - t$; $u \leftarrow info(r)$;
  while $link(r) > n$ **do**
    begin $s \leftarrow info(s)$; $n \leftarrow link(info(s)) + 1$;
    end;
  if $link(r) < n$ **then**
    begin $info(r) \leftarrow info(s)$; $info(s) \leftarrow r$; $decr(link(r))$; $s \leftarrow r$;
    end
  else begin if $width(r) > width(info(s))$ **then** $width(info(s)) \leftarrow width(r)$;
    $free\_node(r, span\_node\_size)$;
    end;
  $r \leftarrow u$;
until $r = end\_span$;
end

This code is used in section 804.

807. 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 807 ⟩ ≡

$save\_ptr \leftarrow save\_ptr - 2$; $pack\_begin\_line \leftarrow -mode\_line$;
if $mode = -vmode$ **then**
  begin $rule\_save \leftarrow overfull\_rule$; $overfull\_rule \leftarrow 0$; { prevent rule from being packaged }
  $p \leftarrow hpack(preamble, saved(1), saved(0))$; $overfull\_rule \leftarrow rule\_save$;
  end
else begin $q \leftarrow link(preamble)$;
  repeat $height(q) \leftarrow width(q)$; $width(q) \leftarrow 0$; $q \leftarrow link(link(q))$;
  until $q = null$;
  $p \leftarrow vpack(preamble, saved(1), saved(0))$; $q \leftarrow link(preamble)$;
  repeat $width(q) \leftarrow height(q)$; $height(q) \leftarrow 0$; $q \leftarrow link(link(q))$;
  until $q = null$;
  end;
$pack\_begin\_line \leftarrow 0$

This code is used in section 803.

808. ⟨Set the glue in all the unset boxes of the current list 808⟩ ≡
$q \leftarrow link(head)$; $s \leftarrow head$;
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 810⟩
    else if $type(q) = rule\_node$ **then**
      ⟨Make the running dimensions in rule q extend to the boundaries of the alignment 809⟩;
  $s \leftarrow q$; $q \leftarrow link(q)$;
  end

This code is used in section 803.

809. ⟨Make the running dimensions in rule q extend to the boundaries of the alignment 809⟩ ≡
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 808.

810. 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 810⟩ ≡
begin if $mode = -vmode$ **then**
  begin $type(q) \leftarrow hlist\_node$; $width(q) \leftarrow width(p)$;
  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 811⟩;
  $r \leftarrow link(link(r))$; $s \leftarrow link(link(s))$;
until $r = null$;
end

This code is used in section 808.

811. 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 811⟩ ≡
$n \leftarrow span\_count(r)$; $t \leftarrow width(s)$; $w \leftarrow t$; $u \leftarrow hold\_head$;
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 812⟩;
  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 813⟩
else ⟨Make the unset node r into a $vlist\_node$ of height w, setting the glue as if the height were t 814⟩;
$shift\_amount(r) \leftarrow 0$;
if $u \neq hold\_head$ **then** { append blank boxes to account for spanned nodes }
  begin $link(u) \leftarrow link(r)$; $link(r) \leftarrow link(hold\_head)$; $r \leftarrow u$;
  end

This code is used in section 810.

812. ⟨Append tabskip glue and an empty box to list u, and update s and t as the prototype nodes are passed 812⟩ ≡
$s \leftarrow link(s)$; $v \leftarrow glue\_ptr(s)$; $link(u) \leftarrow new\_glue(v)$; $u \leftarrow link(u)$; $subtype(u) \leftarrow tab\_skip\_code + 1$;
$t \leftarrow t + width(v)$;
if $glue\_sign(p) = stretching$ **then**
  begin if $stretch\_order(v) = glue\_order(p)$ **then** $t \leftarrow t + round(float(glue\_set(p)) * stretch(v))$;
  end
else if $glue\_sign(p) = shrinking$ **then**
    begin if $shrink\_order(v) = glue\_order(p)$ **then** $t \leftarrow t - round(float(glue\_set(p)) * shrink(v))$;
    end;
$s \leftarrow link(s)$; $link(u) \leftarrow new\_null\_box$; $u \leftarrow link(u)$; $t \leftarrow t + width(s)$;
if $mode = -vmode$ **then** $width(u) \leftarrow width(s)$ **else begin** $type(u) \leftarrow vlist\_node$; $height(u) \leftarrow width(s)$;
  end

This code is used in section 811.

813. ⟨Make the unset node r into an $hlist\_node$ of width w, setting the glue as if the width were t 813⟩ ≡
begin $height(r) \leftarrow height(q)$; $depth(r) \leftarrow depth(q)$;
if $t = width(r)$ **then**
  begin $glue\_sign(r) \leftarrow normal$; $glue\_order(r) \leftarrow normal$; $set\_glue\_ratio\_zero(glue\_set(r))$;
  end
else if $t > width(r)$ **then**
    begin $glue\_sign(r) \leftarrow stretching$;
    if $glue\_stretch(r) = 0$ **then** $set\_glue\_ratio\_zero(glue\_set(r))$
    else $glue\_set(r) \leftarrow unfloat((t - width(r))/glue\_stretch(r))$;
    end
  else begin $glue\_order(r) \leftarrow glue\_sign(r)$; $glue\_sign(r) \leftarrow shrinking$;
    if $glue\_shrink(r) = 0$ **then** $set\_glue\_ratio\_zero(glue\_set(r))$
    else if $(glue\_order(r) = normal) \land (width(r) - t > glue\_shrink(r))$ **then**
        $set\_glue\_ratio\_one(glue\_set(r))$
      else $glue\_set(r) \leftarrow unfloat((width(r) - t)/glue\_shrink(r))$;
    end;
$width(r) \leftarrow w$; $type(r) \leftarrow hlist\_node$;
end

This code is used in section 811.

814. ⟨ Make the unset node r into a *vlist_node* of height w, setting the glue as if the height were t 814 ⟩ ≡

begin $width(r) \leftarrow width(q)$;
if $t = height(r)$ **then**
  begin $glue\_sign(r) \leftarrow normal$; $glue\_order(r) \leftarrow normal$; $set\_glue\_ratio\_zero(glue\_set(r))$;
  end
else if $t > height(r)$ **then**
    begin $glue\_sign(r) \leftarrow stretching$;
    if $glue\_stretch(r) = 0$ **then** $set\_glue\_ratio\_zero(glue\_set(r))$
    else $glue\_set(r) \leftarrow unfloat((t - height(r))/glue\_stretch(r))$;
    end
  else begin $glue\_order(r) \leftarrow glue\_sign(r)$; $glue\_sign(r) \leftarrow shrinking$;
    if $glue\_shrink(r) = 0$ **then** $set\_glue\_ratio\_zero(glue\_set(r))$
    else if $(glue\_order(r) = normal) \land (height(r) - t > glue\_shrink(r))$ **then**
        $set\_glue\_ratio\_one(glue\_set(r))$
      else $glue\_set(r) \leftarrow unfloat((height(r) - t)/glue\_shrink(r))$;
    end;
$height(r) \leftarrow w$; $type(r) \leftarrow vlist\_node$;
end

This code is used in section 811.

815. 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 815 ⟩ ≡

$aux\_save \leftarrow aux$; $p \leftarrow link(head)$; $q \leftarrow tail$; pop_nest;
if $mode = mmode$ **then** ⟨ Finish an alignment in a display 1209 ⟩
else begin $aux \leftarrow aux\_save$; $link(tail) \leftarrow p$;
  if $p \neq null$ **then** $tail \leftarrow q$;
  if $mode = vmode$ **then** $build\_page$;
  end

This code is used in section 803.

816. 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 & 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.

817. 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: *final_widow_penalty* is the amount of additional penalty to be inserted before the final line of the paragraph.

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` are 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 }

818. 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* 829 ⟩
procedure *line_break*(*final_widow_penalty* : *integer*);
  label *done*, *done1*, *done2*, *done3*, *done4*, *done5*, *continue*;
  var ⟨ Local variables for line breaking 865 ⟩
  begin *pack_begin_line* ← *mode_line*; { this is for over/underfull box messages }
  ⟨ Get ready to start line breaking 819 ⟩;
  ⟨ Find optimal breakpoints 866 ⟩;
  ⟨ Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list 879 ⟩;
  ⟨ Clean up the memory by removing the break nodes 868 ⟩;
  pack_begin_line ← 0;
  end;

819. 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 819 ⟩ ≡
link(*temp_head*) ← *link*(*head*);
if *is_char_node*(*tail*) **then** *tail_append*(*new_penalty*(*inf_penalty*))
else if *type*(*tail*) $\neq$ *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*); *init_cur_lang* ← *prev_graf* **mod** ´*200000*;
init_l_hyf ← *prev_graf* **div** ´*20000000*; *init_r_hyf* ← (*prev_graf* **div** ´*200000*) **mod** ´*100*; *pop_nest*;

See also sections 830, 837, and 851.

This code is used in section 818.

820. 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 }

821. 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.

822. 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* = 3 { number of words in 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 }

823. ⟨ Initialize the special list heads and constant nodes 793 ⟩ +≡
type(*last_active*) ← *hyphenated*; *line_number*(*last_active*) ← *max_halfword*; *subtype*(*last_active*) ← 0;
{ the *subtype* is never examined by the algorithm }

824. 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 }

825. 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 $mem[q+2\,..\,q+5].sc$; and the shrink difference appears in $mem[q+6].sc$. The *subtype* field of a delta node is not used.

define $delta\_node\_size = 7$ { number of words in a delta node }
define $delta\_node = 2$ { *type* field in a delta node }

826. 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

$$\textbf{for } k \leftarrow 1 \textbf{ to } 6 \textbf{ do } cur\_active\_width[k] \leftarrow cur\_active\_width[k] + mem[q+k].sc;$$

and we want to do this without the overhead of **for** loops. The *do_all_six* macro makes such six-tuples convenient.

define $do\_all\_six(\#) \equiv \#(1);\ \#(2);\ \#(3);\ \#(4);\ \#(5);\ \#(6)$

⟨ Global variables 13 ⟩ +≡
active_width: **array** [1 .. 6] **of** *scaled*; { distance from first active node to *cur_p* }
cur_active_width: **array** [1 .. 6] **of** *scaled*; { distance from current active node }
background: **array** [1 .. 6] **of** *scaled*; { length of an "empty" line }
break_width: **array** [1 .. 6] **of** *scaled*; { length being computed after current break }

827. 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 $x_1 \ldots x_n$ and let the skips have widths $y_1 \ldots y_n$, so that the paragraph can be represented by $x_1 y_1 \ldots x_n y_n$. Let p_i be the legal breakpoint at y_i; then $\alpha(p_i) = x_1 + y_1 + \cdots + x_i + y_i$, and $\beta(p_i) = x_1 + y_1 + \cdots + x_i$. 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 multiprecise 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.

828. 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? }

829. ⟨Declare subprocedures for *line_break* 829⟩ ≡

```
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 ← false; print_err("Infinite␣glue␣shrinkage␣found␣in␣a␣paragraph");
    help5("The␣paragraph␣just␣ended␣includes␣some␣glue␣that␣has")
    ("infinite␣shrinkability,␣e.g.,␣`\hskip␣0pt␣minus␣1fil´.")
    ("Such␣glue␣doesn´t␣belong␣there---it␣allows␣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;
    end;
  q ← new_spec(p); shrink_order(q) ← normal; delete_glue_ref(p); finite_shrink ← q;
  end;
```

See also sections 832, 880, 898, and 945.

This code is used in section 818.

830. ⟨ Get ready to start line breaking 819 ⟩ +≡
no_shrink_error_yet ← *true*;
check_shrinkage(*left_skip*); *check_shrinkage*(*right_skip*);
q ← *left_skip*; r ← *right_skip*; *background*[1] ← *width*(q) + *width*(r);
background[2] ← 0; *background*[3] ← 0; *background*[4] ← 0; *background*[5] ← 0;
background[2 + *stretch_order*(q)] ← *stretch*(q);
background[2 + *stretch_order*(r)] ← *background*[2 + *stretch_order*(r)] + *stretch*(r);
background[6] ← *shrink*(q) + *shrink*(r);

831. 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*.

⟨ Global variables 13 ⟩ +≡
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 }

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

The parameter *pi* to *try_break* is the penalty associated with a break at *cur_p*; we have *pi* = *eject_penalty* if the break is forced, and *pi* = *inf_penalty* if the break is illegal.

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

define *copy_to_cur_active*(#) ≡ *cur_active_width*[#] ← *active_width*[#]
define *deactivate* = 60 { go here when node *r* should be deactivated }

⟨ Declare subprocedures for *line_break* 829 ⟩ +≡
procedure *try_break*(*pi* : *integer*; *break_type* : *small_number*);
  label *exit*, *done*, *done1*, *continue*, *deactivate*;
  var *r*: *pointer*; { runs through the active list }
    prev_r: *pointer*; { stays a step behind *r* }
    old_l: *halfword*; { maximum line number in current equivalence class of lines }
    no_break_yet: *boolean*; { have we found a feasible break at *cur_p*? }
    ⟨ Other local variables for *try_break* 833 ⟩
  begin ⟨ Make sure that *pi* is in the proper range 834 ⟩;
  no_break_yet ← *true*; *prev_r* ← *active*; *old_l* ← 0; *do_all_six*(*copy_to_cur_active*);
  loop begin *continue*: *r* ← *link*(*prev_r*); ⟨ If node *r* is of type *delta_node*, update *cur_active_width*, set *prev_r* and *prev_prev_r*, then **goto** *continue* 835 ⟩;
    ⟨ 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* 838 ⟩;
    ⟨ 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 854 ⟩;
    end;
exit: **stat** ⟨ Update the value of *printed_node* for symbolic displays 861 ⟩ **tats**
  end;

833. ⟨ Other local variables for *try_break* 833 ⟩ ≡
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 }

This code is used in section 832.

834. ⟨ Make sure that *pi* is in the proper range 834 ⟩ ≡
  if *abs*(*pi*) $\geq$ *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 832.

835. The following code uses the fact that *type*(*last_active*) ≠ *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* 835 ⟩ ≡
  if *type*(*r*) = *delta_node* **then**
    begin *do_all_six*(*update_width*); *prev_prev_r* ← *prev_r*; *prev_r* ← *r*; **goto** *continue*;
    end

This code is used in section 832.

836. 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, *minimal_demerits*[*tight_fit*] contains the fewest total demerits of feasible line breaks ending at *cur_p* with a *tight_fit* line; *best_place*[*tight_fit*] points to the passive node for the break before *cur_p* that achieves such an optimum; and *best_pl_line*[*tight_fit*] is the *line_number* field in the active node corresponding to *best_place*[*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* ≡ *´7777777777* { 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 }

837. ⟨ Get ready to start line breaking 819 ⟩ +≡
  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*;

838. 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* = *last_active* and *line_number*(*last_active*) > *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* = *last_active*, otherwise compute the new *line_width* 838 ⟩ ≡
  begin *l* ← *line_number*(*r*);
  if *l* > *old_l* **then**
    begin { now we are no longer in the inner loop }
    if (*minimum_demerits* < *awful_bad*) ∧ ((*old_l* ≠ *easy_line*) ∨ (*r* = *last_active*)) **then**
      ⟨ Create new active nodes for the best feasible breaks just found 839 ⟩;
    if *r* = *last_active* **then return**;
    ⟨ Compute the new line width 853 ⟩;
    end;
  end

This code is used in section 832.

839. It is not necessary to create new active nodes having *minimal_demerits* greater than *minimum_demerits* + *abs*(*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 839⟩ ≡
  begin if *no_break_yet* **then** ⟨Compute the values of *break_width* 840⟩;
  ⟨Insert a delta node to prepare for breaks at *cur_p* 846⟩;
  if *abs*(*adj_demerits*) $\geq$ *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*] $\leq$ *minimum_demerits* **then**
      ⟨Insert a new active node from *best_place*[*fit_class*] to *cur_p* 848⟩;
    minimal_demerits[*fit_class*] ← *awful_bad*;
    end;
  minimum_demerits ← *awful_bad*; ⟨Insert a delta node to prepare for the next active node 847⟩;
  end

This code is used in section 838.

840. 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 'δ *cur_p* δ'' before a, where $\delta = \alpha(a) - \alpha(cur\_p)$ and $\delta' = \alpha(cur\_p) - \alpha(a)$ in the notation explained above. The *cur_active_width* array now holds $\gamma + \beta(cur\_p) - \alpha(a)$; so δ can be obtained by subtracting *cur_active_width* from the quantity $\gamma + \beta(cur\_p) - \alpha(cur\_p)$. 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* 840⟩ ≡
  begin *no_break_yet* ← *false*; *do_all_six*(*set_break_width_to_background*); *s* ← *cur_p*;
  if *break_type* > *unhyphenated* **then**
    if *cur_p* $\neq$ *null* **then** ⟨Compute the discretionary *break_width* values 843⟩;
  while *s* $\neq$ *null* **do**
    begin if *is_char_node*(*s*) **then goto** *done*;
    case *type*(*s*) **of**
    glue_node: ⟨Subtract glue from *break_width* 841⟩;
    penalty_node: *do_nothing*;
    math_node: *break_width*[1] ← *break_width*[1] − *width*(*s*);
    kern_node: **if** *subtype*(*s*) $\neq$ *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 839.

841. ⟨Subtract glue from *break_width* 841⟩ ≡
begin $v \leftarrow$ *glue_ptr*(s); *break_width*[1] $\leftarrow$ *break_width*[1] $-$ *width*(v);
break_width[2 + *stretch_order*(v)] $\leftarrow$ *break_width*[2 + *stretch_order*(v)] $-$ *stretch*(v);
break_width[6] $\leftarrow$ *break_width*[6] $-$ *shrink*(v);
end

This code is used in section 840.

842. 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 $\gamma+\beta(q)-\alpha(cur\_p)$, where $\beta(q)=\beta(cur\_p)-l_0+l$. 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*.

⟨Global variables 13⟩ +≡
disc_width: *scaled*; { the length of discretionary material preceding a break }

843. ⟨Compute the discretionary *break_width* values 843⟩ ≡
begin $t \leftarrow$ *replace_count*(*cur_p*); $v \leftarrow$ *cur_p*; $s \leftarrow$ *post_break*(*cur_p*);
while $t > 0$ **do**
  begin *decr*(t); $v \leftarrow$ *link*(v); ⟨Subtract the width of node v from *break_width* 844⟩;
  end;
while $s \neq$ *null* **do**
  begin ⟨Add the width of node s to *break_width* 845⟩;
  $s \leftarrow$ *link*(s);
  end;
break_width[1] $\leftarrow$ *break_width*[1] + *disc_width*;
if *post_break*(*cur_p*) = *null* **then** $s \leftarrow$ *link*(v); { nodes may be discardable after the break }
end

This code is used in section 840.

844. Replacement texts and discretionary texts are supposed to contain only character nodes, kern nodes, ligature nodes, and box or rule nodes.

⟨Subtract the width of node v from *break_width* 844⟩ ≡
if *is_char_node*(v) **then**
  begin $f \leftarrow$ *font*(v);
  if *is_wchar_node*(v) **then** *break_width*[1] $\leftarrow$ *break_width*[1] $-$ *cfont_width*[f]
  else *break_width*[1] $\leftarrow$ *break_width*[1] $-$ *char_width*(f)(*char_info*(f)(*character*(v)));
  end
else case *type*(v) **of**
  ligature_node: **begin** $f \leftarrow$ *font*(*lig_char*(v));
    if *is_wchar*(*character*(*lig_char*(v))) **then** *break_width*[1] $\leftarrow$ *break_width*[1] $-$ *cfont_width*[f]
    else *break_width*[1] $\leftarrow$ *break_width*[1] $-$ *char_width*(f)(*char_info*(f)(*character*(*lig_char*(v))));
    end;
  hlist_node, *vlist_node*, *rule_node*, *kern_node*: *break_width*[1] $\leftarrow$ *break_width*[1] $-$ *width*(v);
  othercases *confusion*("disc1")
  endcases

This code is used in section 843.

845. ⟨Add the width of node s to *break_width* 845⟩ ≡
if *is_char_node*(*s*) **then**
begin $f \leftarrow font(s)$;
if *is_wchar_node*(*s*) **then** $break\_width[1] \leftarrow break\_width[1] + cfont\_width[f]$
else $break\_width[1] \leftarrow break\_width[1] + char\_width(f)(char\_info(f)(character(s)))$;
end
else case *type*(*s*) **of**
ligature_node: **begin** $f \leftarrow font(lig\_char(s))$;
if *is_wchar*(*character*(*lig_char*(*s*))) **then** $break\_width[1] \leftarrow break\_width[1] + cfont\_width[f]$
else $break\_width[1] \leftarrow break\_width[1] + char\_width(f)(char\_info(f)(character(lig\_char(s))))$;
end;
hlist_node, *vlist_node*, *rule_node*, *kern_node*: $break\_width[1] \leftarrow break\_width[1] + width(s)$;
othercases *confusion*("disc2")
endcases

This code is used in section 843.

846. 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* 846⟩ ≡
if $type(prev\_r) = delta\_node$ **then** { modify an existing delta node }
begin *do_all_six*(*convert_to_break_width*);
end
else if $prev\_r = active$ **then** { no delta node needed at the beginning }
begin *do_all_six*(*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_six(*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 839.

847. When the following code is performed, we will have just inserted at least one active node before r, so $type(prev\_r) \neq delta\_node$.

define $new\_delta\_from\_break\_width(\#) \equiv mem[q + \#].sc \leftarrow cur\_active\_width[\#] - break\_width[\#]$

⟨Insert a delta node to prepare for the next active node 847⟩ ≡
if $r \neq last\_active$ **then**
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_six(*new_delta_from_break_width*); $link(prev\_r) \leftarrow q$; $prev\_prev\_r \leftarrow prev\_r$; $prev\_r \leftarrow q$;
end

This code is used in section 839.

848. 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* 848⟩ ≡
begin $q \leftarrow$ *get_node*(*passive_node_size*); *link*(q) $\leftarrow$ *passive*; *passive* $\leftarrow q$; *cur_break*(q) $\leftarrow$ *cur_p*;
stat *incr*(*pass_number*); *serial*(q) $\leftarrow$ *pass_number*; **tats**
prev_break(q) $\leftarrow$ *best_place*[*fit_class*];
$q \leftarrow$ *get_node*(*active_node_size*); *break_node*(q) $\leftarrow$ *passive*; *line_number*(q) $\leftarrow$ *best_pl_line*[*fit_class*] + 1;
fitness(q) $\leftarrow$ *fit_class*; *type*(q) $\leftarrow$ *break_type*; *total_demerits*(q) $\leftarrow$ *minimal_demerits*[*fit_class*];
link(q) $\leftarrow r$; *link*(*prev_r*) $\leftarrow q$; *prev_r* $\leftarrow q$;
stat if *tracing_paragraphs* > 0 **then** ⟨Print a symbolic description of the new break node 849⟩;
tats
end

This code is used in section 839.

849. ⟨Print a symbolic description of the new break node 849⟩ ≡
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)); *print*("␣->␣@@");
if *prev_break*(*passive*) = *null* **then** *print_char*("0")
else *print_int*(*serial*(*prev_break*(*passive*)));
end

This code is used in section 848.

850. 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 = *hang_after*; if $n \geq 0$, hanging indentation takes place on lines $n+1$, $n+2$, ..., otherwise it takes place on lines 1, ..., $|n|$. When hanging indentation is active, the left margin is *hang_indent*, if *hang_indent* ≥ 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* }

851. 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 819⟩ +≡
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 852⟩
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*

852. ⟨Set line length parameters in preparation for hanging indentation 852⟩ ≡
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 851.

853. 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 lth 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 853⟩ ≡
if l > *easy_line* **then**
  begin *line_width* ← *second_width*; *old_l* ← *max_halfword* − 1;
  end
else begin *old_l* ← l;
  if l > *last_special_line* **then** *line_width* ← *second_width*
  else if *par_shape_ptr* = *null* **then** *line_width* ← *first_width*
    else *line_width* ← *mem*[*par_shape_ptr* + 2 ∗ l].*sc*;
  end

This code is used in section 838.

854. 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 854⟩ ≡

```
begin artificial_demerits ← false;
shortfall ← line_width − cur_active_width[1]; { we're this much too short }
if shortfall > 0 then
   ⟨Set the value of b to the badness for stretching the line, and compute the corresponding fit_class 855⟩
else ⟨Set the value of b to the badness for shrinking the line, and compute the corresponding fit_class 856⟩;
if (b > inf_bad) ∨ (pi = eject_penalty) then ⟨Prepare to deactivate node r, and goto deactivate unless
      there is a reason to consider lines of text from r to cur_p 857⟩
else begin prev_r ← r;
   if b > threshold then goto continue;
   node_r_stays_active ← true;
   end;
⟨Record a new feasible break 858⟩;
if node_r_stays_active then goto continue; { prev_r has been set to r }
deactivate: ⟨Deactivate node r 863⟩;
end
```

This code is used in section 832.

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

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

⟨Set the value of b to the badness for stretching the line, and compute the corresponding *fit_class* 855⟩ ≡

```
if (cur_active_width[3] ≠ 0) ∨ (cur_active_width[4] ≠ 0) ∨ (cur_active_width[5] ≠ 0) then
   begin 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 854.

856. 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* 856 ⟩ ≡
  begin if $-$*shortfall* $>$ *cur_active_width*[6] **then** $b \leftarrow$ *inf_bad* $+ 1$
  else $b \leftarrow$ *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 854.

857. 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* 857 ⟩ ≡
  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 854.

858. 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 858 ⟩ ≡
  if *artificial_demerits* **then** $d \leftarrow 0$
  else ⟨ Compute the demerits, d, from r to *cur_p* 862 ⟩;
  stat if *tracing_paragraphs* > 0 **then** ⟨ Print a symbolic description of this feasible break 859 ⟩;
  tats
  $d \leftarrow d +$ *total_demerits*(r); { this is the minimum total demerits from the beginning to *cur_p* via r }
  if $d \le$ *minimal_demerits*[*fit_class*] **then**
    begin *minimal_demerits*[*fit_class*] ← d; *best_place*[*fit_class*] ← *break_node*(r); *best_pl_line*[*fit_class*] ← l;
    if $d <$ *minimum_demerits* **then** *minimum_demerits* ← d;
    end

This code is used in section 854.

859. ⟨Print a symbolic description of this feasible break 859⟩ ≡

```
begin if printed_node ≠ cur_p then
  ⟨Print the list between printed_node and cur_p, then set printed_node ← cur_p 860⟩;
print_nl("@");
if cur_p = null then print_esc("par")
else if type(cur_p) ≠ 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 858.

860. ⟨Print the list between *printed_node* and *cur_p*, then set *printed_node* ← *cur_p* 860⟩ ≡

```
begin print_nl("");
if cur_p = null then short_display(link(printed_node))
else begin save_link ← link(cur_p); link(cur_p) ← null; print_nl("");
  short_display(link(printed_node)); link(cur_p) ← save_link;
  end;
printed_node ← cur_p;
end
```

This code is used in section 859.

861. 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 861⟩ ≡

```
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 832.

862. ⟨Compute the demerits, d, from r to *cur_p* 862⟩ ≡

```
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(intcast(fit_class) − intcast(fitness(r))) > 1 then d ← d + adj_demerits;
end
```

This code is used in section 858.

863. 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 863⟩ ≡

```
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 864⟩
else if type(prev_r) = delta_node then
    begin r ← link(prev_r);
    if r = last_active then
      begin do_all_six(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_six(update_width); do_all_six(combine_two_deltas); link(prev_r) ← link(r);
        free_node(r, delta_node_size);
        end;
    end
```

This code is used in section 854.

864. The following code uses the fact that *type*(*last_active*) ≠ *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 864⟩ ≡

```
begin r ← link(active);
if type(r) = delta_node then
  begin do_all_six(update_active); do_all_six(copy_to_cur_active); link(active) ← link(r);
  free_node(r, delta_node_size);
  end;
end
```

This code is used in section 863.

865. 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 865⟩ ≡
auto_breaking: *boolean*; { is node *cur_p* outside a formula? }
prev_p: *pointer*; { helps to determine when glue nodes are breakpoints }
q, *r*, *s*, *prev_s*: *pointer*; { miscellaneous nodes of temporary interest }
f: *internal_font_number*; { used when calculating character widths }

See also section 896.

This code is used in section 818.

866. 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 866 ⟩ ≡
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 894 ⟩;
⟨ Create an active breakpoint representing the beginning of the paragraph 867 ⟩;
cur_p ← *link*(*temp_head*); *auto_breaking* ← *true*;
prev_p ← *cur_p*; { glue at beginning is not a legal breakpoint }
while (*cur_p* ≠ *null*) ∧ (*link*(*active*) ≠ *last_active*) **do** ⟨ Call *try_break* if *cur_p* is a legal breakpoint; on the second pass, also try to hyphenate the next word, if *cur_p* is a glue node; then advance *cur_p* to the next node of the paragraph that could possibly be a legal breakpoint 869 ⟩;
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 876 ⟩;
⟨ Clean up the memory by removing the break nodes 868 ⟩;
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

This code is used in section 818.

867. 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 867 ⟩ ≡
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*;
do_all_six(*store_background*);
passive ← *null*; *printed_node* ← *temp_head*; *pass_number* ← 0; *cfont_in_short_display* ← *null_cfont*;
font_in_short_display ← *null_font*

This code is used in section 866.

868. ⟨Clean up the memory by removing the break nodes 868⟩ ≡

$q \leftarrow link(active)$;
while $q \neq last\_active$ **do**
  begin $cur\_p \leftarrow link(q)$;
  if $type(q) = delta\_node$ **then** $free\_node(q, delta\_node\_size)$
  else $free\_node(q, active\_node\_size)$;
  $q \leftarrow cur\_p$;
  end;
$q \leftarrow passive$;
while $q \neq null$ **do**
  begin $cur\_p \leftarrow link(q)$; $free\_node(q, passive\_node\_size)$; $q \leftarrow cur\_p$;
  end

This code is used in sections 818 and 866.

869. 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 869 ⟩ ≡
begin if *is_char_node*(*cur_p*) **then**
 ⟨ Advance *cur_p* to the node following the present string of characters 870 ⟩;
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 1365 ⟩;
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*) 871 ⟩;
 if *second_pass* ∧ *auto_breaking* **then** ⟨ Try to hyphenate the following word 897 ⟩;
 end;
kern_node: **if** *subtype*(*cur_p*) = *explicit* **then** *kern_break*
 else *act_width* ← *act_width* + *width*(*cur_p*);
ligature_node: **begin** *f* ← *font*(*lig_char*(*cur_p*));
 if *is_wchar*(*character*(*lig_char*(*cur_p*))) **then** *act_width* ← *act_width* + *cfont_width*[*f*]
 else *act_width* ← *act_width* + *char_width*(*f*)(*char_info*(*f*)(*character*(*lig_char*(*cur_p*))));
 end;
disc_node: ⟨ Try to break after a discretionary fragment, then **goto** *done5* 872 ⟩;
math_node: **begin** *auto_breaking* ← (*subtype*(*cur_p*) = *after*); *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 866.

870. 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 *link*(*cur_p*) = *null* when *cur_p* is a character node.

⟨ Advance *cur_p* to the node following the present string of characters 870 ⟩ ≡
begin *prev_p* ← *cur_p*;
repeat *f* ← *font*(*cur_p*);
 if *is_wchar_node*(*cur_p*) **then** *act_width* ← *act_width* + *cfont_width*[*f*]
 else *act_width* ← *act_width* + *char_width*(*f*)(*char_info*(*f*)(*character*(*cur_p*)));
 cur_p ← *link*(*cur_p*);
until ¬*is_char_node*(*cur_p*);
end

This code is used in section 869.

871. 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 *cur_p* is a legal breakpoint, call *try_break*; then update the active widths by including the glue in *glue_ptr*(*cur_p*) 871 ⟩ ≡

```
if auto_breaking then
  begin if is_char_node(prev_p) then try_break(0, unhyphenated)
  else if precedes_break(prev_p) then try_break(0, unhyphenated)
    else if (type(prev_p) = kern_node) ∧ (subtype(prev_p) ≠ explicit) then try_break(0, unhyphenated);
  end;
check_shrinkage(glue_ptr(cur_p)); q ← glue_ptr(cur_p); act_width ← act_width + width(q);
active_width[2 + stretch_order(q)] ← active_width[2 + stretch_order(q)] + stretch(q);
active_width[6] ← active_width[6] + shrink(q)
```

This code is used in section 869.

872. 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** *done5* 872 ⟩ ≡

```
begin s ← pre_break(cur_p); disc_width ← 0;
if s = null then try_break(ex_hyphen_penalty, hyphenated)
else begin repeat ⟨ Add the width of node s to disc_width 873 ⟩;
    s ← link(s);
  until s = null;
  act_width ← act_width + disc_width; try_break(hyphen_penalty, hyphenated);
  act_width ← act_width − disc_width;
  end;
r ← replace_count(cur_p); s ← link(cur_p);
while r > 0 do
  begin ⟨ Add the width of node s to act_width 874 ⟩;
  decr(r); s ← link(s);
  end;
prev_p ← cur_p; cur_p ← s; goto done5;
end
```

This code is used in section 869.

873. ⟨ Add the width of node *s* to *disc_width* 873 ⟩ ≡

```
if is_char_node(s) then
  begin f ← font(s);
  if is_wchar_node(s) then disc_width ← disc_width + cfont_width[f]
  else disc_width ← disc_width + char_width(f)(char_info(f)(character(s)));
  end
else case type(s) of
  ligature_node: begin f ← font(lig_char(s));
    if is_wchar(character(lig_char(s))) then disc_width ← disc_width + cfont_width[f]
    else disc_width ← disc_width + char_width(f)(char_info(f)(character(lig_char(s))));
    end;
  hlist_node, vlist_node, rule_node, kern_node: disc_width ← disc_width + width(s);
  othercases confusion("disc3")
  endcases
```

This code is used in section 872.

874. ⟨ Add the width of node s to *act_width* 874 ⟩ ≡

```
if is_char_node(s) then
  begin f ← font(s);
  if is_wchar_node(s) then act_width ← act_width + cfont_width[f]
  else act_width ← act_width + char_width(f)(char_info(f)(character(s)));
  end
else case type(s) of
  ligature_node: begin f ← font(lig_char(s));
    if is_wchar(character(lig_char(s))) then act_width ← act_width + cfont_width[f]
    else act_width ← act_width + char_width(f)(char_info(f)(character(lig_char(s))));
    end;
  hlist_node, vlist_node, rule_node, kern_node: act_width ← act_width + width(s);
  othercases confusion("disc4")
  endcases
```

This code is used in section 872.

875. 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 }
```

876. ⟨ Try the final line break at the end of the paragraph, and **goto** *done* if the desired breakpoints have been found 876 ⟩ ≡

```
begin try_break(eject_penalty, hyphenated);
if link(active) ≠ last_active then
  begin ⟨ Find an active node with fewest demerits 877 ⟩;
  if looseness = 0 then goto done;
  ⟨ Find the best active node for the desired looseness 878 ⟩;
  if (actual_looseness = looseness) ∨ final_pass then goto done;
  end;
end
```

This code is used in section 866.

877. ⟨ Find an active node with fewest demerits 877 ⟩ ≡

```
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 876.

878. 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 878 ⟩ ≡

begin $r \gets link(active)$; $actual\_looseness \gets 0$;
repeat if $type(r) \neq delta\_node$ **then**
  begin $line\_diff \gets intcast(line\_number(r)) - intcast(best\_line)$;
  if $((line\_diff < actual\_looseness) \land (looseness \leq line\_diff)) \lor$
    $((line\_diff > actual\_looseness) \land (looseness \geq line\_diff))$ **then**
   begin $best\_bet \gets r$; $actual\_looseness \gets line\_diff$; $fewest\_demerits \gets total\_demerits(r)$;
   end
  else if $(line\_diff = actual\_looseness) \land (total\_demerits(r) < fewest\_demerits)$ **then**
    begin $best\_bet \gets r$; $fewest\_demerits \gets total\_demerits(r)$;
    end;
  end;
 $r \gets link(r)$;
until $r = last\_active$;
$best\_line \gets line\_number(best\_bet)$;
end

This code is used in section 876.

879. 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 879 ⟩ ≡

$post\_line\_break(final\_widow\_penalty)$

This code is used in section 818.

880. 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* 829 ⟩ +≡
procedure *post_line_break*(*final_widow_penalty* : *integer*);
 label *done*, *done1*;
 var *q*, *r*, *s*: *pointer*; { temporary registers for list manipulation }
  disc_break: *boolean*; { was the current break at a discretionary node? }
  post_disc_break: *boolean*; { and did it have a nonempty post-break part? }
  cur_width: *scaled*; { width of line number *cur_line* }
  cur_indent: *scaled*; { left margin of line number *cur_line* }
  t: *quarterword*; { used for replacement counts in discretionary nodes }
  pen: *integer*; { use when calculating penalties between lines }
  cur_line: *halfword*; { the current line number being justified }
 begin ⟨ Reverse the links of the relevant passive nodes, setting *cur_p* to the first breakpoint 881 ⟩;
 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 883 ⟩;
  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 882 ⟩;
 until *cur_p* = *null*;
 if (*cur_line* ≠ *best_line*) ∨ (*link*(*temp_head*) ≠ *null*) **then** *confusion*("line␣breaking");
 prev_graf ← *best_line* − 1;
 end;

881. 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 881 ⟩ ≡
 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 880.

882. 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 882⟩ ≡
  begin $r \leftarrow$ *temp_head*;
  loop begin $q \leftarrow$ *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*;
    $r \leftarrow q$; { now *type*(q) = *glue_node*, *kern_node*, *math_node* or *penalty_node* }
    end;
done1: **if** $r \neq$ *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 880.

883. 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 883⟩ ≡
  ⟨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* 884⟩;
  ⟨Put the `\leftskip` glue at the left and detach this line 890⟩;
  ⟨Call the packaging subroutine, setting *just_box* to the justified box 892⟩;
  ⟨Append the new box to the current vertical list, followed by the list of special nodes taken out of the box by the packager 891⟩;
  ⟨Append a penalty node, if a nonzero penalty is appropriate 893⟩

This code is used in section 880.

884. 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* 884 ⟩ ≡

$q \leftarrow cur\_break(cur\_p)$; $disc\_break \leftarrow false$; $post\_disc\_break \leftarrow false$;
if $q \neq null$ **then** { q cannot be a *char_node* }
  if $type(q) = glue\_node$ **then**
    begin $delete\_glue\_ref(glue\_ptr(q))$; $glue\_ptr(q) \leftarrow right\_skip$; $subtype(q) \leftarrow right\_skip\_code + 1$;
    $add\_glue\_ref(right\_skip)$; **goto** *done*;
    end
  else begin if $type(q) = disc\_node$ **then**
      ⟨ Change discretionary to compulsory and set $disc\_break \leftarrow true$ 885 ⟩
    else if $(type(q) = math\_node) \lor (type(q) = kern\_node)$ **then** $width(q) \leftarrow 0$;
    end
else begin $q \leftarrow temp\_head$;
  while $link(q) \neq null$ **do** $q \leftarrow link(q)$;
  end;
⟨ Put the `\rightskip` glue after node q 889 ⟩;
done:

This code is used in section 883.

885. ⟨ Change discretionary to compulsory and set $disc\_break \leftarrow true$ 885 ⟩ ≡

begin $t \leftarrow replace\_count(q)$;
⟨ Destroy the t nodes following q, and make r point to the following node 886 ⟩;
if $post\_break(q) \neq null$ **then** ⟨ Transplant the post-break list 887 ⟩;
if $pre\_break(q) \neq null$ **then** ⟨ Transplant the pre-break list 888 ⟩;
$link(q) \leftarrow r$; $disc\_break \leftarrow true$;
end

This code is used in section 884.

886. ⟨ Destroy the t nodes following q, and make r point to the following node 886 ⟩ ≡

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 885.

887. 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.

⟨ Transplant the post-break list 887 ⟩ ≡

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 885.

888. 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.

⟨Transplant the pre-break list 888⟩ ≡
  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 885.

889. ⟨Put the `\rightskip` glue after node q 889⟩ ≡
  $r \leftarrow new\_param\_glue(right\_skip\_code)$; $link(r) \leftarrow link(q)$; $link(q) \leftarrow r$; $q \leftarrow r$

This code is used in section 884.

890. 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.

⟨Put the `\leftskip` glue at the left and detach this line 890⟩ ≡
  $r \leftarrow link(q)$; $link(q) \leftarrow null$; $q \leftarrow link(temp\_head)$; $link(temp\_head) \leftarrow r$;
  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 883.

891. ⟨Append the new box to the current vertical list, followed by the list of special nodes taken out of the box by the packager 891⟩ ≡
  $append\_to\_vlist(just\_box)$;
  if $adjust\_head \neq adjust\_tail$ **then**
    begin $link(tail) \leftarrow link(adjust\_head)$; $tail \leftarrow adjust\_tail$;
    end;
  $adjust\_tail \leftarrow null$

This code is used in section 883.

892. 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 892⟩ ≡
  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;
  $adjust\_tail \leftarrow adjust\_head$; $just\_box \leftarrow hpack(q, cur\_width, exactly)$; $shift\_amount(just\_box) \leftarrow cur\_indent$

This code is used in section 883.

893. 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 893⟩ ≡
  if $cur\_line + 1 \neq best\_line$ **then**
    begin $pen \leftarrow inter\_line\_penalty$;
    if $cur\_line = prev\_graf + 1$ **then** $pen \leftarrow pen + club\_penalty$;
    if $cur\_line + 2 = best\_line$ **then** $pen \leftarrow pen + final\_widow\_penalty$;
    if $disc\_break$ **then** $pen \leftarrow pen + broken\_penalty$;
    if $pen \neq 0$ **then**
      begin $r \leftarrow new\_penalty(pen)$; $link(tail) \leftarrow r$; $tail \leftarrow r$;
      end;
    end

This code is used in section 883.

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

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

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

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

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

⟨ Initialize for hyphenating a paragraph 894 ⟩ ≡
  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*;
  end

This code is used in section 866.

895. The letters $c_1 \ldots c_n$ 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: *small_number*; { the number of positions occupied in *hc* }
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 }

896. Hyphenation routines need a few more local variables.

⟨ Local variables for line breaking 865 ⟩ +≡
j: *small_number*; { an index into *hc* or *hu* }
c: 0 .. 255; { character being considered for hyphenation }

897. 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 897 ⟩ ≡
  begin *prev_s* ← *cur_p*; *s* ← *link*(*prev_s*);
  if $s \neq null$ **then**
    begin ⟨ Skip to node *ha*, or **goto** *done1* if no hyphenation should be attempted 899 ⟩;
    if *l_hyf* + *r_hyf* > 63 **then goto** *done1*;
    ⟨ Skip to node *hb*, putting letters into *hu* and *hc* 900 ⟩;
    ⟨ Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto** *done1* 902 ⟩;
    hyphenate;
    end;
done1: **end**

This code is used in section 869.

898. ⟨ Declare subprocedures for *line_break* 829 ⟩ +≡
⟨ Declare the function called *reconstitute* 909 ⟩
procedure *hyphenate*;
  label *common_ending*, *done*, *found*, *found1*, *found2*, *not_found*, *exit*;
  var ⟨ Local variables for hyphenation 904 ⟩
  begin ⟨ Find hyphen locations for the word in *hc*, or **return** 926 ⟩;
  ⟨ If no hyphens were found, **return** 905 ⟩;
  ⟨ Replace nodes *ha* .. *hb* by a sequence of nodes that includes the discretionary hyphens 906 ⟩;
exit: **end**;

899. 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 899 ⟩ ≡
  loop begin if *is_char_node*(*s*) **then**
      begin $c \leftarrow qo(character(s))$; $hf \leftarrow font(s)$;
      end
    else if $type(s) = ligature\_node$ **then**
        if $lig\_ptr(s) = null$ **then goto** *continue*
        else begin $q \leftarrow lig\_ptr(s)$; $c \leftarrow qo(character(q))$; $hf \leftarrow font(q)$;
          end
      else if $(type(s) = kern\_node) \land (subtype(s) = normal)$ **then goto** *continue*
        else if $type(s) = whatsit\_node$ **then**
            begin ⟨ Advance past a whatsit node in the pre-hyphenation loop 1366 ⟩;
            goto *continue*;
            end
          else goto *done1*;
    if $lc\_code(c) \neq 0$ **then**
      if $(lc\_code(c) = c) \lor (uc\_hyph > 0)$ **then goto** *done2*
      else goto *done1*;
  continue: $prev\_s \leftarrow s$; $s \leftarrow link(prev\_s)$;
    end;
done2: $hyf\_char \leftarrow hyphen\_char[hf]$;
  if $hyf\_char < 0$ **then goto** *done1*;
  if $hyf\_char > 255$ **then goto** *done1*;
  $ha \leftarrow prev\_s$

This code is used in section 897.

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

⟨ Skip to node *hb*, putting letters into *hu* and *hc* 900 ⟩ ≡
  $hn \leftarrow 0$;
  loop begin if *is_char_node*(*s*) **then**
      begin if $font(s) \neq hf$ **then goto** *done3*;
      $hyf\_bchar \leftarrow character(s)$; $c \leftarrow qo(hyf\_bchar)$;
      if $lc\_code(c) = 0$ **then goto** *done3*;
      if $hn = 63$ **then goto** *done3*;
      $hb \leftarrow s$; *incr*(*hn*); $hu[hn] \leftarrow c$; $hc[hn] \leftarrow lc\_code(c)$; $hyf\_bchar \leftarrow non\_char$;
      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 901 ⟩
      else if $(type(s) = kern\_node) \land (subtype(s) = normal)$ **then**
          begin $hb \leftarrow s$; $hyf\_bchar \leftarrow font\_bchar[hf]$;
          end
        else goto *done3*;
    $s \leftarrow link(s)$;
    end;
done3:

This code is used in section 897.

901. 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 901⟩ ≡

```
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));
  if lc_code(c) = 0 then goto done3;
  if j = 63 then goto done3;
  incr(j); hu[j] ← c; hc[j] ← lc_code(c);
  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 900.

902. ⟨Check that the nodes following hb permit hyphenation and that at least $l\_hyf + r\_hyf$ letters have been found, otherwise **goto** *done1* 902⟩ ≡

```
if hn < l_hyf + r_hyf then goto done1;  { l_hyf and r_hyf are ≥ 1 }
loop begin if ¬(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;
    othercases goto done1
    endcases;
  s ← link(s);
  end;
done4:
```

This code is used in section 897.

903. 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? }

904. ⟨Local variables for hyphenation 904⟩ ≡
i, j, l: 0 . . 65; { indices into hc or hu }
q, r, s: *pointer*; { temporary registers for list manipulation }
bchar: *halfword*; { right boundary character of hyphenated word, or *non_char* }

See also sections 915, 925, and 932.

This code is used in section 898.

905. TEX will never insert a hyphen that has fewer than `\lefthyphenmin` 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** 905⟩ ≡
  for $j \leftarrow l\_hyf$ **to** $hn - r\_hyf$ **do**
    if $odd(hyf[j])$ **then goto** *found1*;
  return;
found1:

This code is used in section 898.

906. 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 906⟩ ≡
  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 916⟩;
  flush_list(init_list)
```

This code is used in section 898.

907. 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`.

908. The processing is facilitated by a subroutine called *reconstitute*. Given a string of characters $x_j \ldots x_n$, there is a smallest index $m \geq j$ such that the "translation" of $x_j \ldots x_n$ by ligatures and kerning has the form $y_1 \ldots y_t$ followed by the translation of $x_{m+1} \ldots x_n$, where $y_1 \ldots y_t$ is some nonempty sequence of character, ligature, and kern nodes. We call $x_j \ldots x_m$ a "cut prefix" of $x_j \ldots x_n$. For example, if $x_1x_2x_3 =$ `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 $x_{m+1} \ldots x_n$ until the entire translation has been found.

The *reconstitute* function returns the integer m and puts the nodes $y_1 \ldots y_t$ into a linked list starting at *link*(*hold_head*), getting the input $x_j \ldots x_n$ 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 }

909. ⟨ Declare the function called *reconstitute* 909 ⟩ ≡
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* ← 0; *t* ← *hold_head*; *w* ← 0; *link*(*hold_head*) ← *null*;
      { at this point *ligature_present* = *lft_hit* = *rt_hit* = *false* }
  ⟨ Set up data structures with the cursor following position *j* 911 ⟩;
continue: ⟨ If there's a ligature or kern at the cursor position, update the data structures, possibly advancing *j*; continue until the cursor moves 912 ⟩;
  ⟨ Append a ligature and/or kern to the translation; **goto** *continue* if the stack of inserted ligatures is nonempty 913 ⟩;
  reconstitute ← *j*;
  end;

This code is used in section 898.

910. The reconstitution procedure shares many of the global data structures by which TEX 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? }

911. **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* 911⟩ ≡
  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 909.

912. 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 912⟩ ≡

if *cur_l* = *non_char* **then**
  begin $k \leftarrow$ *bchar_label*[*hf*];
  if k = *non_address* **then goto** *done* **else** $q \leftarrow$ *font_info*[k].*qqqq*;
  end
else begin $q \leftarrow$ *char_info*(*hf*)(*cur_l*);
  if *char_tag*(q) ≠ *lig_tag* **then goto** *done*;
  $k \leftarrow$ *lig_kern_start*(*hf*)(q); $q \leftarrow$ *font_info*[k].*qqqq*;
  if *skip_byte*(q) > *stop_flag* **then**
    begin $k \leftarrow$ *lig_kern_restart*(*hf*)(q); $q \leftarrow$ *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* 914⟩;
        $w \leftarrow$ *char_kern*(*hf*)(q); **goto** *done*; { this kern will be inserted below }
        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;
  $k \leftarrow k + qo(skip\_byte(q)) + 1$; $q \leftarrow$ *font_info*[k].*qqqq*;
  end;
done:

This code is used in section 909.

913. **define** *wrap_lig*(#) $\equiv$
```
      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 913⟩ $\equiv$
```
  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 909.

914. ⟨ Carry out a ligature replacement, updating the cursor structure and possibly advancing j; **goto** *continue* if the cursor doesn't advance, otherwise **goto** *done* 914 ⟩ ≡

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 912.

915. 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 904 ⟩ +≡
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 }

916. When the following code is performed, *hyf* [0] and *hyf* [*hn*] will be zero.

⟨ Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 916 ⟩ ≡
repeat $l \gets j$; $j \gets$ *reconstitute*(*j*, *hn*, *bchar*, *qi*(*hyf_char*)) + 1;
  if *hyphen_passed* = 0 **then**
    begin *link*(*s*) ← *link*(*hold_head*);
    while *link*(*s*) > *null* **do** *s* ← *link*(*s*);
    if *odd*(*hyf* [*j* − 1]) **then**
      begin *l* ← *j*; *hyphen_passed* ← *j* − 1; *link*(*hold_head*) ← *null*;
      end;
    end;
  if *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 917 ⟩;
until *j* > *hn*;
link(*s*) ← *q*

This code is used in section 906.

917. 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 *advance_major_tail* ≡
    begin *major_tail* ← *link*(*major_tail*); *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 917 ⟩ ≡
repeat *r* ← *get_node*(*small_node_size*); *link*(*r*) ← *link*(*hold_head*); *type*(*r*) ← *disc_node*; *major_tail* ← *r*;
  r_count ← 0;
  while *link*(*major_tail*) > *null* **do** *advance_major_tail*;
  i ← *hyphen_passed*; *hyf* [*i*] ← 0; ⟨ Put the characters *hu*[*l* . . *i*] and a hyphen into *pre_break*(*r*) 918 ⟩;
  ⟨ Put the characters *hu*[*i* + 1 . .] into *post_break*(*r*), appending to this list and to *major_tail* until synchronization has been achieved 919 ⟩;
  ⟨ Move pointer *s* to the end of the current list, and set *replace_count*(*r*) appropriately 921 ⟩;
  hyphen_passed ← *j* − 1; *link*(*hold_head*) ← *null*;
until ¬*odd*(*hyf* [*j* − 1])

This code is used in section 916.

918. The new hyphen might combine with the previous character via ligature or kern. At this point we have $l-1 \le i < j$ and $i < hn$.

⟨Put the characters $hu[l\,..\,i]$ and a hyphen into $pre\_break(r)$ 918⟩ ≡

```
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(hold_head) > null then
    begin if minor_tail = null then pre_break(r) ← link(hold_head)
    else link(minor_tail) ← link(hold_head);
    minor_tail ← link(hold_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 917.

919. The synchronization algorithm begins with $l = i + 1 \le j$.

⟨Put the characters $hu[i+1\,..\,]$ into $post\_break(r)$, appending to this list and to $major\_tail$ until synchronization has been achieved 919⟩ ≡

```
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(hold_head) > null then
      begin if minor_tail = null then post_break(r) ← link(hold_head)
      else link(minor_tail) ← link(hold_head);
      minor_tail ← link(hold_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 920⟩;
  end
```

This code is used in section 917.

920. ⟨Append characters of $hu[j\,..\,]$ to $major\_tail$, advancing j 920⟩ ≡

```
begin j ← reconstitute(j, hn, bchar, non_char) + 1; link(major_tail) ← link(hold_head);
while link(major_tail) > null do advance_major_tail;
end
```

This code is used in section 919.

921. 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 921 ⟩ ≡
  if *r_count* > 127 **then** { we have to forget the discretionary hyphen }
    begin *link*(s) ← *link*(r); *link*(r) ← *null*; *flush_node_list*(r);
    end
  else begin *link*(s) ← r; *replace_count*(r) ← *r_count*;
    end;
  s ← *major_tail*

This code is used in section 917.

922. Hyphenation. When a word $hc[1\mathrel{..}hn]$ 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 $hc[0\mathrel{..}(hn+1)]$. Each pattern $p_1\ldots p_k$ of length k has an associated sequence of $k+1$ numbers $n_0\ldots n_k$; and if the pattern occurs in $hc[(j+1)\mathrel{..}(j+k)]$, TEX will set $hyf[j+i]\leftarrow\max(hyf[j+i],n_i)$ for $0\le i\le k$. 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 $p_1\ldots p_k$ and associated numbers $n_0\ldots n_k$ 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.

923. 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 $p_1\ldots p_k$ by letting z_0 be one greater than the relevant language index and then, for $1\le i\le k$, 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 $n_0\ldots n_k$ associated with $p_1\ldots p_k$.

The theory that comparatively few different number sequences $n_0\ldots n_k$ actually occur, since most of the n's are generally zero, seems to fail at least for the large German hyphenation patterns. Therefore the number sequences cannot any longer be encoded in such a way that $trie\_op(z_k)$ is only one byte long. We have introduced a new constant *max_trie_op* for the maximum allowable hyphenation operation code value; *max_trie_op* might be different for TEX and `INITEX` and must not exceed *max_halfword*. An opcode will occupy a halfword if *max_trie_op* exceeds *max_quarterword* or a quarterword otherwise. If $trie\_op(z_k)\ne min\_trie\_op$, when $p_1\ldots p_k$ has matched the letters in $hc[(l-k+1)\mathrel{..}l]$ 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 $v=min\_trie\_op$.

⟨Types in the outer block 18⟩ +≡
trie_pointer $=0\mathrel{..}ssup\_trie\_size$; { an index into *trie* }
trie_opcode $=0\mathrel{..}ssup\_trie\_opcode$; { a trie opcode }

924. For more than 255 trie op codes, the three fields *trie_link*, *trie_char*, and *trie_op* will no longer fit into one memory word; thus using web2c we define *trie* as three array instead of an array of records. The variant will be implented by reusing the opcode field later on with another macro.

define *trie_link*(#) ≡ *trie_trl*[#] { "downward" link in a trie }
define *trie_char*(#) ≡ *trie_trc*[#] { character matched at this trie location }
define *trie_op*(#) ≡ *trie_tro*[#] { program for hyphenation at this trie location }

⟨Global variables 13⟩ +≡
{ We will dynamically allocate these arrays. }
trie_trl: ↑*trie_pointer*; { *trie_link* }
trie_tro: ↑*trie_pointer*; { *trie_op* }
trie_trc: ↑*quarterword*; { *trie_char* }
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** *trie_opcode*; { continuation code }
op_start: **array** [*ASCII_code*] **of** 0 .. *trie_op_size*; { offset for current language }

925. ⟨Local variables for hyphenation 904⟩ +≡
z: *trie_pointer*; { an index into *trie* }
v: *integer*; { an index into *hyf_distance*, etc. }

926. 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** 926⟩ ≡
  for $j \leftarrow 0$ **to** *hn* **do** $hyf[j] \leftarrow 0$;
  ⟨Look for the word $hc[1\,..\,hn]$ in the exception table, and **goto** *found* (with *hyf* containing the hyphens) if an entry is found 933⟩;
  if $trie\_char(cur\_lang+1) \neq qi(cur\_lang)$ **then return**; { no patterns for *cur_lang* }
  $hc[0] \leftarrow 0$; $hc[hn+1] \leftarrow 0$; $hc[hn+2] \leftarrow 256$; { insert delimiters }
  for $j \leftarrow 0$ **to** $hn - r\_hyf + 1$ **do**
    begin $z \leftarrow trie\_link(cur\_lang+1) + hc[j]$; $l \leftarrow j$;
    while $hc[l] = qo(trie\_char(z))$ **do**
      begin if $trie\_op(z) \neq min\_trie\_op$ **then** ⟨Store maximum values in the *hyf* table 927⟩;
      $incr(l)$; $z \leftarrow trie\_link(z) + hc[l]$;
      end;
    end;
found: **for** $j \leftarrow 0$ **to** $l\_hyf - 1$ **do** $hyf[j] \leftarrow 0$;
  for $j \leftarrow 0$ **to** $r\_hyf - 1$ **do** $hyf[hn-j] \leftarrow 0$

This code is used in section 898.

927. ⟨Store maximum values in the *hyf* table 927⟩ ≡
  begin $v \leftarrow trie\_op(z)$;
  repeat $v \leftarrow v + op\_start[cur\_lang]$; $i \leftarrow l - hyf\_distance[v]$;
    if $hyf\_num[v] > hyf[i]$ **then** $hyf[i] \leftarrow hyf\_num[v]$;
    $v \leftarrow hyf\_next[v]$;
  until $v = min\_trie\_op$;
  end

This code is used in section 926.

928. 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, $h-1$, ..., until encountering the first word $\leq \alpha$. If this word is different from α, we can conclude that α is not in the table. This is a clever scheme which saves the need for a hash link array. However, it is difficult to increase the size of the hyphen exception arrays. To make this easier, the ordered hash has been replaced by a simple hash, using an additional array *hyph_link*. The value 0 in $hyph\_link[k]$ means that there are no more entries corresponding to the specific hash chain. When $hyph\_link[k] > 0$, the next entry in the hash chain is $hyph\_link[k] - 1$. This value is used because the arrays start at 0.

The words in the table point to lists in *mem* that specify hyphen positions in their *info* fields. The list for $c_1 \ldots c_n$ contains the number k if the word $c_1 \ldots c_n$ has a discretionary hyphen between c_k and c_{k+1}.

⟨Types in the outer block 18⟩ +≡
  $hyph\_pointer = 0\,..\,ssup\_hyph\_size$;
    { index into hyphen exceptions hash table; enlarging this requires changing (un)dump code }

929. ⟨ Global variables 13 ⟩ +≡
hyph_word: ↑*str_number*; { exception words }
hyph_list: ↑*pointer*; { lists of hyphen positions }
hyph_link: ↑*hyph_pointer*; { link array for hyphen exceptions hash table }
hyph_count: *integer*; { the number of words in the exception dictionary }
hyph_next: *integer*; { next free slot in hyphen exceptions hash table }

930. ⟨ Local variables for initialization 19 ⟩ +≡
z: *hyph_pointer*; { runs through the exception dictionary }

931. ⟨ Set initial values of key variables 21 ⟩ +≡
  for $z \leftarrow 0$ **to** *hyph_size* **do**
    begin *hyph_word*[z] ← 0; *hyph_list*[z] ← *null*; *hyph_link*[z] ← 0;
    end;
  hyph_count ← 0; *hyph_next* ← *hyph_prime* + 1;
  if *hyph_next* > *hyph_size* **then** *hyph_next* ← *hyph_prime*;

932. The algorithm for exception lookup is quite simple, as soon as we have a few more local variables to work with.

⟨ Local variables for hyphenation 904 ⟩ +≡
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* }

933. 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.

⟨ Look for the word *hc*[1 .. *hn*] in the exception table, and **goto** *found* (with *hyf* containing the hyphens) if an entry is found 933 ⟩ ≡
  $h \leftarrow hc[1]$; *incr*(*hn*); *hc*[*hn*] ← *cur_lang*;
  for $j \leftarrow 2$ **to** *hn* **do** $h \leftarrow (h + h + hc[j])$ **mod** *hyph_prime*;
  loop begin ⟨ 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* 934 ⟩;
    $h \leftarrow$ *hyph_link*[h];
    if $h = 0$ **then goto** *not_found*;
    decr(h);
    end;
not_found: *decr*(*hn*)

This code is used in section 926.

934. ⟨ If the string *hyph_word*[h] is less than $hc[1\mathrel{.\,.}hn]$, **goto** *not_found*; but if the two strings are equal, set *hyf* to the hyphen positions and **goto** *found* 934 ⟩ ≡

{ This is now a simple hash list, not an ordered one, so the module title is no longer descriptive. }

```
k ← hyph_word[h];
if k = 0 then goto not_found;
if length(k) = hn then
  begin j ← 1; u ← str_start[k];
  repeat if so(str_pool[u]) ≠ hc[j] then goto done;
    incr(j); incr(u);
  until j > hn;
  ⟨ Insert hyphens as specified in hyph_list[h] 935 ⟩;
  decr(hn); goto found;
  end;
done:
```

This code is used in section 933.

935. ⟨ Insert hyphens as specified in *hyph_list*[h] 935 ⟩ ≡

```
s ← hyph_list[h];
while s ≠ null do
  begin hyf[info(s)] ← 1; s ← link(s);
  end
```

This code is used in section 934.

936. ⟨ Search *hyph_list* for pointers to p 936 ⟩ ≡

```
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 172.

937. 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* $\leftarrow 0$
    else if *language* > 255 **then** *cur_lang* $\leftarrow 0$
    else *cur_lang* $\leftarrow$ *language*

procedure *new_hyph_exceptions*; { enters new exceptions }
  label *reswitch*, *exit*, *found*, *not_found*;
  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: *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;
  ⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then **return** 938 ⟩;
exit: **end**;

938. ⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then **return** 938 ⟩ ≡
  n ← 0; *p* ← *null*;
  loop begin *get_x_token*;
  reswitch: **case** *cur_cmd* **of**
    letter, *other_char*, *char_given*: ⟨ Append a new letter or hyphen 940 ⟩;
    char_num: **begin** *scan_char_num*; *cur_chr* ← *cur_val*; *cur_cmd* ← *char_given*; **goto** *reswitch*;
      end;
    pux_char_given: ⟨ Give improper hyphenation error for Chinese characters inside 1444 ⟩;
    pux_char_num: **begin** *scan_wchar_num*; *cur_chr* ← *cur_val*; *cur_cmd* ← *pux_char_given*;
      goto *reswitch*;
      end;
    spacer, *right_brace*: **begin if** $n > 1$ **then** ⟨ Enter a hyphenation exception 942 ⟩;
      if *cur_cmd* = *right_brace* **then return**;
      n ← 0; *p* ← *null*;
      end;
    othercases ⟨ Give improper `\hyphenation` error 939 ⟩
    endcases;
    end

This code is used in section 937.

939. ⟨ Give improper `\hyphenation` error 939 ⟩ ≡
  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 938.

940. ⟨Append a new letter or hyphen 940⟩ ≡
if *cur_chr* = "-" **then** ⟨Append the value n to list p 941⟩
else begin if *is_wchar*(*cur_chr*) **then**
begin *print_err*("Chinese␣character␣can´t␣appear␣here");
help2("Letters␣in␣\hyphenation␣words␣can´t␣be␣Chinese␣characters.")
("Proceed;␣I´ll␣ignore␣the␣character␣I␣just␣read."); *error*;
end
else if *lc_code*(*cur_chr*) = 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); $hc[n] \leftarrow$ *lc_code*(*cur_chr*);
end;
end

This code is used in section 938.

941. ⟨Append the value n to list p 941⟩ ≡
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 940.

942. ⟨Enter a hyphenation exception 942⟩ ≡
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_prime*; *append_char*($hc[j]$);
end;
$s \leftarrow$ *make_string*; ⟨Insert the pair (s,p) into the exception table 943⟩;
end

This code is used in section 938.

943. ⟨Insert the pair (s,p) into the exception table 943⟩ ≡
if *hyph_next* ≤ *hyph_prime* **then**
while (*hyph_next* > 0) ∧ (*hyph_word*[*hyph_next* − 1] > 0) **do** *decr*(*hyph_next*);
if (*hyph_count* = *hyph_size*) ∨ (*hyph_next* = 0) **then** *overflow*("exception␣dictionary", *hyph_size*);
incr(*hyph_count*);
while *hyph_word*[h] ≠ 0 **do**
begin ⟨If the string *hyph_word*[h] is less than or equal to s, interchange (*hyph_word*[h], *hyph_list*[h]) with (s,p) 944⟩;
if *hyph_link*[h] = 0 **then**
begin *hyph_link*[h] ← *hyph_next*;
if *hyph_next* ≥ *hyph_size* **then** *hyph_next* ← *hyph_prime*;
if *hyph_next* > *hyph_prime* **then** *incr*(*hyph_next*);
end;
$h \leftarrow$ *hyph_link*[h] − 1;
end;
found: *hyph_word*[h] ← s; *hyph_list*[h] ← p

This code is used in section 942.

944. ⟨If the string *hyph_word*[*h*] is less than or equal to *s*, interchange (*hyph_word*[*h*], *hyph_list*[*h*]) with (*s*, *p*) 944⟩ ≡

{ This is now a simple hash list, not an ordered one, so the module title is no longer descriptive. }
k ← *hyph_word*[*h*];
if *length*(*k*) ≠ *length*(*s*) **then goto** *not_found*;
u ← *str_start*[*k*]; *v* ← *str_start*[*s*];
repeat if *str_pool*[*u*] ≠ *str_pool*[*v*] **then goto** *not_found*;
incr(*u*); *incr*(*v*);
until *u* = *str_start*[*k* + 1]; { repeat hyphenation exception; flushing old data }
flush_string; *s* ← *hyph_word*[*h*]; { avoid *slow_make_string*! }
decr(*hyph_count*); { We could also *flush_list*(*hyph_list*[*h*]);, but it interferes with `trip.log`. }
goto *found*;
not_found:

This code is used in section 943.

945. 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 subtries. Finally the trie is packed into the efficient sequential form that the hyphenation algorithm actually uses.

⟨ Declare subprocedures for *line_break* 829 ⟩ +≡
init ⟨ Declare procedures for preprocessing hyphenation patterns 947 ⟩
tini

946. 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 $n_0 \ldots n_5 =$ 0 0 2 0 0 1 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 $hyf\_next[v'] = min\_trie\_op$.

TEX computes an appropriate value v with the *new_trie_op* subroutine below, by setting

$$v' \leftarrow new\_trie\_op(0, 1, min\_trie\_op), \qquad v \leftarrow new\_trie\_op(3, 2, v').$$

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** [*neg_trie_op_size* .. *trie_op_size*] **of** 0 .. *trie_op_size*;
{ trie op codes for quadruples }
trie_used: **array** [*ASCII_code*] **of** *trie_opcode*; { 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** *trie_opcode*; { opcode corresponding to a hashed quadruple }
trie_op_ptr: 0 .. *trie_op_size*; { number of stored ops so far }
tini
max_op_used: *trie_opcode*; { largest opcode used for any language }
small_op: *boolean*; { flag used while dumping or undumping }

947. It's tempting to remove the *overflow* stops in the following procedure; *new_trie_op* could return *min_trie_op* (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 947⟩ ≡
```
function new_trie_op(d, n : small_number; v : trie_opcode): trie_opcode;
  label exit;
  var h: neg_trie_op_size .. trie_op_size; { trial hash location }
    u: trie_opcode; { trial op code }
    l: 0 .. trie_op_size; { pointer to stored data }
  begin h ← abs(intcast(n) + 313 * intcast(d) + 361 * intcast(v) + 1009 * intcast(cur_lang)) mod
      (trie_op_size − neg_trie_op_size) + neg_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_trie_op then
        overflow("pattern␣memory␣ops␣per␣language", max_trie_op − min_trie_op);
      incr(trie_op_ptr); incr(u); trie_used[cur_lang] ← u;
      if u > max_op_used then max_op_used ← 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 951, 952, 956, 960, 962, 963, and 969.

This code is used in section 945.

948. 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 948⟩ ≡
```
  op_start[0] ← −min_trie_op;
  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 955.

949. 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) 164⟩ +≡
  for $k \leftarrow -trie\_op\_size$ **to** $trie\_op\_size$ **do** $trie\_op\_hash[k] \leftarrow 0$;
  for $k \leftarrow 0$ **to** 255 **do** $trie\_used[k] \leftarrow min\_trie\_op$;
  $max\_op\_used \leftarrow min\_trie\_op$; $trie\_op\_ptr \leftarrow 0$;

950. 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 $trie\_l[0]$, and the number of nodes is $trie\_ptr$. Null trie pointers are represented by zero. To initialize the trie, we simply set $trie\_l[0]$ and $trie\_ptr$ to zero. We also set $trie\_c[0]$ to some arbitrary value, since the algorithm may access it.

The algorithms maintain the condition

$$trie\_c[trie\_r[z]] > trie\_c[z] \qquad \text{whenever } z \neq 0 \text{ and } trie\_r[z] \neq 0;$$

in other words, sibling nodes are ordered by their c fields.

  define $trie\_root \equiv trie\_l[0]$ { root of the linked trie }

⟨Global variables 13⟩ +≡
  init $trie\_c$: ↑$packed\_ASCII\_code$; { characters to match }
  $trie\_o$: ↑$trie\_opcode$; { operations to perform }
  $trie\_l$: ↑$trie\_pointer$; { left subtrie links }
  $trie\_r$: ↑$trie\_pointer$; { right subtrie links }
  $trie\_ptr$: $trie\_pointer$; { the number of nodes in the trie }
  $trie\_hash$: ↑$trie\_pointer$; { used to identify equivalent subtries }
  tini

951. Let us suppose that a linked trie has already been constructed. Experience shows that we can often reduce its size by recognizing common subtries; 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 *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 subtries 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 947⟩ +≡

```
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(intcast(trie_c[p]) + 1009 ∗ intcast(trie_o[p]) +
        2718 ∗ intcast(trie_l[p]) + 3142 ∗ intcast(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;
```

952. 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 947⟩ +≡

```
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;
```

953. 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_tro*[#] { use the opcode field now for backward links }

⟨ Global variables 13 ⟩ +≡
init *trie_taken*: ↑*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

954. 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) 164 ⟩ +≡
trie_not_ready ← *true*;

955. 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 955 ⟩ ≡
⟨ Sort the hyphenation op tables into proper order 948 ⟩;
for $p \leftarrow 0$ **to** *trie_size* **do** *trie_hash*[p] ← 0;
trie_root ← *compress_trie*(*trie_root*); { identify equivalent subtries }
for $p \leftarrow 0$ **to** *trie_ptr* **do** *trie_ref*[p] ← 0;
for $p \leftarrow 0$ **to** 255 **do** *trie_min*[p] ← $p + 1$;
trie_link(0) ← 1; *trie_max* ← 0

This code is used in section 969.

956. 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 $c = trie\_c[p]$, 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 947⟩ +≡
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 $c \leftarrow so(trie\_c[p])$; $z \leftarrow trie\_min[c]$; { get the first conceivably good hole }
  loop begin $h \leftarrow z - c$;
    ⟨Ensure that $trie\_max \geq h + 256$ 957⟩;
    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* 958⟩;
  not_found: $z \leftarrow trie\_link(z)$; { move to the next hole }
    end;
found: ⟨Pack the family into *trie* relative to h 959⟩;
  end;

957. By making sure that *trie_max* is at least $h + 256$, we can be sure that $trie\_max > z$, since $h = z - c$. It follows that location *trie_max* will never be occupied in *trie*, and we will have $trie\_max \geq trie\_link(z)$.

⟨Ensure that $trie\_max \geq h + 256$ 957⟩ ≡
  if $trie\_max < h + 256$ **then**
    begin if $trie\_size \leq h + 256$ **then** *overflow*("pattern␣memory", *trie_size*);
    repeat *incr*(*trie_max*); $trie\_taken[trie\_max] \leftarrow false$; $trie\_link(trie\_max) \leftarrow trie\_max + 1$;
      $trie\_back(trie\_max) \leftarrow trie\_max - 1$;
    until $trie\_max = h + 256$;
    end

This code is used in section 956.

958. ⟨If all characters of the family fit relative to h, then **goto** *found*, otherwise **goto** *not_found* 958⟩ ≡
  $q \leftarrow trie\_r[p]$;
  while $q > 0$ **do**
    begin if $trie\_link(h + so(trie\_c[q])) = 0$ **then goto** *not_found*;
    $q \leftarrow trie\_r[q]$;
    end;
  goto *found*

This code is used in section 956.

959. ⟨ Pack the family into *trie* relative to h 959 ⟩ ≡

```
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 956.

960. To pack the entire linked trie, we use the following recursive procedure.

⟨ Declare procedures for preprocessing hyphenation patterns 947 ⟩ +≡

```
procedure trie_pack(p : trie_pointer); { pack subtries 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;
```

961. 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.

```
define clear_trie ≡ { clear trie[r] }
      begin trie_link(r) ← 0; trie_op(r) ← min_trie_op; trie_char(r) ← min_quarterword;
          { trie_char ← qi(0) }
      end
```

⟨ Move the data into *trie* 961 ⟩ ≡

```
if trie_root = 0 then { no patterns were given }
  begin for r ← 0 to 256 do clear_trie;
  trie_max ← 256;
  end
else begin 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); clear_trie; 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 969.

962. The fixing-up procedure is, of course, recursive. Since the linked trie usually has overlapping subtries, 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 947⟩ +≡
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 $z \leftarrow$ *trie_ref* [*p*];
  repeat $q \leftarrow$ *trie_l* [*p*]; $c \leftarrow$ *so*(*trie_c*[*p*]); *trie_link*$(z + c) \leftarrow$ *trie_ref* [*q*]; *trie_char*$(z + c) \leftarrow$ *qi*(*c*);
    trie_op$(z + c) \leftarrow$ *trie_o*[*p*];
    if $q > 0$ **then** *trie_fix*(*q*);
    $p \leftarrow$ *trie_r* [*p*];
  until $p = 0$;
  end;

963. 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 947⟩ +≡
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: *trie_opcode*; { 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 964⟩;
    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;

964. 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 964⟩ ≡
  $k \leftarrow 0$; $hyf[0] \leftarrow 0$; *digit_sensed* ← *false*;
  loop begin *get_x_token*;
    case *cur_cmd* **of**
    letter, *other_char*: ⟨Append a new letter or a hyphen level 965⟩;
    spacer, *right_brace*: **begin if** $k > 0$ **then** ⟨Insert a new pattern into the linked trie 966⟩;
      if *cur_cmd* = *right_brace* **then goto** *done*;
      $k \leftarrow 0$; $hyf[0] \leftarrow 0$; *digit_sensed* ← *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 963.

965. ⟨Append a new letter or a hyphen level 965⟩ ≡
  if *digit_sensed* ∨ (*cur_chr* < `"0"`) ∨ (*cur_chr* > `"9"`) **then**
    begin if *cur_chr* = `"."` **then** *cur_chr* ← 0 { edge-of-word delimiter }
    else begin *cur_chr* ← *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*] ← *cur_chr*; *hyf*[*k*] ← 0; *digit_sensed* ← *false*;
      end;
    end
  else if $k < 63$ **then**
      begin *hyf*[*k*] ← *cur_chr* − `"0"`; *digit_sensed* ← *true*;
      end

This code is used in section 964.

966. When the following code comes into play, the pattern $p_1 \ldots p_k$ appears in $hc[1 \mathrel{..} k]$, and the corresponding sequence of numbers $n_0 \ldots n_k$ appears in $hyf[0 \mathrel{..} k]$.

⟨Insert a new pattern into the linked trie 966⟩ ≡
  begin ⟨Compute the trie op code, v, and set $l \leftarrow 0$ 968⟩;
  $q \leftarrow 0$; $hc[0] \leftarrow cur\_lang$;
  while $l \le k$ **do**
    begin $c \leftarrow hc[l]$; $incr(l)$; $p \leftarrow trie\_l[q]$; $first\_child \leftarrow true$;
    while $(p > 0) \land (c > so(trie\_c[p]))$ **do**
      begin $q \leftarrow p$; $p \leftarrow trie\_r[q]$; $first\_child \leftarrow false$;
      end;
    if $(p = 0) \lor (c < so(trie\_c[p]))$ **then**
      ⟨Insert a new trie node between q and p, and make p point to it 967⟩;
    $q \leftarrow p$; { now node q represents $p_1 \ldots p_{l-1}$ }
    end;
  if $trie\_o[q] \ne min\_trie\_op$ **then**
    begin *print_err*("Duplicate␣pattern"); *help1*("(See␣Appendix␣H.)"); *error*;
    end;
  $trie\_o[q] \leftarrow v$;
  end

This code is used in section 964.

967. ⟨Insert a new trie node between q and p, and make p point to it 967⟩ ≡
  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** $trie\_r[q] \leftarrow p$;
  $trie\_c[p] \leftarrow si(c)$; $trie\_o[p] \leftarrow min\_trie\_op$;
  end

This code is used in section 966.

968. ⟨Compute the trie op code, v, and set $l \leftarrow 0$ 968⟩ ≡
  if $hc[1] = 0$ **then** $hyf[0] \leftarrow 0$;
  if $hc[k] = 0$ **then** $hyf[k] \leftarrow 0$;
  $l \leftarrow k$; $v \leftarrow min\_trie\_op$;
  loop begin if $hyf[l] \ne 0$ **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 966.

969. 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 947⟩ +≡
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* }
  begin ⟨Get ready to compress the trie 955⟩;
  if *trie_root* $\neq 0$ **then**
    begin *first_fit*(*trie_root*); *trie_pack*(*trie_root*);
    end;
  ⟨Move the data into *trie* 961⟩;
  trie_not_ready ← *false*;
  end;

970. 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.

971. 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.

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

function *prune_page_top*(*p* : *pointer*): *pointer*; { adjust top after page break }
  var *prev_p*: *pointer*; { lags one step behind *p* }
    q: *pointer*; { temporary variable for list manipulation }
  begin *prev_p* ← *temp_head*; *link*(*temp_head*) ← *p*;
  while $p \neq null$ **do**
    case *type*(*p*) **of**
    hlist_node, *vlist_node*, *rule_node*: ⟨ Insert glue for *split_top_skip* and set *p* ← *null* 972 ⟩;
    whatsit_node, *mark_node*, *ins_node*: **begin** *prev_p* ← *p*; *p* ← *link*(*prev_p*);
      end;
    glue_node, *kern_node*, *penalty_node*: **begin** *q* ← *p*; *p* ← *link*(*q*); *link*(*q*) ← *null*; *link*(*prev_p*) ← *p*;
      flush_node_list(*q*);
      end;
    othercases *confusion*("pruning")
    endcases;
  prune_page_top ← *link*(*temp_head*);
  end;

972. ⟨ Insert glue for *split_top_skip* and set *p* ← *null* 972 ⟩ ≡
  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 971.

973. 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 975 ⟩;
  prev_p ← *p*; *p* ← *link*(*prev_p*);
  end;
done: *vert_break* ← *best_place*;
 end;

974. 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 }

975. 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 975⟩ ≡
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 976⟩;
⟨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 977⟩;
if $(type(p) < glue\_node) \lor (type(p) > kern\_node)$ **then goto** *not_found*;
update_heights: ⟨Update the current height and depth measurements with respect to a glue or kern node p 979⟩;
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 973.

976. ⟨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 976⟩ ≡
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* 1368⟩;
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 975.

977. **define** *deplorable* $\equiv 100000$ { 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 977 ⟩ ≡
  if $pi < inf\_penalty$ **then**
    begin ⟨ Compute the badness, b, using *awful_bad* if the box is too full 978 ⟩;
    if $b < awful\_bad$ **then**
      if $pi \le eject\_penalty$ **then** $b \leftarrow pi$
      else if $b < inf\_bad$ **then** $b \leftarrow b + pi$
        else $b \leftarrow deplorable$;
    if $b \le least\_cost$ **then**
      begin $best\_place \leftarrow p$; $least\_cost \leftarrow b$; $best\_height\_plus\_depth \leftarrow cur\_height + prev\_dp$;
      end;
    if $(b = awful\_bad) \lor (pi \le eject\_penalty)$ **then goto** *done*;
    end

This code is used in section 975.

978. ⟨ Compute the badness, b, using *awful_bad* if the box is too full 978 ⟩ ≡
  if $cur\_height < h$ **then**
    if $(active\_height[3] \ne 0) \lor (active\_height[4] \ne 0) \lor (active\_height[5] \ne 0)$ **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 977.

979. 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 979 ⟩ ≡
  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 $(shrink\_order(q) \ne normal) \land (shrink(q) \ne 0)$ **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␣0pt␣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 975.

980. 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).

function *vsplit*(n : *eight_bits*; 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 $v \leftarrow$ *box*(n);
  if *split_first_mark* $\neq$ *null* **then**
    begin *delete_token_ref*(*split_first_mark*); *split_first_mark* $\leftarrow$ *null*; *delete_token_ref*(*split_bot_mark*);
    split_bot_mark $\leftarrow$ *null*;
    end;
  ⟨ Dispense with trivial cases of void or bad boxes 981 ⟩;
  $q \leftarrow$ *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 982 ⟩;
  $q \leftarrow$ *prune_page_top*(q); $p \leftarrow$ *list_ptr*(v); *free_node*(v, *box_node_size*);
  if q = *null* **then** *box*(n) $\leftarrow$ *null* { the *eq_level* of the box stays the same }
  else *box*(n) $\leftarrow$ *vpack*(q, *natural*);
  vsplit $\leftarrow$ *vpackage*(p, h, *exactly*, *split_max_depth*);
exit: **end**;

981. ⟨ Dispense with trivial cases of void or bad boxes 981 ⟩ ≡
  if v = *null* **then**
    begin *vsplit* $\leftarrow$ *null*; **return**;
    end;
  if *type*(v) $\neq$ *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* $\leftarrow$ *null*; **return**;
    end

This code is used in section 980.

982. 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 982⟩ ≡

```
p ← list_ptr(v);
if p = q then list_ptr(v) ← null
else loop begin if type(p) = mark_node then
      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 980.

983. 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* }

984. 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 $r_1 = \mathit{link}(\mathit{page\_ins\_head})$; node r_j is followed by $r_{j+1} = \mathit{link}(r_j)$; and if there are n items we have $r_{n+1} = \mathit{page\_ins\_head}$. 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 $\mathit{qi}(250)$ (the same as the *subtype* field of the relevant *ins_node*). These *subtype* fields are in increasing order, and $\mathit{subtype}(\mathit{page\_ins\_head}) = \mathit{qi}(255)$, 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 $\mathit{type}(r) = \mathit{inserting}$, then $\mathit{height}(r)$ contains the total of the height-plus-depth dimensions of the box and all its inserts seen so far. If $\mathit{type}(r) = \mathit{split\_up}$, 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; $\mathit{broken\_ptr}(r)$ points to the node where that insertion will be split, if TeX decides to split it, $\mathit{broken\_ins}(r)$ points to the insertion node that was tentatively split, and $\mathit{height}(r)$ includes also the natural height plus depth of the part that would be split off.

In both cases, $\mathit{last\_ins\_ptr}(r)$ points to the last *ins_node* encountered for box $\mathit{qo}(\mathit{subtype}(r))$ that would be at least partially inserted on the next page; and $\mathit{best\_ins\_ptr}(r)$ 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 $\mathit{best\_ins\_ptr}(r) = \mathit{null}$ 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 $\mathit{page\_ins\_node\_size} = 4$ { number of words for a page insertion node }
define $\mathit{inserting} = 0$ { an insertion class that has not yet overflowed }
define $\mathit{split\_up} = 1$ { an overflowed insertion class }
define $\mathit{broken\_ptr}(\#) \equiv \mathit{link}(\# + 1)$ { an insertion for this class will break here if anywhere }
define $\mathit{broken\_ins}(\#) \equiv \mathit{info}(\# + 1)$ { this insertion might break at *broken_ptr* }
define $\mathit{last\_ins\_ptr}(\#) \equiv \mathit{link}(\# + 2)$ { the most recent insertion for this *subtype* }
define $\mathit{best\_ins\_ptr}(\#) \equiv \mathit{info}(\# + 2)$ { the optimum most recent insertion }

⟨ Initialize the special list heads and constant nodes 793 ⟩ +≡
$\mathit{subtype}(\mathit{page\_ins\_head}) \leftarrow \mathit{qi}(255)$; $\mathit{type}(\mathit{page\_ins\_head}) \leftarrow \mathit{split\_up}$; $\mathit{link}(\mathit{page\_ins\_head}) \leftarrow \mathit{page\_ins\_head}$;

985. 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* and *last_kern* 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` }
insert_penalties: *integer*; { sum of the penalties for held-over insertions }

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

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

988. **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)("filll");
  if page_shrink ≠ 0 then
    begin print("␣minus␣"); print_scaled(page_shrink);
    end;
  end;
```

989. ⟨Show the status of the current page 989⟩ ≡

```
  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("total␣height␣"); 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 218.

990. 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;
```

991. 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 793⟩ +≡
type(*page_head*) ← *glue_node*; *subtype*(*page_head*) ← *normal*;

992. 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? }

993. ⟨Set initial values of key variables 21⟩ +≡
output_active ← *false*; *insert_penalties* ← 0;

994. 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 994⟩ ≡
page_contents ← *empty*; *page_tail* ← *page_head*; *link*(*page_head*) ← *null*;
last_glue ← *max_halfword*; *last_penalty* ← 0; *last_kern* ← 0; *page_depth* ← 0; *page_max_depth* ← 0

This code is used in section 1020.

995. 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;

996. 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;
```

997. TeX is not always in vertical mode at the time *build_page* is called; the current mode reflects what TeX 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 TeX's central control point.

```
  define contribute = 80 { go here to link a node into the current page }
⟨ Declare the procedure called fire_up 1015 ⟩
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 999 ⟩;
    ⟨ 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 1000 ⟩;
  until link(contrib_head) = null;
  ⟨ Make the contribution list empty by setting its tail to contrib_head 998 ⟩;
exit: end;
```

998. **define** *contrib_tail* ≡ *nest*[0].*tail_field* { tail of the contribution list }

```
⟨ Make the contribution list empty by setting its tail to contrib_head 998 ⟩ ≡
  if nest_ptr = 0 then tail ← contrib_head { vertical mode }
  else contrib_tail ← contrib_head { other modes }
```

This code is used in section 997.

999. ⟨ Update the values of *last_glue*, *last_penalty*, and *last_kern* 999 ⟩ ≡

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

This code is used in section 997.

1000. 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.

⟨ 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 1000 ⟩ ≡

⟨ 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 1003 ⟩;
⟨ 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* 1008 ⟩;
if (*type* (*p*) < *glue_node*) ∨ (*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* 1007 ⟩;
contribute: ⟨ Make sure that *page_max_depth* is not exceeded 1006 ⟩;
⟨ Link node *p* into the current page and **goto** *done* 1001 ⟩;
done1: ⟨ Recycle node *p* 1002 ⟩;
done:

This code is used in section 997.

1001. ⟨ Link node *p* into the current page and **goto** *done* 1001 ⟩ ≡

link (*page_tail*) ← *p*; *page_tail* ← *p*; *link* (*contrib_head*) ← *link* (*p*); *link* (*p*) ← *null*; **goto** *done*

This code is used in section 1000.

1002. ⟨ Recycle node *p* 1002 ⟩ ≡

link (*contrib_head*) ← *link* (*p*); *link* (*p*) ← *null*; *flush_node_list* (*p*)

This code is used in section 1000.

1003. 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 1003⟩ ≡

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* 1004⟩
else ⟨Prepare to move a box or rule node to the current page, then **goto** *contribute* 1005⟩;
whatsit_node: ⟨Prepare to move whatsit p to the current page, then **goto** *contribute* 1367⟩;
glue_node: **if** *page_contents* < *box_there* **then goto** *done1*
else if *precedes_break*(*page_tail*) **then** *pi* ← 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* ← 0
else goto *update_heights*;
penalty_node: **if** *page_contents* < *box_there* **then goto** *done1* **else** *pi* ← *penalty*(p);
mark_node: **goto** *contribute*;
ins_node: ⟨Append an insertion to the current page and **goto** *contribute* 1011⟩;
othercases *confusion*("page")
endcases

This code is used in section 1000.

1004. ⟨Initialize the current page, insert the `\topskip` glue ahead of p, and **goto** *continue* 1004⟩ ≡
begin if *page_contents* = *empty* **then** *freeze_page_specs*(*box_there*)
else *page_contents* ← *box_there*;
q ← *new_skip_param*(*top_skip_code*); { 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;
link(q) ← p; *link*(*contrib_head*) ← q; **goto** *continue*;
end

This code is used in section 1003.

1005. ⟨Prepare to move a box or rule node to the current page, then **goto** *contribute* 1005⟩ ≡
begin *page_total* ← *page_total* + *page_depth* + *height*(p); *page_depth* ← *depth*(p); **goto** *contribute*;
end

This code is used in section 1003.

1006. ⟨Make sure that *page_max_depth* is not exceeded 1006⟩ ≡
if *page_depth* > *page_max_depth* **then**
begin *page_total* ← *page_total* + *page_depth* − *page_max_depth*;
page_depth ← *page_max_depth*;
end;

This code is used in section 1000.

1007. ⟨ Update the current page measurements with respect to the glue or kern specified by node p 1007 ⟩ ≡

if *type*(*p*) = *kern_node* **then** $q \leftarrow p$
else begin $q \leftarrow$ *glue_ptr*(*p*);
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␣found␣on␣current␣page");
help4("The␣page␣about␣to␣be␣output␣contains␣some␣infinitely")
("shrinkable␣glue,␣e.g.,␣`\vss´␣or␣`\vskip␣0pt␣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*) ← *normal*; *delete_glue_ref*(*q*); *glue_ptr*(*p*) ← *r*; $q \leftarrow r$;
end;
end;
page_total ← *page_total* + *page_depth* + *width*(*q*); *page_depth* ← 0

This code is used in section 1000.

1008. ⟨ 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* 1008 ⟩ ≡

if *pi* < *inf_penalty* **then**
begin ⟨ Compute the badness, b, of the current page, using *awful_bad* if the box is too full 1010 ⟩;
if b < *awful_bad* **then**
if *pi* ≤ *eject_penalty* **then** $c \leftarrow$ *pi*
else if b < *inf_bad* **then** $c \leftarrow b$ + *pi* + *insert_penalties*
else $c \leftarrow$ *deplorable*
else $c \leftarrow b$;
if *insert_penalties* ≥ 10000 **then** $c \leftarrow$ *awful_bad*;
stat if *tracing_pages* > 0 **then** ⟨ Display the page break cost 1009 ⟩;
tats
if c ≤ *least_page_cost* **then**
begin *best_page_break* ← p; *best_size* ← *page_goal*; *least_page_cost* ← c; $r \leftarrow$ *link*(*page_ins_head*);
while r ≠ *page_ins_head* **do**
begin *best_ins_ptr*(*r*) ← *last_ins_ptr*(*r*); $r \leftarrow$ *link*(*r*);
end;
end;
if (c = *awful_bad*) ∨ (*pi* ≤ *eject_penalty*) **then**
begin *fire_up*(*p*); { output the current page at the best place }
if *output_active* **then 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 1000.

1009. ⟨Display the page break cost 1009⟩ ≡
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 $c \le least\_page\_cost$ **then** *print_char*("#");
end_diagnostic(*false*);
end

This code is used in section 1008.

1010. ⟨Compute the badness, b, of the current page, using *awful_bad* if the box is too full 1010⟩ ≡
if $page\_total < page\_goal$ **then**
  if $(page\_so\_far[3] \ne 0) \lor (page\_so\_far[4] \ne 0) \lor (page\_so\_far[5] \ne 0)$ **then** $b \gets 0$
  else $b \gets badness(page\_goal - page\_total, page\_so\_far[2])$
else if $page\_total - page\_goal > page\_shrink$ **then** $b \gets awful\_bad$
  else $b \gets badness(page\_total - page\_goal, page\_shrink)$

This code is used in section 1008.

1011. ⟨Append an insertion to the current page and **goto** *contribute* 1011⟩ ≡
begin if $page\_contents = empty$ **then** *freeze_page_specs*(*inserts_only*);
$n \gets subtype(p)$; $r \gets page\_ins\_head$;
while $n \ge subtype(link(r))$ **do** $r \gets link(r)$;
$n \gets qo(n)$;
if $subtype(r) \ne qi(n)$ **then** ⟨Create a page insertion node with $subtype(r) = qi(n)$, and include the glue correction for box n in the current page state 1012⟩;
if $type(r) = split\_up$ **then** $insert\_penalties \gets insert\_penalties + float\_cost(p)$
else begin $last\_ins\_ptr(r) \gets p$; $delta \gets page\_goal - page\_total - page\_depth + page\_shrink$;
  { this much room is left if we shrink the maximum }
  if $count(n) = 1000$ **then** $h \gets height(p)$
  else $h \gets x\_over\_n(height(p), 1000) * count(n)$; { this much room is needed }
  if $((h \le 0) \lor (h \le delta)) \land (height(p) + height(r) \le dimen(n))$ **then**
    begin $page\_goal \gets page\_goal - h$; $height(r) \gets height(r) + height(p)$;
    end
  else ⟨Find the best way to split the insertion, and change $type(r)$ to *split_up* 1013⟩;
  end;
goto *contribute*;
end

This code is used in section 1003.

1012. We take note of the value of `\skip` n and the height plus depth of `\box` n only when the first `\insert` n node is encountered for a new page. A user who changes the contents of `\box` n after that first `\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 1012⟩ ≡

```
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 1011.

1013. 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 `\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 `\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 `\vsplit` operation.

⟨Find the best way to split the insertion, and change $type(r)$ to *split_up* 1013⟩ ≡

```
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 1014⟩;
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 1011.

1014. ⟨Display the insertion split cost 1014⟩ ≡

```
begin begin_diagnostic; print_nl("%␣split"); print_int(n); print("␣to␣"); print_scaled(w);
print_char(","); print_scaled(best_height_plus_depth);
print("␣p=");
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 1013.

1015. 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* 1015⟩ ≡

```
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 1016⟩;
  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 1017⟩;
  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 1027⟩
    else ⟨Fire up the user's output routine and return 1028⟩;
  ⟨Perform the default output routine 1026⟩;
exit: end;
```

This code is used in section 997.

1016. ⟨ Set the value of *output_penalty* 1016 ⟩ ≡
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 1015.

1017. 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 1017 ⟩ ≡
if c = *best_page_break* **then** *best_page_break* ← *null*; { c not yet linked in }
⟨ Ensure that box 255 is empty before output 1018 ⟩;
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 1021 ⟩;
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 1023 ⟩;
end
else if *type*(p) = *mark_node* **then** ⟨ Update the values of *first_mark* and *bot_mark* 1019 ⟩;
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 1020 ⟩;
⟨ Delete the page-insertion nodes 1022 ⟩

This code is used in section 1015.

1018. ⟨ Ensure that box 255 is empty before output 1018 ⟩ ≡
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 1017.

1019. ⟨ Update the values of *first_mark* and *bot_mark* 1019 ⟩ ≡
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 1017.

1020. When the following code is executed, the current page runs from node *link*(*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 *link*(*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 $q = hold\_head$; and $p = null$ if and only if $prev\_p = page\_tail$. Error messages are suppressed within *vpackage*, since the box might appear to be overfull or underfull simply because the stretch and shrink from the `\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 1020⟩ ≡
  if $p \neq null$ **then**
    begin if *link*(*contrib_head*) = *null* **then**
      if *nest_ptr* = 0 **then** *tail* ← *page_tail*
      else *contrib_tail* ← *page_tail*;
    link(*page_tail*) ← *link*(*contrib_head*); *link*(*contrib_head*) ← p; *link*(*prev_p*) ← *null*;
    end;
  save_vbadness ← *vbadness*; *vbadness* ← *inf_bad*; *save_vfuzz* ← *vfuzz*; *vfuzz* ← *max_dimen*;
      { inhibit error messages }
  box(255) ← *vpackage*(*link*(*page_head*), *best_size*, *exactly*, *page_max_depth*); *vbadness* ← *save_vbadness*;
  vfuzz ← *save_vfuzz*;
  if *last_glue* ≠ *max_halfword* **then** *delete_glue_ref*(*last_glue*);
  ⟨Start a new current page 994⟩; { this sets *last_glue* ← *max_halfword* }
  if $q \neq hold\_head$ **then**
    begin *link*(*page_head*) ← *link*(*hold_head*); *page_tail* ← q;
    end

This code is used in section 1017.

1021. 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 1021⟩ ≡
  begin r ← *link*(*page_ins_head*);
  while $r \neq page\_ins\_head$ **do**
    begin if *best_ins_ptr*(r) ≠ *null* **then**
      begin n ← *qo*(*subtype*(r)); *ensure_vbox*(n);
      if *box*(n) = *null* **then** *box*(n) ← *new_null_box*;
      p ← *box*(n) + *list_offset*;
      while *link*(p) ≠ *null* **do** p ← *link*(p);
      last_ins_ptr(r) ← p;
      end;
    r ← *link*(r);
    end;
  end

This code is used in section 1017.

1022. ⟨Delete the page-insertion nodes 1022⟩ ≡
  r ← *link*(*page_ins_head*);
  while $r \neq page\_ins\_head$ **do**
    begin q ← *link*(r); *free_node*(r, *page_ins_node_size*); r ← q;
    end;
  link(*page_ins_head*) ← *page_ins_head*

This code is used in section 1017.

1023. We will set $best\_ins\_ptr \leftarrow null$ 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 1023⟩ ≡
begin $r \leftarrow link(page\_ins\_head)$;
while $subtype(r) \neq subtype(p)$ **do** $r \leftarrow link(r)$;
if $best\_ins\_ptr(r) = null$ **then** $wait \leftarrow true$
else begin $wait \leftarrow false$; $s \leftarrow last\_ins\_ptr(r)$; $link(s) \leftarrow ins\_ptr(p)$;
  if $best\_ins\_ptr(r) = p$ **then** ⟨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 1024⟩
  else begin while $link(s) \neq null$ **do** $s \leftarrow link(s)$;
    $last\_ins\_ptr(r) \leftarrow s$;
    end;
  end;
⟨Either append the insertion node p after node q, and remove it from the current page, or delete $node(p)$ 1025⟩;
end

This code is used in section 1017.

1024. ⟨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 1024⟩ ≡
begin if $type(r) = split\_up$ **then**
  if $(broken\_ins(r) = p) \land (broken\_ptr(r) \neq null)$ **then**
    begin while $link(s) \neq broken\_ptr(r)$ **do** $s \leftarrow link(s)$;
    $link(s) \leftarrow null$; $split\_top\_skip \leftarrow split\_top\_ptr(p)$; $ins\_ptr(p) \leftarrow prune\_page\_top(broken\_ptr(r))$;
    if $ins\_ptr(p) \neq null$ **then**
      begin $temp\_ptr \leftarrow vpack(ins\_ptr(p), natural)$; $height(p) \leftarrow height(temp\_ptr) + depth(temp\_ptr)$;
      $free\_node(temp\_ptr, box\_node\_size)$; $wait \leftarrow true$;
      end;
    end;
$best\_ins\_ptr(r) \leftarrow null$; $n \leftarrow qo(subtype(r))$; $temp\_ptr \leftarrow list\_ptr(box(n))$;
$free\_node(box(n), box\_node\_size)$; $box(n) \leftarrow vpack(temp\_ptr, natural)$;
end

This code is used in section 1023.

1025. ⟨Either append the insertion node p after node q, and remove it from the current page, or delete $node(p)$ 1025⟩ ≡
$link(prev\_p) \leftarrow link(p)$; $link(p) \leftarrow null$;
if $wait$ **then**
  begin $link(q) \leftarrow p$; $q \leftarrow p$; $incr(insert\_penalties)$;
  end
else begin $delete\_glue\_ref(split\_top\_ptr(p))$; $free\_node(p, ins\_node\_size)$;
  end;
$p \leftarrow prev\_p$

This code is used in section 1023.

1026. 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 1026⟩ ≡
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;
ship_out(*box*(255)); *box*(255) ← *null*;
end

This code is used in section 1015.

1027. ⟨Explain that too many dead cycles have occurred in a row 1027⟩ ≡
begin *print_err*("Output␣loop---"); *print_int*(*dead_cycles*); *print*("␣consecutive␣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 1015.

1028. ⟨Fire up the user's output routine and **return** 1028⟩ ≡
begin *output_active* ← *true*; *incr*(*dead_cycles*); *push_nest*; *mode* ← −*vmode*;
prev_depth ← *ignore_depth*; *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 1015.

1029. When the user's output routine finishes, it has constructed a vlist in internal vertical mode, and $\TeX$ will do the following:

⟨Resume the page builder after an output routine has come to an end 1029⟩ ≡
begin if (*loc* ≠ *null*) ∨ ((*token_type* ≠ *output_text*) ∧ (*token_type* ≠ *backed_up*)) **then**
⟨Recover from an unbalanced output routine 1030⟩;
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 1031⟩;
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;
pop_nest; *build_page*;
end

This code is used in section 1103.

1030. ⟨ Recover from an unbalanced output routine 1030 ⟩ ≡
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 $null = min\_halfword \le 0$ }

This code is used in section 1029.

1031. ⟨ Ensure that box 255 is empty after output 1031 ⟩ ≡
if $box(255) \neq null$ **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 1029.

1032. The chief executive. We come now to the *main_control* routine, which contains the master switch that causes all the various pieces of TEX 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 $abs(mode) + cur\_cmd$ 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 TEX'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 TEX's "inner loop"; the whole program runs efficiently when its inner loop is fast, so this part has been written with particular care.

1033. 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_wchar* = 130 { go here to typeset a string of consecutive double-byte characters }
define *save_cur_wchar* = 132 { go here to typeset a double-byte characters }
define *next_is_a_char* = 133 { go here if next token is a single-byte character }
define *fetch_next_tok* = 134 { go here to fetch next token }
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* 1046 ⟩
⟨ Declare the procedure called *handle_right_brace* 1071 ⟩
procedure *main_control*; { governs TEX's activities }
label *big_switch*, *reswitch*, *main_loop_wchar*, *main_loop_wchar* + 1, *save_cur_wchar*, *next_is_a_char*, *fetch_next_tok*, *main_loop*, *main_loop* + 1, *main_loop_wrapup*, *main_loop_lookahead* + 2, *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 }
begin if *every_job* ≠ *null* **then** *begin_token_list*(*every_job*, *every_job_text*);
⟨ Initialization of global variables done in the *main_control* procedure 1449 ⟩
big_switch: *get_x_token*;
reswitch: ⟨ Give diagnostic information, if requested 1034 ⟩;
case *abs*(*mode*) + *cur_cmd* **of**
hmode + *letter*, *hmode* + *other_char*, *hmode* + *char_given*: **if** *is_wchar*(*cur_chr*) **then**
goto *main_loop_wchar*
else goto *main_loop*;
hmode + *pux_char_given*: **goto** *main_loop_wchar*;
hmode + *char_num*: **begin** *scan_char_num*; *cur_chr* ← *cur_val*; **goto** *main_loop*; **end**;
hmode + *pux_char_num*: **begin** *scan_wchar_num*; *cur_chr* ← *cur_val*; **goto** *main_loop_wchar*; **end**;
hmode + *no_boundary*: **begin** *get_x_token*;
if (*cur_cmd* = *letter*) ∨ (*cur_cmd* = *other_char*) ∨ (*cur_cmd* = *char_given*) ∨ (*cur_cmd* = *char_num*) ∨ (*cur_cmd* = *pux_char_num*) ∨ (*cur_cmd* = *pux_char_given*) **then**
cancel_boundary ← *true*;
goto *reswitch*;
end;
⟨ Cases of *main_control* that handle spacer 1461 ⟩
⟨ Cases of *main_control* that are not part of the inner loop 1048 ⟩
end; { of the big **case** statement }
goto *big_switch*;
main_loop_wchar: ⟨ Append double-byte character *cur_chr* and the following double-byte characters (if any) to the current hlist in the current font; **goto** *main_loop* when a single-byte character has been fetched; **goto** *reswitch* when a non-character has been fetched 1453 ⟩;
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 1037 ⟩;
append_normal_space: ⟨ Append a normal inter-word space to the current list, then **goto** *big_switch* 1044 ⟩;
exit: **end**;

1034. When a new token has just been fetched at *big_switch*, we have an ideal place to monitor TEX's activity.

⟨ Give diagnostic information, if requested 1034 ⟩ ≡
  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 1033.

1035. 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*; { right 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? }

1036. 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*;

1037. 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 ≡
          if (cur_chr < 256) then main_s ← sf_code(cur_chr)
          else main_s ← 1000;
          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 1037⟩ ≡
  if ((head = tail) ∧ (mode > 0)) then
    begin if (insert_src_special_auto) then append_src_special;
    end;
  main_cf ← cur_cfont;   { in case the first letter is not a Chinese character }
  ⟨If the preceding node is wchar node, then append a cespace 1451⟩;
main_loop + 1: adjust_space_factor; 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;
  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 1038⟩;
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 1039⟩;
main_loop_lookahead: ⟨Look ahead for another character, or leave lig_stack empty if there's none there 1041⟩;
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 1042⟩;
main_loop_move_lig: ⟨Move the cursor past a pseudo-ligature, then goto main_loop_lookahead or
        main_lig_loop 1040⟩
```

This code is used in section 1033.

1038. 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;
      link(cur_q) ← main_p; tail ← main_p; ligature_present ← false;
      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
```

⟨Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1038⟩ ≡

```
  wrapup(rt_hit)
```

This code is used in section 1037.

1039. ⟨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* 1039⟩ ≡

```
  if lig_stack = null then goto reswitch;
  cur_q ← tail; cur_l ← character(lig_stack);
main_loop_move + 1: if ¬is_char_node(lig_stack) then goto main_loop_move_lig;
main_loop_move + 2: if (qo(effective_char(false, main_f,
      qi(cur_chr))) > font_ec[main_f]) ∨ (qo(effective_char(false, main_f, qi(cur_chr))) < font_bc[main_f])
      then
    begin char_warning(main_f, cur_chr); free_avail(lig_stack); goto big_switch;
    end;
  main_i ← effective_char_info(main_f, cur_l);
  if ¬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 1037.

1040. 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* 1040 ⟩ ≡
  $main\_p \leftarrow lig\_ptr(lig\_stack)$;
  if $main\_p > null$ **then** $tail\_append(main\_p)$; { append a single character }
  $temp\_ptr \leftarrow lig\_stack$; $lig\_stack \leftarrow link(temp\_ptr)$; $free\_node(temp\_ptr, small\_node\_size)$;
  $main\_i \leftarrow char\_info(main\_f)(cur\_l)$; $ligature\_present \leftarrow true$;
  if $lig\_stack = null$ **then**
    if $main\_p > null$ **then goto** *main_loop_lookahead*
    else $cur\_r \leftarrow bchar$
  else $cur\_r \leftarrow character(lig\_stack)$;
  goto *main_lig_loop*

This code is used in section 1037.

1041. 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 1041 ⟩ ≡
  ⟨ Look ahead for next character. If it is a wide character then append a cespace, or leave *lig_stack* empty if there's no character there 1460 ⟩

This code is used in section 1037.

1042. 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 $main\_i = char\_info(main\_f)(cur\_l)$.

⟨ If there's a ligature/kern command relevant to *cur_l* and *cur_r*, adjust the text appropriately; exit to *main_loop_wrapup* 1042 ⟩ ≡
  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* 1043 ⟩;
  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 1037.

1043. 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 1043⟩ ≡
  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 1042.

1044. 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 1044⟩ ≡
  if space_skip = zero_glue then
    begin ⟨Find the glue specification, main_p, for text spaces in the current font 1045⟩;
    temp_ptr ← new_glue(main_p);
    end
  else temp_ptr ← new_param_glue(space_skip_code);
  link(tail) ← temp_ptr; tail ← temp_ptr;
  if pux_xspace = 0 then goto reswitch
  else goto big_switch
```

This code is used in section 1033.

1045. 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 1045⟩ ≡
  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 1044 and 1046.

1046. ⟨Declare action procedures for use by *main_control* 1046⟩ ≡

```
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 1045⟩;
    main_p ← new_spec(main_p);
    ⟨Modify the glue specification in main_p according to the space factor 1047⟩;
    q ← new_glue(main_p); glue_ref_count(main_p) ← null;
    end;
  link(tail) ← q; tail ← q;
  end;
```

See also sections 1050, 1052, 1053, 1054, 1057, 1063, 1064, 1067, 1072, 1073, 1078, 1082, 1087, 1089, 1094, 1096, 1098, 1099, 1102, 1104, 1106, 1108, 1113, 1116, 1120, 1122, 1126, 1130, 1132, 1134, 1138, 1139, 1141, 1145, 1154, 1158, 1162, 1163, 1166, 1168, 1175, 1177, 1179, 1184, 1194, 1197, 1203, 1214, 1273, 1278, 1282, 1291, 1296, 1305, 1351, 1379, and 1406.

This code is used in section 1033.

1047. ⟨ Modify the glue specification in *main_p* according to the space factor 1047 ⟩ ≡
if *space_factor* $\geq$ 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 1046.

1048. 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 1048 ⟩ ≡
any_mode(*relax*), *vmode* + *spacer*, *mmode* + *spacer*, *mmode* + *no_boundary*: *do_nothing*;
any_mode(*ignore_spaces*): **begin** ⟨ Get the next non-blank non-call token 409 ⟩;
goto *reswitch*;
end;
vmode + *stop*: **if** *its_all_over* **then return**; { this is the only way out }
⟨ Forbidden cases detected in *main_control* 1051 ⟩ *any_mode*(*mac_param*): *report_illegal_case*;
⟨ Math-only cases in non-math modes, or vice versa 1049 ⟩: *insert_dollar_sign*;
⟨ Cases of *main_control* that build boxes and lists 1059 ⟩
⟨ Cases of *main_control* that don't depend on *mode* 1213 ⟩
⟨ Cases of *main_control* that are for extensions to TeX 1350 ⟩

This code is used in section 1033.

1049. 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 1049 ⟩ ≡
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*(*vcenter*),
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 1048.

1050. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
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;

1051. 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* 1051 ⟩ ≡
vmode + *vmove*, *hmode* + *hmove*, *mmode* + *hmove*, *any_mode*(*last_item*),

See also sections 1101, 1114, and 1147.

This code is used in section 1048.

1052. 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* 1046⟩ +≡
procedure *you_cant*;
  begin *print_err*("You␣can´t␣use␣`"); *print_cmd_chr*(*cur_cmd*, *cur_chr*); *print_in_mode*(*mode*);
  end;

1053. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
procedure *report_illegal_case*;
  begin *you_cant*; *help4*("Sorry,␣but␣I´m␣not␣programmed␣to␣handle␣this␣case;")
  ("I´ll␣just␣pretend␣that␣you␣didn´t␣ask␣for␣it.")
  ("If␣you´re␣in␣the␣wrong␣mode,␣you␣might␣be␣able␣to")
  ("return␣to␣the␣right␣one␣by␣typing␣`I}´␣or␣`I$´␣or␣`I\par´.");
  error;
  end;

1054. 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* 1046⟩ +≡
function *privileged*: *boolean*;
  begin if $mode > 0$ **then** *privileged* ← *true*
  else begin *report_illegal_case*; *privileged* ← *false*;
    end;
  end;

1055. 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 226⟩ +≡
  primitive("end", *stop*, 0);
  primitive("dump", *stop*, 1);

1056. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
stop: **if** $chr\_code = 1$ **then** *print_esc*("dump") **else** *print_esc*("end");

1057. 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* 1046 ⟩ +≡
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**;

1058. 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.

1059. 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* ← *ignore_depth*.

⟨Cases of *main_control* that build boxes and lists 1059⟩ ≡
vmode + *hrule*, *hmode* + *vrule*, *mmode* + *vrule*: **begin** *tail_append*(*scan_rule_spec*);
  if *abs*(*mode*) = *vmode* **then** *prev_depth* ← *ignore_depth*
  else if *abs*(*mode*) = *hmode* **then** *space_factor* ← 1000;
  end;

See also sections 1060, 1066, 1070, 1076, 1093, 1095, 1097, 1100, 1105, 1107, 1112, 1115, 1119, 1125, 1129, 1133, 1137, 1140, 1143, 1153, 1157, 1161, 1165, 1167, 1170, 1174, 1178, 1183, 1193, 1196, and 1445.

This code is used in section 1048.

1060. 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 1059⟩ +≡
vmode + *vskip*, *hmode* + *hskip*, *mmode* + *hskip*, *mmode* + *mskip*: *append_glue*;
any_mode(*kern*), *mmode* + *mkern*: *append_kern*;

1061. 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 226⟩ +≡
  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*);

1062. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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");

1063. 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* 1046 ⟩ +≡
procedure *append_glue*;
  var *s*: *small_number*; { modifier of skip command }
  begin $s \leftarrow$ *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 $s \geq$ *skip_code* **then**
    begin *decr*(*glue_ref_count*(*cur_val*));
    if $s >$ *skip_code* **then** *subtype*(*tail*) ← *mu_glue*;
    end;
  end;

1064. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *append_kern*;
  var *s*: *quarterword*; { *subtype* of the kern node }
  begin $s \leftarrow$ *cur_chr*; *scan_dimen*($s =$ *mu_glue*, *false*, *false*); *tail_append*(*new_kern*(*cur_val*));
  subtype(*tail*) ← *s*;
  end;

1065. 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*; TEX 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.

1066. 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 1059⟩ +≡
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*;

1067. 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. TEX 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* 1046⟩ +≡
procedure *off_save*;
  var *p*: *pointer*; { inserted token }
  begin if *cur_group* = *bottom_level* **then** ⟨Drop current token and complain that it was unmatched 1069⟩
  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 1068⟩;
    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;

1068. 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 1068 ⟩ ≡
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 1067.

1069. ⟨ Drop current token and complain that it was unmatched 1069 ⟩ ≡
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 1067.

1070. 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 1059 ⟩ +≡
any_mode(*right_brace*): *handle_right_brace*;

1071. ⟨ Declare the procedure called *handle_right_brace* 1071 ⟩ ≡
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: *unsave*;
bottom_level: **begin** *print_err*("Too␣many␣}´s");
help2("You´ve␣closed␣more␣groups␣than␣you␣opened.")
("Such␣booboos␣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 1088 ⟩
othercases *confusion*("rightbrace")
endcases;
end;

This code is used in section 1033.

1072. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
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␣`I$}´."); *error*; *incr*(*align_state*);
  end;

1073. Here is where we clear the parameters that are supposed to revert to their default values after every paragraph and when internal vertical mode is entered.

⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *normal_paragraph*;
  begin if *looseness* $\neq 0$ **then** *eq_word_define*(*int_base* + *looseness_code*, 0);
  if *hang_indent* $\neq 0$ **then** *eq_word_define*(*dimen_base* + *hang_indent_code*, 0);
  if *hang_after* $\neq 1$ **then** *eq_word_define*(*int_base* + *hang_after_code*, 1);
  if *par_shape_ptr* $\neq$ *null* **then** *eq_define*(*par_shape_loc*, *shape_ref*, *null*);
  end;

1074. 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 $\geq$ *box_flag*, where *box_flag* $= 2^{30}$. Codes *box_flag* through *box_flag* $+ 255$ represent '`\setbox0`' through '`\setbox255`'; codes *box_flag* $+ 256$ through *box_flag* $+ 511$ represent '`\global\setbox0`' through '`\global\setbox255`'; code *box_flag* $+ 512$ represents '`\shipout`'; and codes *box_flag* $+ 513$ through *box_flag* $+ 515$ 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 denote `\vbox` and `\hbox`, respectively.

define *box_flag* ≡ ´10000000000 { context code for '`\setbox0`' }
define *ship_out_flag* ≡ *box_flag* + 512 { context code for '`\shipout`' }
define *leader_flag* ≡ *box_flag* + 513 { context code for '`\leaders`' }
define *box_code* = 0 { *chr_code* for '`\box`' }
define *copy_code* = 1 { *chr_code* for '`\copy`' }
define *last_box_code* = 2 { *chr_code* for '`\lastbox`' }
define *vsplit_code* = 3 { *chr_code* for '`\vsplit`' }
define *vtop_code* = 4 { *chr_code* for '`\vtop`' }

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

1075. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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");

1076. 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 1059 ⟩ +≡
vmode + *hmove*, *hmode* + *vmove*, *mmode* + *vmove*: **begin** $t \leftarrow$ *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);

1077. 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 }

1078. 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* 1046 ⟩ +≡
procedure *box_end*(*box_context* : *integer*);
  var *p*: *pointer*; { *ord_noad* for new box in math mode }
  begin if *box_context* < *box_flag* **then**
    ⟨ Append box *cur_box* to the current list, shifted by *box_context* 1079 ⟩
  else if *box_context* < *ship_out_flag* **then** ⟨ Store *cur_box* in a box register 1080 ⟩
    else if *cur_box* ≠ *null* **then**
        if *box_context* > *ship_out_flag* **then** ⟨ Append a new leader node that uses *cur_box* 1081 ⟩
        else *ship_out*(*cur_box*);
  end;

1079. 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* 1079 ⟩ ≡

```
begin if cur_box ≠ null then
  begin shift_amount(cur_box) ← box_context;
  if abs(mode) = vmode then
    begin append_to_vlist(cur_box);
    if adjust_tail ≠ null then
      begin if adjust_head ≠ adjust_tail then
        begin link(tail) ← link(adjust_head); tail ← adjust_tail;
        end;
      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 1078.

1080. ⟨ Store *cur_box* in a box register 1080 ⟩ ≡

```
if box_context < box_flag + 256 then eq_define(box_base − box_flag + box_context, box_ref, cur_box)
else geq_define(box_base − box_flag − 256 + box_context, box_ref, cur_box)
```

This code is used in section 1078.

1081. ⟨ Append a new leader node that uses *cur_box* 1081 ⟩ ≡

```
begin ⟨ Get the next non-blank non-relax non-call token 407 ⟩;
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 1078.

1082. 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* 1046 ⟩ +≡
procedure *begin_box*(*box_context* : *integer*);
  label *exit*, *done*;
  var *p*, *q*: *pointer*; { run through the current list }
    m: *quarterword*; { the length of a replacement list }
    k: *halfword*; { 0 or *vmode* or *hmode* }
    n: *eight_bits*; { a box number }
  begin case *cur_chr* **of**
  box_code: **begin** *scan_eight_bit_int*; *cur_box* ← *box*(*cur_val*); *box*(*cur_val*) ← *null*;
      { the box becomes void, at the same level }
    end;
  copy_code: **begin** *scan_eight_bit_int*; *cur_box* ← *copy_node_list*(*box*(*cur_val*));
    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* 1083 ⟩;
  vsplit_code: ⟨ Split off part of a vertical box, make *cur_box* point to it 1085 ⟩;
  othercases ⟨ Initiate the construction of an hbox or vbox, then **return** 1086 ⟩
  endcases;
  box_end(*box_context*); { in simple cases, we use the box immediately }
exit: **end**;

1083. Note that the condition ¬*is_char_node*(*tail*) implies that *head* ≠ *tail*, since *head* is a one-word node.

⟨ 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* 1083 ⟩ ≡
  begin *cur_box* ← *null*;
  if *abs*(*mode*) = *mmode* **then**
    begin *you_cant*; *help1*("Sorry;␣this␣\lastbox␣will␣be␣void."); *error*;
    end
  else if (*mode* = *vmode*) ∧ (*head* = *tail*) **then**
      begin *you_cant*; *help2*("Sorry...I␣usually␣can´t␣take␣things␣from␣the␣current␣page.")
      ("This␣\lastbox␣will␣therefore␣be␣void."); *error*;
      end
    else begin if ¬*is_char_node*(*tail*) **then**
        if (*type*(*tail*) = *hlist_node*) ∨ (*type*(*tail*) = *vlist_node*) **then**
          ⟨ Remove the last box, unless it's part of a discretionary 1084 ⟩;
      end;
  end

This code is used in section 1082.

1084. ⟨Remove the last box, unless it's part of a discretionary 1084⟩ ≡

begin $q \leftarrow head$;
repeat $p \leftarrow q$;
  if $\neg is\_char\_node(q)$ **then**
    if $type(q) = disc\_node$ **then**
      begin for $m \leftarrow 1$ **to** $replace\_count(q)$ **do** $p \leftarrow link(p)$;
      if $p = tail$ **then goto** $done$;
      end;
  $q \leftarrow link(p)$;
until $q = tail$;
$cur\_box \leftarrow tail$; $shift\_amount(cur\_box) \leftarrow 0$; $tail \leftarrow p$; $link(p) \leftarrow null$;
$done$: **end**

This code is used in section 1083.

1085. Here we deal with things like '`\vsplit 13 to 100pt`'.

⟨Split off part of a vertical box, make cur_box point to it 1085⟩ ≡

begin $scan\_eight\_bit\_int$; $n \leftarrow cur\_val$;
if $\neg scan\_keyword($"`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;
$scan\_normal\_dimen$; $cur\_box \leftarrow vsplit(n, cur\_val)$;
end

This code is used in section 1082.

1086. 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** 1086⟩ ≡

begin $k \leftarrow cur\_chr - vtop\_code$; $saved(0) \leftarrow box\_context$;
if $k = hmode$ **then**
  if $(box\_context < box\_flag) \land (abs(mode) = vmode)$ **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 $scan\_spec(vtop\_group, true)$; $k \leftarrow vmode$;
    end;
  $normal\_paragraph$;
  end;
$push\_nest$; $mode \leftarrow -k$;
if $k = vmode$ **then**
  begin $prev\_depth \leftarrow ignore\_depth$;
  if $every\_vbox \neq null$ **then** $begin\_token\_list(every\_vbox, every\_vbox\_text)$;
  end
else begin $space\_factor \leftarrow 1000$;
  if $every\_hbox \neq null$ **then** $begin\_token\_list(every\_hbox, every\_hbox\_text)$;
  end;
return;
end

This code is used in section 1082.

1087. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
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 407 ⟩;
  if *cur_cmd* = *make_box* **then** *begin_box*(*box_context*)
  else if (*box_context* $\geq$ *leader_flag*) $\land$ ((*cur_cmd* = *hrule*) $\lor$ (*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;

1088. 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 1088 ⟩ ≡
hbox_group: ⟨ Setup *hbox_tail* and package 1462 ⟩;
adjusted_hbox_group: **begin** *adjust_tail* ← *adjust_head*; ⟨ Setup *hbox_tail* and package 1462 ⟩;
  end;
vbox_group: **begin** *end_graf*; *package*(0);
  end;
vtop_group: **begin** *end_graf*; *package*(*vtop_code*);
  end;

See also sections 1103, 1121, 1135, 1136, 1171, 1176, and 1189.

This code is used in section 1071.

1089. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
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*; *unsave*; *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` 1090 ⟩;
    end;
  pop_nest; *box_end*(*saved*(0));
  end;

1090. 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` 1090 ⟩ ≡
  begin *h* ← 0; *p* ← *list_ptr*(*cur_box*);
  if *p* $\neq$ *null* **then**
    if *type*(*p*) $\leq$ *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 1089.

1091. A paragraph begins when horizontal-mode material occurs in vertical mode, or when the paragraph is explicitly started by '\indent' or '\noindent'.

⟨ Put each of TeX's primitives into the hash table 226 ⟩ +≡
primitive("indent", *start_par*, 1); *primitive*("noindent", *start_par*, 0);

1092. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
start_par: **if** *chr_code* = 0 **then** *print_esc*("noindent") **else** *print_esc*("indent");

1093. ⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
vmode + *start_par*: *new_graf*(*cur_chr* > 0);
vmode + *letter*, *vmode* + *other_char*, *vmode* + *char_num*, *vmode* + *char_given*, *vmode* + *pux_char_num*, *vmode* + *pux_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;

1094. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
function *norm_min*(*h* : *integer*): *small_number*;
begin if $h \le 0$ **then** *norm_min* ← 1 **else if** $h \ge 63$ **then** *norm_min* ← 63 **else** *norm_min* ← *h*;
end;

procedure *new_graf*(*indented* : *boolean*);
begin *prev_graf* ← 0;
if (*mode* = *vmode*) ∨ (*head* ≠ *tail*) **then** *tail_append*(*new_param_glue*(*par_skip_code*));
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*;
if *indented* **then**
begin *tail* ← *new_null_box*; *link*(*head*) ← *tail*; *width*(*tail*) ← *par_indent*;
if (*insert_src_special_every_par*) **then** *insert_src_special*;
end;
if *every_par* ≠ *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;

1095. ⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
hmode + *start_par*, *mmode* + *start_par*: *indent_in_hmode*;

1096. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *indent_in_hmode*;
var *p*, *q*: *pointer*;
begin if *cur_chr* > 0 **then** { \indent }
begin *p* ← *new_null_box*; *width*(*p*) ← *par_indent*;
if *abs*(*mode*) = *hmode* **then** *space_factor* ← 1000
else begin *q* ← *new_noad*; *math_type*(*nucleus*(*q*)) ← *sub_box*; *info*(*nucleus*(*q*)) ← *p*; *p* ← *q*;
end;
tail_append(*p*);
end;
end;

1097. 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`.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
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*;

1098. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
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;

1099. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *end_graf*;
  begin if *mode* = *hmode* **then**
    begin if *head* = *tail* **then** *pop_nest* { null paragraphs are ignored }
    else *line_break*(*widow_penalty*);
    normal_paragraph; *error_count* ← 0;
    end;
  end;

1100. Insertion and adjustment and mark nodes are constructed by the following pieces of the program.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
any_mode(*insert*), *hmode* + *vadjust*, *mmode* + *vadjust*: *begin_insert_or_adjust*;
any_mode(*mark*): *make_mark*;

1101. ⟨ Forbidden cases detected in *main_control* 1051 ⟩ +≡
  vmode + *vadjust*,

1102. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

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*; *incr*(*save_ptr*); *new_save_level*(*insert_group*); *scan_left_brace*; *normal_paragraph*;
  push_nest; *mode* ← −*vmode*; *prev_depth* ← *ignore_depth*;
  end;

1103. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1088 ⟩ +≡

insert_group: **begin** *end_graf*; *q* ← *split_top_skip*; *add_glue_ref*(*q*); *d* ← *split_max_depth*;
  f ← *floating_penalty*; *unsave*; *decr*(*save_ptr*);
    { 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*;
    subtype(*tail*) ← 0; { the *subtype* is not used }
    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 1029 ⟩;

1104. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

procedure *make_mark*;
  var *p*: *pointer*; { new node }
  begin *p* ← *scan_toks*(*false*, *true*); *p* ← *get_node*(*small_node_size*); *type*(*p*) ← *mark_node*;
  subtype(*p*) ← 0; { the *subtype* is not used }
  mark_ptr(*p*) ← *def_ref*; *link*(*tail*) ← *p*; *tail* ← *p*;
  end;

1105. Penalty nodes get into a list via the *break_penalty* command.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
any_mode(*break_penalty*): *append_penalty*;

1106. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

procedure *append_penalty*;
  begin *scan_int*; *tail_append*(*new_penalty*(*cur_val*));
  if *mode* = *vmode* **then** *build_page*;
  end;

1107. 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 1059⟩ +≡
any_mode(*remove_item*): *delete_last*;

1108. 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* 1046⟩ +≡

```
procedure delete_last;
  label exit;
  var p, q: pointer;  { run through the current list }
    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 1109⟩
  else begin if ¬is_char_node(tail) then
      if type(tail) = cur_chr then
        begin q ← head;
        repeat p ← q;
          if ¬is_char_node(q) then
            if type(q) = disc_node then
              begin for m ← 1 to replace_count(q) do p ← link(p);
              if p = tail then return;
              end;
          q ← link(p);
        until q = tail;
        link(p) ← null; flush_node_list(tail); tail ← p;
        end;
    end;
exit: end;
```

1109. ⟨Apologize for inability to do the operation now, unless `\unskip` follows non-glue 1109⟩ ≡

```
  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␣`I\vskip-\lastskip´␣instead.");
    if cur_chr = kern_node then help_line[0] ← ("Try␣`I\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 1108.

1110. ⟨Put each of TEX's primitives into the hash table 226⟩ +≡

```
  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);
```

1111. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
remove_item: **if** *chr_code* = *glue_node* **then** *print_esc*("unskip")
  else if *chr_code* = *kern_node* **then** *print_esc*("unkern")
    else *print_esc*("unpenalty");
un_hbox: **if** *chr_code* = *copy_code* **then** *print_esc*("unhcopy")
  else *print_esc*("unhbox");
un_vbox: **if** *chr_code* = *copy_code* **then** *print_esc*("unvcopy")
  else *print_esc*("unvbox");

1112. The *un_hbox* and *un_vbox* commands unwrap one of the 256 current boxes.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
vmode + *un_vbox*, *hmode* + *un_hbox*, *mmode* + *un_hbox*: *unpackage*;

1113. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *unpackage*;
  label *exit*;
  var *p*: *pointer*; { the box }
    c: *box_code* . . *copy_code*; { should we copy? }
  begin *c* ← *cur_chr*; *scan_eight_bit_int*; *p* ← *box*(*cur_val*);
  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*); *box*(*cur_val*) ← *null*; *free_node*(*p*, *box_node_size*);
    end;
  while *link*(*tail*) ≠ *null* **do** *tail* ← *link*(*tail*);
exit: **end**;

1114. ⟨ Forbidden cases detected in *main_control* 1051 ⟩ +≡
  vmode + *ital_corr*,

1115. Italic corrections are converted to kern nodes when the *ital_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.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
hmode + *ital_corr*: *append_italic_correction*;
mmode + *ital_corr*: *tail_append*(*new_kern*(0));

1116. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
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* $\neq$ *head* **then**
    begin if *is_char_node*(*tail*) $\land \neg$*is_wchar_node*(*tail*) **then** $p \leftarrow$ *tail*
    else if *type*(*tail*) = *ligature_node* **then** $p \leftarrow$ *lig_char*(*tail*)
      else return;
    $f \leftarrow$ *font*(*p*); *tail_append*(*new_kern*(*char_italic*(*f*)(*char_info*(*f*)(*character*(*p*)))));
    subtype(*tail*) $\leftarrow$ *explicit*;
    end;
exit: **end**;

1117. 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 226 ⟩ +≡
  primitive("`-`", *discretionary*, 1); *primitive*("`discretionary`", *discretionary*, 0);

1118. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
discretionary: **if** *chr_code* = 1 **then** *print_esc*("`-`") **else** *print_esc*("`discretionary`");

1119. ⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
hmode + *discretionary*, *mmode* + *discretionary*: *append_discretionary*;

1120. 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* 1046 ⟩ +≡
procedure *append_discretionary*;
  var *c*: *integer*; { hyphen character }
  begin *tail_append*(*new_disc*);
  if *cur_chr* = 1 **then**
    begin $c \leftarrow$ *hyphen_char*[*cur_font*];
    if $c \geq 0$ **then**
      if $c < 256$ **then** *pre_break*(*tail*) $\leftarrow$ *new_character*(*cur_font*, *c*);
    end
  else begin *incr*(*save_ptr*); *saved*(-1) $\leftarrow 0$; *new_save_level*(*disc_group*); *scan_left_brace*; *push_nest*;
    mode $\leftarrow -$*hmode*; *space_factor* $\leftarrow 1000$;
    end;
  end;

1121. 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 1088 ⟩ +≡
disc_group: *build_discretionary*;

1122. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
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 1124⟩;
$p \leftarrow link(head)$; *pop_nest*;
case *saved*(−1) **of**
0: $pre\_break(tail) \leftarrow p$;
1: $post\_break(tail) \leftarrow p$;
2: ⟨Attach list *p* to the current list, and record its length; then finish up and **return** 1123⟩;
end; { there are no other cases }
incr(*saved*(−1)); *new_save_level*(*disc_group*); *scan_left_brace*; *push_nest*; $mode \leftarrow -hmode$;
$space\_factor \leftarrow 1000$;
exit: **end**;

1123. ⟨Attach list *p* to the current list, and record its length; then finish up and **return** 1123⟩ ≡
begin if $(n > 0) \land (abs(mode) = mmode)$ **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*); $n \leftarrow 0$;
error;
end
else $link(tail) \leftarrow p$;
if $n \le max\_quarterword$ **then** $replace\_count(tail) \leftarrow n$
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** $tail \leftarrow q$;
decr(*save_ptr*); **return**;
end

This code is used in section 1122.

1124. During this loop, $p = link(q)$ 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 1124 ⟩ ≡

```
  q ← head; p ← link(q); n ← 0;
  while p ≠ null do
    begin if ¬is_char_node(p) then
      if type(p) > rule_node then
        if type(p) ≠ kern_node then
          if type(p) ≠ 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 ← p; p ← link(q); incr(n);
    end;
done:
```

This code is used in section 1122.

1125. 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 1059 ⟩ +≡
hmode + *accent*: *make_accent*;

1126. 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* 1046 ⟩ +≡

```
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 ← cur_font; p ← new_character(f, cur_val);
  if p ≠ null then
    begin x ← x_height(f); s ← slant(f)/float_constant(65536);
    a ← char_width(f)(char_info(f)(character(p)));
    do_assignments;
    ⟨ Create a character node q for the next character, but set q ← null if problems arise 1127 ⟩;
    if q ≠ null then ⟨ Append the accent with appropriate kerns, then set p ← q 1128 ⟩;
    link(tail) ← p; tail ← p; space_factor ← 1000;
    end;
  end;
```

1127. ⟨ Create a character node q for the next character, but set $q \leftarrow null$ if problems arise 1127 ⟩ ≡

$q \leftarrow null$; $f \leftarrow cur\_font$;
if $(cur\_cmd = letter) \lor (cur\_cmd = other\_char) \lor (cur\_cmd = char\_given)$ **then**
  $q \leftarrow new\_character(f, cur\_chr)$
else if $cur\_cmd = char\_num$ **then**
    begin $scan\_char\_num$; $q \leftarrow new\_character(f, cur\_val)$;
    end
  else $back\_input$

This code is used in section 1126.

1128. 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`.

⟨ Append the accent with appropriate kerns, then set $p \leftarrow q$ 1128 ⟩ ≡

begin $t \leftarrow slant(f)/float\_constant(65536)$; $i \leftarrow char\_info(f)(character(q))$; $w \leftarrow char\_width(f)(i)$;
$h \leftarrow char\_height(f)(height\_depth(i))$;
if $h \neq x$ **then** { the accent must be shifted up or down }
  begin $p \leftarrow hpack(p, natural)$; $shift\_amount(p) \leftarrow x - h$;
  end;
$delta \leftarrow round((w - a)/float\_constant(2) + h * t - x * s)$; $r \leftarrow new\_kern(delta)$; $subtype(r) \leftarrow acc\_kern$;
$link(tail) \leftarrow r$; $link(r) \leftarrow p$; $tail \leftarrow new\_kern(-a - delta)$; $subtype(tail) \leftarrow acc\_kern$; $link(p) \leftarrow tail$;
$p \leftarrow q$;
end

This code is used in section 1126.

1129. 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.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡

$any\_mode(car\_ret)$, $any\_mode(tab\_mark)$: $align\_error$;
$any\_mode(no\_align)$: no_align_error;
$any\_mode(omit)$: $omit\_error$;

1130. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

```
procedure align_error;
  begin if abs(align_state) > 2 then
    ⟨ Express consternation over the fact that no alignment is in progress 1131 ⟩
  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;
```

1131. ⟨ Express consternation over the fact that no alignment is in progress 1131 ⟩ ≡

```
  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 1130.

1132. 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* 1046 ⟩ +≡

```
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;
```

1133. 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 1059 ⟩ +≡
vmode + *halign*, *hmode* + *valign*: *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*;

1134. 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* 1046 ⟩ +≡
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;

1135. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1088 ⟩ +≡
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;

1136. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1088 ⟩ +≡
no_align_group: **begin** *end_graf*; *unsave*; *align_peek*;
 end;

1137. Finally, `\endcsname` is not supposed to get through to *main_control*.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
any_mode(*end_cs_name*): *cs_error*;

1138. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *cs_error*;
 begin *print_err*("Extra␣"); *print_esc*("endcsname");
 help1("I´m␣ignoring␣this,␣since␣I␣wasn´t␣doing␣a␣\csname."); *error*;
 end;

1139. 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* 1046⟩ +≡
procedure *push_math*(*c* : *group_code*);
  begin *push_nest*; *mode* ← −*mmode*; *incompleat_noad* ← *null*; *new_save_level*(*c*);
  end;

1140. 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 1059⟩ +≡
hmode + *math_shift*: *init_math*;

1141. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
procedure *init_math*;
  label *reswitch*, *found*, *not_found*, *done*;
  var *w*: *scaled*; { new or partial *pre_display_size* }
    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 1148⟩
  else begin *back_input*; ⟨Go into ordinary math mode 1142⟩;
    end;
  end;

1142. ⟨Go into ordinary math mode 1142⟩ ≡
  begin *push_math*(*math_shift_group*); *eq_word_define*(*int_base* + *cur_fam_code*, −1);
  if (*insert_src_special_every_math*) **then** *insert_src_special*;
  if *every_math* ≠ *null* **then** *begin_token_list*(*every_math*, *every_math_text*);
  end

This code is used in sections 1141 and 1145.

1143. 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 1059⟩ +≡
mmode + *eq_no*: **if** *privileged* **then**
    if *cur_group* = *math_shift_group* **then** *start_eq_no*
    else *off_save*;

1144. ⟨Put each of TEX's primitives into the hash table 226⟩ +≡
  primitive("`eqno`", *eq_no*, 0); *primitive*("`leqno`", *eq_no*, 1);

1145. When TeX 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* 1046 ⟩ +≡
procedure *start_eq_no*;
  begin *saved*(0) ← *cur_chr*; *incr*(*save_ptr*); ⟨ Go into ordinary math mode 1142 ⟩;
  end;

1146. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
eq_no: **if** *chr_code* = 1 **then** *print_esc*("leqno") **else** *print_esc*("eqno");

1147. ⟨ Forbidden cases detected in *main_control* 1051 ⟩ +≡
  non_math(*eq_no*),

1148. 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 1148 ⟩ ≡
  begin if *head* = *tail* **then** { '\noindent$$' or '$$ $$' }
    begin *pop_nest*; *w* ← −*max_dimen*;
    end
  else begin *line_break*(*display_widow_penalty*);
    ⟨ 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 1149 ⟩;
    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 1152 ⟩;
  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*);
  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 1141.

1149. ⟨ 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 1149 ⟩ ≡
  v ← *shift_amount*(*just_box*) + 2 ∗ *quad*(*cur_font*); *w* ← −*max_dimen*; *p* ← *list_ptr*(*just_box*);
  while *p* ≠ *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* ← *max_dimen* 1150 ⟩;
    if *v* < *max_dimen* **then** *v* ← *v* + *d*;
    goto *not_found*;
  found: **if** *v* < *max_dimen* **then**
      begin *v* ← *v* + *d*; *w* ← *v*;
      end
    else begin *w* ← *max_dimen*; **goto** *done*;
      end;
  not_found: *p* ← *link*(*p*);
    end;
done:

This code is used in section 1148.

1150. ⟨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 $v \leftarrow max\_dimen$ 1150⟩ ≡

reswitch: **if** *is_char_node*(p) **then**
  begin $f \leftarrow font(p)$;
  if *is_wchar_node*(p) **then** $d \leftarrow cfont\_width[f]$
  else $d \leftarrow char\_width(f)(char\_info(f)(character(p)))$;
  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* 655⟩;
kern_node, *math_node*: $d \leftarrow width(p)$;
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 1151⟩;
whatsit_node: ⟨Let d be the width of the whatsit p 1364⟩;
othercases $d \leftarrow 0$
endcases

This code is used in section 1149.

1151. 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.

⟨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 1151⟩ ≡

begin $q \leftarrow glue\_ptr(p)$; $d \leftarrow width(q)$;
if $glue\_sign(just\_box) = stretching$ **then**
  begin if $(glue\_order(just\_box) = stretch\_order(q)) \land (stretch(q) \neq 0)$ **then** $v \leftarrow max\_dimen$;
  end
else if $glue\_sign(just\_box) = shrinking$ **then**
  begin if $(glue\_order(just\_box) = shrink\_order(q)) \land (shrink(q) \neq 0)$ **then** $v \leftarrow max\_dimen$;
  end;
if $subtype(p) \geq a\_leaders$ **then goto** *found*;
end

This code is used in section 1150.

1152. 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 1152 $\rangle \equiv$

if *par_shape_ptr* $=$ *null* **then**
  if $(hang\_indent \neq 0) \land (((hang\_after \geq 0) \land (prev\_graf + 2 > hang\_after)) \lor$
      $(prev\_graf + 1 < -hang\_after))$ **then**
    begin $l \leftarrow hsize - abs(hang\_indent)$;
    if $hang\_indent > 0$ **then** $s \leftarrow hang\_indent$ **else** $s \leftarrow 0$;
    end
  else begin $l \leftarrow hsize$; $s \leftarrow 0$;
    end
else begin $n \leftarrow info(par\_shape\_ptr)$;
  if $prev\_graf + 2 \geq n$ **then** $p \leftarrow par\_shape\_ptr + 2 * n$
  else $p \leftarrow par\_shape\_ptr + 2 * (prev\_graf + 2)$;
  $s \leftarrow mem[p - 1].sc$; $l \leftarrow mem[p].sc$;
  end

This code is used in section 1148.

1153. 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 1059 $\rangle +\equiv$

mmode $+$ *left_brace*: **begin** *tail_append*(*new_noad*); *back_input*; *scan_math*(*nucleus*(*tail*));
  end;

1154. Recall that the *nucleus*, *subscr*, and *supscr* fields in a noad 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* 1046⟩ +≡
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 407⟩;
reswitch: **case** *cur_cmd* **of**
  letter, *other_char*, *char_given*: **begin if** *is_wchar*(*cur_chr*) **then**
      begin *print_err*("Chinese␣character␣is␣ignored␣in␣math␣mode");
      help1("Did␣you␣forget␣putting␣it␣into␣an␣\hbox?"); *error*; **goto** *restart*;
      end
    else begin *c* ← *ho*(*math_code*(*cur_chr*));
      if *c* = ´100000 **then**
        begin ⟨Treat *cur_chr* as an active character 1155⟩;
        goto *restart*;
        end;
      end;
    end;
  char_num: **begin** *scan_char_num*; *cur_chr* ← *cur_val*; *cur_cmd* ← *char_given*; **goto** *reswitch*;
    end;
  pux_char_num: **begin** *scan_wchar_num*; *cur_chr* ← *cur_val*; *cur_cmd* ← *pux_char_given*; **goto** *reswitch*;
    end;
  pux_char_given: **begin** *print_err*("Chinese␣character␣is␣ignored␣in␣math␣mode");
    help1("Did␣you␣forget␣putting␣it␣into␣an␣\hbox?"); *error*; **goto** *restart*;
    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** 1156⟩
  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**;

1155. An active character that is an *outer_call* is allowed here.

⟨Treat *cur_chr* as an active character 1155⟩ ≡
  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 1154 and 1158.

1156. The pointer p is placed on *save_stack* while a complex subformula is being scanned.

⟨ Scan a subformula enclosed in braces and **return** 1156 ⟩ ≡
  begin *back_input*; *scan_left_brace*;
  saved(0) ← p; *incr*(*save_ptr*); *push_math*(*math_group*); **return**;
  end

This code is used in section 1154.

1157. 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 1059 ⟩ +≡
mmode + *letter*, *mmode* + *other_char*, *mmode* + *char_given*: **if** *is_wchar*(*cur_chr*) **then**
    begin *print_err*("`Chinese␣character␣is␣ignored␣in␣math␣mode`");
    help1("`Did␣you␣forget␣putting␣it␣into␣an␣\hbox?`"); *error*;
    end
  else *set_math_char*(*ho*(*math_code*(*cur_chr*)));
mmode + *char_num*: **begin** *scan_char_num*; *cur_chr* ← *cur_val*; *set_math_char*(*ho*(*math_code*(*cur_chr*)));
  end;
mmode + *math_char_num*: **begin** *scan_fifteen_bit_int*; *set_math_char*(*cur_val*);
  end;
mmode + *math_given*: *set_math_char*(*cur_chr*);
mmode + *delim_num*: **begin** *scan_twenty_seven_bit_int*; *set_math_char*(*cur_val* **div** ´*10000*);
  end;

1158. 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* 1046 ⟩ +≡
procedure *set_math_char*(c : *integer*);
  var p: *pointer*; { the new noad }
  begin if $c \geq$ ´*100000* **then** ⟨ Treat *cur_chr* as an active character 1155 ⟩
  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 $c \geq$ *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;

1159. 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 226 ⟩ +≡
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*);

1160. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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");

1161. ⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
mmode + *math_comp*: **begin** *tail_append*(*new_noad*); *type*(*tail*) ← *cur_chr*; *scan_math*(*nucleus*(*tail*));
end;
mmode + *limit_switch*: *math_limit_switch*;

1162. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
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."); *error*;
exit: **end**;

1163. 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 1046⟩ +≡
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 407⟩;
    case cur_cmd of
    letter, other_char: if is_wchar(cur_chr) then cur_val ← −1
      else 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 1164⟩;
  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;
```

1164. ⟨Report that an invalid delimiter code is being changed to null; set *cur_val* ← 0 1164⟩ ≡

```
  begin print_err("Missing␣delimiter␣(.␣inserted)");
  help6("I␣was␣expecting␣to␣see␣something␣like␣`(´␣or␣`\{´␣or")
  ("`\}´␣here.␣If␣you␣typed,␣e.g.,␣`{´␣instead␣of␣`\{´,␣you")
  ("should␣probably␣delete␣the␣`{´␣by␣typing␣`1´␣now,␣so␣that")
  ("braces␣don´t␣get␣unbalanced.␣Otherwise␣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 1163.

1165. ⟨Cases of *main_control* that build boxes and lists 1059⟩ +≡

```
mmode + radical: math_radical;
```

1166. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡

```
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;
```

1167. ⟨Cases of *main_control* that build boxes and lists 1059⟩ +≡

```
mmode + accent, mmode + math_accent: math_ac;
```

1168. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
procedure *math_ac*;
begin if *cur_cmd* = *accent* **then** ⟨Complain that the user should have said `\mathaccent` 1169⟩;
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;

1169. ⟨Complain that the user should have said `\mathaccent` 1169⟩ ≡
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 1168.

1170. ⟨Cases of *main_control* that build boxes and lists 1059⟩ +≡
mmode + *vcenter*: **begin** *scan_spec*(*vcenter_group*, *false*); *normal_paragraph*; *push_nest*; *mode* ← −*vmode*;
prev_depth ← *ignore_depth*;
if (*insert_src_special_every_vbox*) **then** *insert_src_special*;
if *every_vbox* ≠ *null* **then** *begin_token_list*(*every_vbox*, *every_vbox_text*);
end;

1171. ⟨Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1088⟩ +≡
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;

1172. The routine that inserts a *style_node* holds no surprises.

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

1173. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
math_style: *print_style*(*chr_code*);

1174. ⟨Cases of *main_control* that build boxes and lists 1059⟩ +≡
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*;

1175. The routine that scans the four mlists of a `\mathchoice` is very much like the routine that builds discretionary nodes.

⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *append_choices*;
  begin *tail_append*(*new_choice*); *incr*(*save_ptr*); *saved*(−1) ← 0; *push_math*(*math_choice_group*);
  scan_left_brace;
  end;

1176. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1088 ⟩ +≡
math_choice_group: *build_choices*;

1177. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
⟨ Declare the function called *fin_mlist* 1187 ⟩
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**;

1178. 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.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
mmode + *sub_mark*, *mmode* + *sup_mark*: *sub_sup*;

1179. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *sub_sup*;
  var *t*: *small_number*; { type of previous sub/superscript }
    p: *pointer*; { field to be filled by *scan_math* }
  begin *t* ← *empty*; *p* ← *null*;
  if *tail* ≠ *head* **then**
    if *scripts_allowed*(*tail*) **then**
      begin *p* ← *supscr*(*tail*) + *cur_cmd* − *sup_mark*; { *supscr* or *subscr* }
      t ← *math_type*(*p*);
      end;
  if (*p* = *null*) ∨ (*t* ≠ *empty*) **then** ⟨ Insert a dummy noad to be sub/superscripted 1180 ⟩;
  scan_math(*p*);
  end;

1180. ⟨ Insert a dummy noad to be sub/superscripted 1180 ⟩ ≡
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 1179.

1181. 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. }

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

1182. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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;

1183. ⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
mmode + *above*: *math_fraction*;

1184. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *math_fraction*;
  var *c*: *small_number*; { the type of generalized fraction we are scanning }
  begin $c \leftarrow$ *cur_chr*;
  if *incompleat_noad* $\neq$ *null* **then**
    ⟨ Ignore the fraction operation and complain about this ambiguous case 1186 ⟩
  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 1185 ⟩;
    end;
  end;

1185. ⟨ Use code *c* to distinguish between generalized fractions 1185 ⟩ ≡
  if $c \geq$ *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 1184.

1186. ⟨ Ignore the fraction operation and complain about this ambiguous case 1186 ⟩ ≡
  begin if $c \geq$ *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 1184.

1187. 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* 1187⟩ ≡
function *fin_mlist*(*p* : *pointer*): *pointer*;
  var *q*: *pointer*; { the mlist to return }
  begin if *incompleat_noad* $\neq$ *null* **then** ⟨Compleat the incompleat noad 1188⟩
  else begin *link*(*tail*) $\leftarrow$ *p*; *q* $\leftarrow$ *link*(*head*);
    end;
  pop_nest; *fin_mlist* $\leftarrow$ *q*;
  end;

This code is used in section 1177.

1188. ⟨Compleat the incompleat noad 1188⟩ ≡
  begin *math_type*(*denominator*(*incompleat_noad*)) $\leftarrow$ *sub_mlist*;
  info(*denominator*(*incompleat_noad*)) $\leftarrow$ *link*(*head*);
  if $p = null$ **then** *q* $\leftarrow$ *incompleat_noad*
  else begin *q* $\leftarrow$ *info*(*numerator*(*incompleat_noad*));
    if *type*(*q*) $\neq$ *left_noad* **then** *confusion*("right");
    info(*numerator*(*incompleat_noad*)) $\leftarrow$ *link*(*q*); *link*(*q*) $\leftarrow$ *incompleat_noad*; *link*(*incompleat_noad*) $\leftarrow$ *p*;
    end;
  end

This code is used in section 1187.

1189. 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 1088⟩ +≡
math_group: **begin** *unsave*; *decr*(*save_ptr*);
  math_type(*saved*(0)) $\leftarrow$ *sub_mlist*; *p* $\leftarrow$ *fin_mlist*(*null*); *info*(*saved*(0)) $\leftarrow$ *p*;
  if $p \neq null$ **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* $\leftarrow$ *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* 1190⟩;
  end;

1190. ⟨Replace the tail of the list by *p* 1190⟩ ≡
  begin *q* $\leftarrow$ *head*;
  while *link*(*q*) $\neq$ *tail* **do** *q* $\leftarrow$ *link*(*q*);
  link(*q*) $\leftarrow$ *p*; *free_node*(*tail*, *noad_size*); *tail* $\leftarrow$ *p*;
  end

This code is used in section 1189.

1191. 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 226 ⟩ +≡
primitive("left", *left_right*, *left_noad*); *primitive*("right", *left_right*, *right_noad*);
text(*frozen_right*) ← "right"; *eqtb*[*frozen_right*] ← *eqtb*[*cur_val*];

1192. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
left_right: **if** *chr_code* = *left_noad* **then** *print_esc*("left")
else *print_esc*("right");

1193. ⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
mmode + *left_right*: *math_left_right*;

1194. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡

```
procedure math_left_right;
  var t: small_number;  { left_noad or right_noad }
    p: pointer;  { new noad }
  begin t ← cur_chr;
  if (t = right_noad) ∧ (cur_group ≠ math_left_group) then ⟨ Try to recover from mismatched \right 1195 ⟩
  else begin p ← new_noad; type(p) ← t; scan_delimiter(delimiter(p), false);
    if t = left_noad then
      begin push_math(math_left_group); link(head) ← p; tail ← p;
      end
    else begin p ← fin_mlist(p); unsave;  { end of math_left_group }
      tail_append(new_noad); type(tail) ← inner_noad; math_type(nucleus(tail)) ← sub_mlist;
      info(nucleus(tail)) ← p;
      end;
    end;
  end;
```

1195. ⟨ Try to recover from mismatched \right 1195 ⟩ ≡

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

This code is used in section 1194.

1196. Here is the only way out of math mode.

⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡

```
mmode + math_shift: if cur_group = math_shift_group then
    begin after_math;
    if math_mode_save < 0 then
      begin get_x_token; ⟨ If the token is a wide character, then append a cspace 1452 ⟩;
      goto reswitch;
      end;
    end
  else off_save;
```

1197. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
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 1201 ⟩
  begin *danger* ← *false*; ⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1198 ⟩;
  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 1200 ⟩;
    cur_mlist ← *p*; *cur_style* ← *text_style*; *mlist_penalties* ← *false*; *mlist_to_hlist*;
    a ← *hpack*(*link*(*temp_head*), *natural*); *unsave*; *decr*(*save_ptr*); { now *cur_group* = *math_shift_group* }
    if *saved*(0) = 1 **then** *l* ← *true*;
    danger ← *false*; ⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1198 ⟩;
    m ← *mode*; *p* ← *fin_mlist*(*null*);
    end
  else *a* ← *null*;
  if *m* < 0 **then** ⟨ Finish math in text 1199 ⟩
  else begin if *a* = *null* **then** ⟨ Check that another $ follows 1200 ⟩;
    ⟨ Finish displayed math 1202 ⟩;
    end;
  end;

1198. ⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1198 ⟩ ≡
  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_mathex_params*) ∨
      (*font_params*[*fam_fnt*(3 + *script_size*)] < *total_mathex_params*) ∨
      (*font_params*[*fam_fnt*(3 + *script_script_size*)] < *total_mathex_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 1197 and 1197.

1199. The *unsave* 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 1199 ⟩ ≡
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*);
math_mode_save ← *m*; *tail_append*(*new_math*(*math_surround*, *after*)); *space_factor* ← 1000; *unsave*;
end

This code is used in section 1197.

1200. TEX gets to the following part of the program when the first '`$`' ending a display has been scanned.

⟨ Check that another `$` follows 1200 ⟩ ≡
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 1197, 1197, and 1209.

1201. 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 1201 ⟩ ≡
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 }

This code is used in section 1197.

1202. 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 1202⟩ ≡

cur_mlist ← p; *cur_style* ← *display_style*; *mlist_penalties* ← *false*; *mlist_to_hlist*; p ← *link*(*temp_head*);
adjust_tail ← *adjust_head*; b ← *hpack*(p, *natural*); p ← *list_ptr*(b); t ← *adjust_tail*; *adjust_tail* ← *null*;
w ← *width*(b); z ← *display_width*; s ← *display_indent*;
if (a = *null*) ∨ *danger* **then**
  begin $e \leftarrow 0$; $q \leftarrow 0$;
  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 $e \leftarrow 0$ 1204⟩;
⟨Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming that l = *false* 1205⟩;
⟨Append the glue or equation number preceding the display 1206⟩;
⟨Append the display and perhaps also the equation number 1207⟩;
⟨Append the glue or equation number following the display 1208⟩;
resume_after_display

This code is used in section 1197.

1203. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡

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 446⟩;
if *nest_ptr* = 1 **then** *build_page*;
end;

1204. 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 $e \leftarrow 0$ 1204⟩ ≡

begin if ($e \neq 0$) ∧ ((w − *total_shrink*[*normal*] + $q \le z$) ∨
  (*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 $e \leftarrow 0$;
  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 1202.

1205. We try first to center the display without regard to the existence of the equation number. If that would make it too close (where "too close" means that the space between display and equation number is less than the width of the equation number), we either center it in the remaining space or move it as far from the equation number as possible. The latter alternative is taken only if the display begins with glue, since we assume that the user put glue there to control the spacing precisely.

⟨ Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming that $l = false$ 1205 ⟩ ≡
  $d \leftarrow half(z - w)$;
  if $(e > 0) \land (d < 2 * e)$ **then** { too close }
    begin $d \leftarrow half(z - w - e)$;
    if $p \neq null$ **then**
      if $\neg is\_char\_node(p)$ **then**
        if $type(p) = glue\_node$ **then** $d \leftarrow 0$;
    end

This code is used in section 1202.

1206. 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 1206 ⟩ ≡
  $tail\_append(new\_penalty(pre\_display\_penalty))$;
  if $(d + s \leq pre\_display\_size) \lor l$ **then** { not enough clearance }
    begin $g1 \leftarrow above\_display\_skip\_code$; $g2 \leftarrow below\_display\_skip\_code$;
    end
  else begin $g1 \leftarrow above\_display\_short\_skip\_code$; $g2 \leftarrow below\_display\_short\_skip\_code$;
    end;
  if $l \land (e = 0)$ **then** { it follows that $type(a) = hlist\_node$ }
    begin $shift\_amount(a) \leftarrow s$; $append\_to\_vlist(a)$; $tail\_append(new\_penalty(inf\_penalty))$;
    end
  else $tail\_append(new\_param\_glue(g1))$

This code is used in section 1202.

1207. ⟨ Append the display and perhaps also the equation number 1207 ⟩ ≡
  if $e \neq 0$ **then**
    begin $r \leftarrow new\_kern(z - w - e - d)$;
    if l **then**
      begin $link(a) \leftarrow r$; $link(r) \leftarrow b$; $b \leftarrow a$; $d \leftarrow 0$;
      end
    else begin $link(b) \leftarrow r$; $link(r) \leftarrow a$;
      end;
    $b \leftarrow hpack(b, natural)$;
    end;
  $shift\_amount(b) \leftarrow s + d$; $append\_to\_vlist(b)$

This code is used in section 1202.

1208. ⟨Append the glue or equation number following the display 1208⟩ ≡
if $(a \neq \mathit{null}) \land (e = 0) \land \lnot l$ **then**
begin *tail_append*(*new_penalty*(*inf_penalty*)); *shift_amount*(a) ← $s + z -$ *width*(a); *append_to_vlist*(a);
g2 ← 0;
end;
if $t \neq$ *adjust_head* **then** { migrating material comes after equation number }
begin *link*(*tail*) ← *link*(*adjust_head*); *tail* ← t;
end;
tail_append(*new_penalty*(*post_display_penalty*));
if *g2* > 0 **then** *tail_append*(*new_param_glue*(*g2*))

This code is used in section 1202.

1209. When `\halign` appears in a display, the alignment routines operate essentially as they do in vertical mode. Then the following program is activated, with p and q pointing to the beginning and end of the resulting list, and with *aux_save* holding the *prev_depth* value.

⟨Finish an alignment in a display 1209⟩ ≡
begin *do_assignments*;
if *cur_cmd* ≠ *math_shift* **then** ⟨Pontificate about improper alignment in display 1210⟩
else ⟨Check that another `$` follows 1200⟩;
pop_nest; *tail_append*(*new_penalty*(*pre_display_penalty*));
tail_append(*new_param_glue*(*above_display_skip_code*)); *link*(*tail*) ← p;
if $p \neq$ *null* **then** *tail* ← q;
tail_append(*new_penalty*(*post_display_penalty*)); *tail_append*(*new_param_glue*(*below_display_skip_code*));
prev_depth ← *aux_save.sc*; *resume_after_display*;
end

This code is used in section 815.

1210. ⟨Pontificate about improper alignment in display 1210⟩ ≡
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 1209.

1211. 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`' or '`\outer`', and it might or might not be expanded. The prefixes '`\global`', '`\long`', and '`\outer`' can occur in any order. Therefore we assign binary numeric codes, making it possible to accumulate the union of all specified prefixes by adding the corresponding codes. (Pascal's **set** operations could also have been used.)

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

1212. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
prefix: **if** *chr_code* = 1 **then** *print_esc*("`long`")
  else if *chr_code* = 2 **then** *print_esc*("`outer`")
    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`");

1213. 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* 1213 ⟩ ≡
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*(*set_cfont*),
  any_mode(*pux_cface_def*), *any_mode*(*pux_face_match*), *any_mode*(*pux_font_match*),
  any_mode(*pux_set_cface*), *any_mode*(*puxg_assign_flag*), *any_mode*(*puxg_assign_int*),
  any_mode(*pux_get_int*), *any_mode*(*pux_set_cface_attrib*), *any_mode*(*pux_set_cfont_attrib*),
  any_mode(*pux_range_catcode*), *any_mode*(*pux_range_type_code*), *any_mode*(*pux_split_number*),
  any_mode(*puxg_assign_space*), *any_mode*(*pux_set_default_cface*), *any_mode*(*pux_dump_font_info*),
  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 1271, 1274, 1277, 1279, 1288, and 1293.

This code is used in section 1048.

1214. If the user says, e.g., '\global\global', the redundancy is silently accepted.

```
⟨Declare action procedures for use by main_control 1046⟩ +≡
⟨Declare PUTeX subprocedures for prefixed_command 1505⟩
⟨Declare subprocedures for prefixed_command 1218⟩
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?}
    ⟨Other variables used by the procedure prefixed_command 1415⟩
  begin a ← 0;
  while cur_cmd = prefix do
    begin if ¬odd(a div cur_chr) then a ← a + cur_chr;
    ⟨Get the next non-blank non-relax non-call token 407⟩;
    if cur_cmd ≤ max_non_prefixed_command then ⟨Discard erroneous prefixes and return 1215⟩;
    end;
  ⟨Discard the prefixes \long and \outer if they are irrelevant 1216⟩;
  ⟨Adjust for the setting of \globaldefs 1217⟩;
  case cur_cmd of
  ⟨Assignments 1220⟩
  othercases confusion("prefix")
  endcases;
done: ⟨Insert a token saved by \afterassignment, if any 1272⟩;
exit: end;
```

1215. ⟨Discard erroneous prefixes and **return** 1215⟩ ≡

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

This code is used in section 1214.

1216. ⟨Discard the prefixes \long and \outer if they are irrelevant 1216⟩ ≡

```
  if (cur_cmd ≠ def) ∧ (a mod 4 ≠ 0) then
    begin print_err("You␣can´t␣use␣`"); print_esc("long"); print("´␣or␣`"); print_esc("outer");
    print("´␣with␣`"); print_cmd_chr(cur_cmd, cur_chr); print_char("´");
    help1("I´ll␣pretend␣you␣didn´t␣say␣\long␣or␣\outer␣here."); error;
    end
```

This code is used in section 1214.

1217. The previous routine does not have to adjust a so that a **mod** $4 = 0$, since the following routines test for the `\global` prefix as follows.

define *global* $\equiv (a \geq 4)$
define *define*(#) $\equiv$
  if *global* **then** *geq_define*(#) **else** *eq_define*(#)
define *word_define*(#) $\equiv$
  if *global* **then** *geq_word_define*(#) **else** *eq_word_define*(#)

⟨Adjust for the setting of `\globaldefs` 1217⟩ $\equiv$
if *global_defs* $\neq 0$ **then**
  if *global_defs* < 0 **then**
    begin if *global* **then** $a \leftarrow a - 4$;
    end
  else begin if $\neg$*global* **then** $a \leftarrow a + 4$;
    end

This code is used in section 1214.

1218. 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* 1218⟩ $\equiv$
procedure *get_r_token*;
  label *restart*;
  begin *restart*: **repeat** *get_token*;
  until *cur_tok* $\neq$ *space_token*;
  if (*cur_cs* $= 0$) $\lor$ (*cur_cs* $>$ *eqtb_top*) $\lor$ ((*cur_cs* $>$ *frozen_control_sequence*) $\land$ (*cur_cs* $\leq$ *eqtb_size*)) **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 $\leftarrow$ *cs_token_flag* + *frozen_protection*; *ins_error*; **goto** *restart*;
    end;
  end;

See also sections 1232, 1239, 1246, 1247, 1248, 1249, 1250, 1260, 1268, 1482, 1524, 1526, 1530, and 1535.

This code is used in section 1214.

1219. ⟨Initialize table entries (done by `INITEX` only) 164⟩ $+\equiv$
text(*frozen_protection*) $\leftarrow$ "inaccessible";

1220. Here's an example of the way many of the following routines operate. (Unfortunately, they aren't all as simple as this.)

⟨Assignments 1220⟩ $\equiv$
set_font: **begin** *define*(*cur_font_loc*, *data*, *cur_chr*);
  ⟨Set the matching CJK font 1537⟩;
  end;

See also sections 1221, 1224, 1227, 1228, 1229, 1231, 1235, 1237, 1238, 1244, 1245, 1251, 1255, 1256, 1259, 1267, 1416, 1427, 1430, 1471, 1477, 1522, 1534, 1539, 1541, 1544, 1546, 1550, 1558, 1563, and 1567.

This code is used in section 1214.

1221. When a *def* command has been scanned, *cur_chr* is odd if the definition is supposed to be global, and $cur\_chr \ge 2$ if the definition is supposed to be expanded.

⟨ Assignments 1220 ⟩ +≡
def: **begin if** $odd(cur\_chr) \land \lnot global \land (global\_defs \ge 0)$ **then** $a \leftarrow a + 4$;
  $e \leftarrow (cur\_chr \ge 2)$; *get_r_token*; $p \leftarrow cur\_cs$; $q \leftarrow scan\_toks(true, e)$; $define(p, call + (a \bmod 4), def\_ref)$;
  end;

1222. Both `\let` and `\futurelet` share the command code *let*.

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive(`"let"`, *let*, *normal*);
  primitive(`"futurelet"`, *let*, $normal + 1$);

1223. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
let: **if** $chr\_code \ne normal$ **then** *print_esc*(`"futurelet"`) **else** *print_esc*(`"let"`);

1224. ⟨ Assignments 1220 ⟩ +≡
let: **begin** $n \leftarrow cur\_chr$; *get_r_token*; $p \leftarrow cur\_cs$;
  if $n = normal$ **then**
    begin repeat *get_token*;
    until $cur\_cmd \ne spacer$;
    if $cur\_tok = other\_token +$ `"="` **then**
      begin *get_token*;
      if $cur\_cmd = spacer$ **then** *get_token*;
      end;
    end
  else begin *get_token*; $q \leftarrow cur\_tok$; *get_token*; *back_input*; $cur\_tok \leftarrow q$; *back_input*;
        { look ahead, then back up }
    end; { note that *back_input* doesn't affect *cur_cmd*, *cur_chr* }
  if $cur\_cmd \ge call$ **then** *add_token_ref*(*cur_chr*);
  define(*p*, *cur_cmd*, *cur_chr*);
  end;

1225. A `\chardef` creates a control sequence whose *cmd* is *char_given*; a `\mathchardef` creates a control sequence whose *cmd* is *math_given*; and the corresponding *chr* is the character code or math code. A `\countdef` or `\dimendef` or `\skipdef` or `\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 *char_def_code* = 0 { *shorthand_def* for `\chardef` }
define *math_char_def_code* = 1 { *shorthand_def* for `\mathchardef` }
define *count_def_code* = 2 { *shorthand_def* for `\countdef` }
define *dimen_def_code* = 3 { *shorthand_def* for `\dimendef` }
define *skip_def_code* = 4 { *shorthand_def* for `\skipdef` }
define *mu_skip_def_code* = 5 { *shorthand_def* for `\muskipdef` }
define *toks_def_code* = 6 { *shorthand_def* for `\toksdef` }
define *pux_char_def_code* = 7 { *shorthand_def* for `\PUXchardef` }
define *char_sub_def_code* = 7 { *shorthand_def* for `\charsubdef` }

⟨ Put each of TeX's primitives into the hash table 226 ⟩ +≡
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*);
primitive("PUXchardef", *shorthand_def*, *pux_char_def_code*);
if *mltex_p* **then**
  begin *primitive*("charsubdef", *shorthand_def*, *char_sub_def_code*);
  end;

1226. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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");
  char_sub_def_code: *print_esc*("charsubdef");
  toks_def_code: *print_esc*("toksdef");
  othercases *print_esc*("PUXchardef")
  endcases;
char_given: **begin** *print_esc*("char"); *print_hex*(*chr_code*);
  end;
math_given: **begin** *print_esc*("mathchar"); *print_hex*(*chr_code*);
  end;

1227. 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 1220⟩ +≡
shorthand_def: if cur_chr = char_sub_def_code then
    begin scan_char_num; p ← char_sub_code_base + cur_val; scan_optional_equals; scan_char_num;
    n ← cur_val; { accent character in substitution }
    scan_char_num;
    if (tracing_char_sub_def > 0) then
      begin begin_diagnostic; print_nl("New␣character␣substitution:␣");
      print_ASCII(p − char_sub_code_base); print("␣=␣"); print_ASCII(n); print_char("␣");
      print_ASCII(cur_val); end_diagnostic(false);
      end;
    n ← n ∗ 256 + cur_val; define(p, data, hi(n));
    if (p − char_sub_code_base) < char_sub_def_min then
      word_define(int_base + char_sub_def_min_code, p − char_sub_code_base);
    if (p − char_sub_code_base) > char_sub_def_max then
      word_define(int_base + char_sub_def_max_code, p − char_sub_code_base);
    end
  else 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;
    pux_char_def_code: begin scan_wchar_num; define(p, pux_char_given, cur_val);
      end;
    othercases begin scan_eight_bit_int;
      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;
```

1228. ⟨Assignments 1220⟩ +≡

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

1229. 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 1220 ⟩ +≡
toks_register, *assign_toks*: **begin** $q \leftarrow cur\_cs$;
  if $cur\_cmd = toks\_register$ **then**
    begin *scan_eight_bit_int*; $p \leftarrow toks\_base + cur\_val$;
    end
  else $p \leftarrow cur\_chr$; { $p = every\_par\_loc$ or $output\_routine\_loc$ or ... }
  scan_optional_equals; ⟨ Get the next non-blank non-relax non-call token 407 ⟩;
  if $cur\_cmd \neq left\_brace$ **then** ⟨ If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1230 ⟩;
  back_input; $cur\_cs \leftarrow q$; $q \leftarrow scan\_toks(false, false)$;
  if $link(def\_ref) = null$ **then** { empty list: revert to the default }
    begin $define(p, undefined\_cs, null)$; $free\_avail(def\_ref)$;
    end
  else begin if $p = output\_routine\_loc$ **then** { enclose in curlies }
      begin $link(q) \leftarrow get\_avail$; $q \leftarrow link(q)$; $info(q) \leftarrow right\_brace\_token +$ "}"; $q \leftarrow get\_avail$;
      $info(q) \leftarrow left\_brace\_token +$ "{"; $link(q) \leftarrow link(def\_ref)$; $link(def\_ref) \leftarrow q$;
      end;
    $define(p, call, def\_ref)$;
    end;
  end;

1230. ⟨ If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1230 ⟩ ≡
  begin if $cur\_cmd = toks\_register$ **then**
    begin *scan_eight_bit_int*; $cur\_cmd \leftarrow assign\_toks$; $cur\_chr \leftarrow toks\_base + cur\_val$;
    end;
  if $cur\_cmd = assign\_toks$ **then**
    begin $q \leftarrow equiv(cur\_chr)$;
    if $q = null$ **then** $define(p, undefined\_cs, null)$
    else begin $add\_token\_ref(q)$; $define(p, call, q)$;
      end;
    goto *done*;
    end;
  end

This code is used in section 1229.

1231. Similar routines are used to assign values to the numeric parameters.

⟨ Assignments 1220 ⟩ +≡
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;

1232. 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* 1218 ⟩ +≡
procedure *trap_zero_glue*;
  begin if (*width*(*cur_val*) = 0) ∧ (*stretch*(*cur_val*) = 0) ∧ (*shrink*(*cur_val*) = 0) **then**
    begin *add_glue_ref*(*zero_glue*); *delete_glue_ref*(*cur_val*); *cur_val* ← *zero_glue*;
    end;
  end;

1233. 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 226 ⟩ +≡
  primitive(`"catcode"`, *def_code*, *cat_code_base*); *primitive*(`"PUXcatcode"`, *def_code*, *pux_cat_code_base*);
  primitive(`"PUXtypecode"`, *def_code*, *pux_type_code_base*);
  primitive(`"PUXlocalnames"`, *def_code*, *pux_local_names_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*);

1234. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
def_code: **if** *chr_code* = *cat_code_base* **then** *print_esc*(`"catcode"`)
  else if *chr_code* = *pux_cat_code_base* **then** *print_esc*(`"PUXcatcode"`)
    else if *chr_code* = *pux_type_code_base* **then** *print_esc*(`"PUXtypecode"`)
      else if *chr_code* = *pux_local_names_base* **then** *print_esc*(`"PUXlocalnames"`)
        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*);

1235. The different types of code values have different legal ranges; the following program is careful to check each case properly.

```
⟨Assignments 1220⟩ +≡
def_code: begin ⟨Let n be the largest legal code value, based on cur_chr 1236⟩;
  p ← cur_chr;
  if p = pux_cat_code_base then
    begin scan_wchar_num; p ← cat_code_base;
    end
  else if p = pux_type_code_base then scan_wchar_num
    else if p = pux_local_names_base then scan_eight_bit_int
      else scan_char_num;
  p ← p + cur_val; scan_optional_equals;
  if p = pux_local_names_base then scan_wchar_num
  else 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;
```

1236.

```
⟨Let n be the largest legal code value, based on cur_chr 1236⟩ ≡
  if cur_chr = cat_code_base then n ← max_char_code
  else if cur_chr = pux_cat_code_base then n ← max_char_code
    else if cur_chr = pux_type_code_base then n ← max_type_code
      else if cur_chr = pux_local_names_base then n ← 65535
        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 1235.

1237.

```
⟨Assignments 1220⟩ +≡
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;
```

1238. Next we consider changes to TEX's numeric registers.

```
⟨Assignments 1220⟩ +≡
register, advance, multiply, divide: do_register_command(a);
```

1239. We use the fact that *register* < *advance* < *multiply* < *divide*.

⟨ Declare subprocedures for *prefixed_command* 1218 ⟩ +≡
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 }
  begin *q* ← *cur_cmd*; ⟨ Compute the register location *l* and its type *p*; but **return** if invalid 1240 ⟩;
  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* 1241 ⟩
  else ⟨ Compute result of *multiply* or *divide*, put it in *cur_val* 1243 ⟩;
  if *arith_error* **then**
    begin *print_err* ("Arithmetic␣overflow");
    help2 ("I␣can´t␣carry␣out␣that␣multiplication␣or␣division,")
    ("since␣the␣result␣is␣out␣of␣range.");
    if *p* ≥ *glue_val* **then** *delete_glue_ref* (*cur_val*);
    error; **return**;
    end;
  if *p* < *glue_val* **then** *word_define* (*l*, *cur_val*)
  else begin *trap_zero_glue*; *define* (*l*, *glue_ref*, *cur_val*);
    end;
exit: **end**;

1240. 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 1240 ⟩ ≡
  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␣can´t␣use␣`"); *print_cmd_chr* (*cur_cmd*, *cur_chr*); *print* ("´␣after␣");
      print_cmd_chr (*q*, 0); *help1* ("I´m␣forgetting␣what␣you␣said␣and␣not␣changing␣anything.");
      error; **return**;
      end;
    end;
  p ← *cur_chr*; *scan_eight_bit_int*;
  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;
found:

This code is used in section 1239.

1241. ⟨Compute result of *register* or *advance*, put it in *cur_val* 1241⟩ ≡

if $p < glue\_val$ **then**
  begin if $p = int\_val$ **then** *scan_int* **else** *scan_normal_dimen*;
  if $q = advance$ **then** $cur\_val \leftarrow cur\_val + eqtb[l].int$;
  end
else begin *scan_glue*(p);
  if $q = advance$ **then** ⟨Compute the sum of two glue specs 1242⟩;
  end

This code is used in section 1239.

1242. ⟨Compute the sum of two glue specs 1242⟩ ≡

begin $q \leftarrow new\_spec(cur\_val)$; $r \leftarrow equiv(l)$; $delete\_glue\_ref(cur\_val)$; $width(q) \leftarrow width(q) + width(r)$;
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 $(stretch\_order(q) < stretch\_order(r)) \land (stretch(r) \neq 0)$ **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 $(shrink\_order(q) < shrink\_order(r)) \land (shrink(r) \neq 0)$ **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 1241.

1243. ⟨Compute result of *multiply* or *divide*, put it in *cur_val* 1243⟩ ≡

begin *scan_int*;
if $p < glue\_val$ **then**
  if $q = multiply$ **then**
    if $p = int\_val$ **then** $cur\_val \leftarrow mult\_integers(eqtb[l].int, cur\_val)$
    else $cur\_val \leftarrow nx\_plus\_y(eqtb[l].int, cur\_val, 0)$
  else $cur\_val \leftarrow x\_over\_n(eqtb[l].int, cur\_val)$
else begin $s \leftarrow equiv(l)$; $r \leftarrow new\_spec(s)$;
  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 1239.

1244. The processing of boxes is somewhat different, because we may need to scan and create an entire box before we actually change the value of the old one.

⟨ Assignments 1220 ⟩ +≡
set_box: **begin** *scan_eight_bit_int*;
  if *global* **then** $n \leftarrow 256 + cur\_val$ **else** $n \leftarrow cur\_val$;
  scan_optional_equals;
  if *set_box_allowed* **then**
    begin *in_set_box* ← *true*; *scan_box*(*box_flag* + *n*); *in_set_box* ← *false*;
    end
  else begin *print_err*("Improper␣"); *print_esc*("setbox");
    help2("Sorry,␣\setbox␣is␣not␣allowed␣after␣\halign␣in␣a␣display,")
    ("or␣between␣\accent␣and␣an␣accented␣character."); *error*;
    end;
  end;

1245. 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 1220 ⟩ +≡
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*;

1246. ⟨ Declare subprocedures for *prefixed_command* 1218 ⟩ +≡
procedure *alter_aux*;
  var *c*: *halfword*; { *hmode* or *vmode* }
  begin if $cur\_chr \neq abs(mode)$ **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 $(cur\_val \le 0) \lor (cur\_val > 32767)$ **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;

1247. ⟨Declare subprocedures for *prefixed_command* 1218⟩ +≡
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;

1248. ⟨Declare subprocedures for *prefixed_command* 1218⟩ +≡
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;

1249. ⟨Declare subprocedures for *prefixed_command* 1218⟩ +≡
procedure *alter_integer*;
  var *c*: 0 .. 1; { 0 for `\deadcycles`, 1 for `\insertpenalties` }
  begin *c* ← *cur_chr*; *scan_optional_equals*; *scan_int*;
  if *c* = 0 **then** *dead_cycles* ← *cur_val*
  else *insert_penalties* ← *cur_val*;
  end;

1250. ⟨Declare subprocedures for *prefixed_command* 1218⟩ +≡
procedure *alter_box_dimen*;
  var *c*: *small_number*; { *width_offset* or *height_offset* or *depth_offset* }
    b: *eight_bits*; { box number }
  begin *c* ← *cur_chr*; *scan_eight_bit_int*; *b* ← *cur_val*; *scan_optional_equals*; *scan_normal_dimen*;
  if *box*(*b*) ≠ *null* **then** *mem*[*box*(*b*) + *c*].*sc* ← *cur_val*;
  end;

1251. Paragraph shapes are set up in the obvious way.

⟨Assignments 1220⟩ +≡
set_shape: **begin** *scan_optional_equals*; *scan_int*; *n* ← *cur_val*;
  if *n* ≤ 0 **then** *p* ← *null*
  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(*par_shape_loc*, *shape_ref*, *p*);
  end;

1252. 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.

⟨ Check the "constant" values for consistency 14 ⟩ +≡
 if $2 * max\_halfword < mem\_top - mem\_min$ **then** $bad \leftarrow 41$;

1253. New hyphenation data is loaded by the *hyph_data* command.

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
 primitive(`"hyphenation"`, *hyph_data*, 0); *primitive*(`"patterns"`, *hyph_data*, 1);

1254. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
hyph_data: **if** $chr\_code = 1$ **then** *print_esc*(`"patterns"`)
 else *print_esc*(`"hyphenation"`);

1255. ⟨ Assignments 1220 ⟩ +≡
hyph_data: **if** $cur\_chr = 1$ **then**
  begin Init *new_patterns*; **goto** *done*; **Tini**
  print_err(`"Patterns␣can␣be␣loaded␣only␣by␣INITEX"`); *help0*; *error*;
  repeat *get_token*;
  until $cur\_cmd = right\_brace$; { flush the patterns }
  return;
  end
 else begin *new_hyph_exceptions*; **goto** *done*;
  end;

1256. All of TEX's parameters are kept in *eqtb* except the font information, the interaction mode, and the hyphenation tables; these are strictly global.

⟨ Assignments 1220 ⟩ +≡
assign_font_dimen: **begin** *find_font_dimen*(*true*); $k \leftarrow cur\_val$; *scan_optional_equals*; *scan_normal_dimen*;
 $font\_info[k].sc \leftarrow cur\_val$;
 end;
assign_font_int: **begin** $n \leftarrow cur\_chr$; *scan_font_ident*; $f \leftarrow cur\_val$; *scan_optional_equals*; *scan_int*;
 if $n = 0$ **then** $hyphen\_char[f] \leftarrow cur\_val$ **else** $skew\_char[f] \leftarrow cur\_val$;
 end;

1257. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
 primitive(`"hyphenchar"`, *assign_font_int*, 0); *primitive*(`"skewchar"`, *assign_font_int*, 1);

1258. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
assign_font_int: **if** $chr\_code = 0$ **then** *print_esc*(`"hyphenchar"`)
 else *print_esc*(`"skewchar"`);

1259. Here is where the information for a new font gets loaded.

⟨ Assignments 1220 ⟩ +≡
def_font: *new_font*(*a*);

1260. ⟨ Declare subprocedures for *prefixed_command* 1218 ⟩ +≡
⟨ Declare the function called *fw_times_sd* 1441 ⟩
⟨ Declare the function called *find_cface_num* 1481 ⟩
⟨ Declare the procedure called *check_cfont* 1512 ⟩
⟨ Declare the procedure called *make_cfont* 1514 ⟩
procedure *new_font*(*a* : *small_number*);
  label *common_ending*;
  var *u*: *pointer*; { user's font identifier }
    j, *k*: *pool_pointer*; *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 }
    ⟨ Other local variables used by procedure *new_font* 1508 ⟩
  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;
  scan_optional_equals; *scan_file_name*;
  ⟨ Scan the font size specification 1261 ⟩;
  if (*length*(*cur_name*) > 5) **then**
    begin *j* ← *str_start*[*cur_name*];
    if (*str_pool*[*j*] = ´`C`´ ∧ *str_pool*[*j*+1] = ´`F`´ ∧ *str_pool*[*j*+2] = ´`O`´ ∧ *str_pool*[*j*+3] = ´`N`´ ∧ *str_pool*[*j*+4] = ´`T`´) **then** ⟨ Define a CJK font and then goto *common_ending* 1509 ⟩;
    end;
  define(*u*, *set_font*, *null_font*);
  ⟨ If this font has already been loaded, set *f* to the internal font number and **goto** *common_ending* 1263 ⟩;
  f ← *read_font_info*(*u*, *cur_name*, *cur_area*, *s*);
common_ending: *equiv*(*u*) ← *f*; *eqtb*[*font_id_base* + *f*] ← *eqtb*[*u*]; *font_id_text*(*f*) ← *t*;
  end;

1261. ⟨ Scan the font size specification 1261 ⟩ ≡
  name_in_progress ← *true*; { this keeps *cur_name* from being changed }
  if *scan_keyword*(`"at"`) **then** ⟨ Put the (positive) 'at' size into *s* 1262 ⟩
  else if *scan_keyword*(`"scaled"`) **then**
      begin *scan_int*; *s* ← −*cur_val*;
      if (*cur_val* ≤ 0) ∨ (*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 ← −1000;
        end;
      end
    else *s* ← −1000;
  name_in_progress ← *false*

This code is used in section 1260.

1262. ⟨ Put the (positive) 'at' size into s 1262 ⟩ ≡
begin *scan_normal_dimen*; $s \leftarrow cur\_val$;
if $(s \le 0) \lor (s \ge '1000000000)$ **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 1261.

1263. 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.

⟨ If this font has already been loaded, set f to the internal font number and **goto** *common_ending* 1263 ⟩ ≡
for $f \leftarrow font\_base + 1$ **to** *font_ptr* **do**
if *str_eq_str*(*font_name*[f], *cur_name*) ∧ *str_eq_str*(*font_area*[f], *cur_area*) **then**
begin 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 1260.

1264. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
set_font: **begin** *print*("select␣font␣"); *slow_print*(*font_name*[*chr_code*]);
if *font_size*[*chr_code*] ≠ *font_dsize*[*chr_code*] **then**
begin *print*("␣at␣"); *print_scaled*(*font_size*[*chr_code*]); *print*("pt");
end;
end;

1265. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
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*);

1266. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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;

1267. ⟨ Assignments 1220 ⟩ +≡
set_interaction: *new_interaction*;

1268. ⟨ Declare subprocedures for *prefixed_command* 1218 ⟩ +≡
procedure *new_interaction*;
  begin *print_ln*; *interaction* ← *cur_chr*;
  if *interaction* = *batch_mode* **then** *kpse_make_tex_discard_errors* ← 1
  else *kpse_make_tex_discard_errors* ← 0;
  ⟨ Initialize the print *selector* based on *interaction* 75 ⟩;
  if *log_opened* **then** *selector* ← *selector* + 2;
  end;

1269. 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 }

1270. ⟨ Set initial values of key variables 21 ⟩ +≡
  after_token ← 0;

1271. ⟨ Cases of *main_control* that don't depend on *mode* 1213 ⟩ +≡
any_mode(*after_assignment*): **begin** *get_token*; *after_token* ← *cur_tok*;
  end;

1272. ⟨ Insert a token saved by `\afterassignment`, if any 1272 ⟩ ≡
  if *after_token* ≠ 0 **then**
    begin *cur_tok* ← *after_token*; *back_input*; *after_token* ← 0;
    end

This code is used in section 1214.

1273. 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* 1046 ⟩ +≡
procedure *do_assignments*;
  label *exit*;
  begin loop
    begin ⟨ Get the next non-blank non-relax non-call token 407 ⟩;
    if *cur_cmd* ≤ *max_non_prefixed_command* **then return**;
    set_box_allowed ← *false*; *prefixed_command*; *set_box_allowed* ← *true*;
    end;
exit: **end**;

1274. ⟨ Cases of *main_control* that don't depend on *mode* 1213 ⟩ +≡
any_mode(*after_group*): **begin** *get_token*; *save_for_after*(*cur_tok*);
  end;

1275. Files for `\read` are opened and closed by the *in_stream* command.

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive(`"openin"`, *in_stream*, 1); *primitive*(`"closein"`, *in_stream*, 0);

1276. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
in_stream: **if** *chr_code* = 0 **then** *print_esc*(`"closein"`)
  else *print_esc*(`"openin"`);

1277. ⟨Cases of *main_control* that don't depend on *mode* 1213⟩ +≡
any_mode(*in_stream*): *open_or_close_in*;

1278. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
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*; *pack_cur_name*; $tex\_input\_type \leftarrow 0$;
        { Tell *open_input* we are `\openin`. }
    if *kpse_in_name_ok*(*stringcast*(*name_of_file* + 1)) ∧ *a_open_in*(*read_file*[*n*], *kpse_tex_format*) **then**
      $read\_open[n] \leftarrow just\_open$;
    end;
  end;

1279. The user can issue messages to the terminal, regardless of the current mode.
⟨Cases of *main_control* that don't depend on *mode* 1213⟩ +≡
any_mode(*message*): *issue_message*;

1280. ⟨Put each of TEX's primitives into the hash table 226⟩ +≡
  primitive(`"message"`, *message*, 0); *primitive*(`"errmessage"`, *message*, 1);

1281. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
message: **if** $chr\_code = 0$ **then** *print_esc*(`"message"`)
  else *print_esc*(`"errmessage"`);

1282. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
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 $c \leftarrow cur\_chr$; $link(garbage) \leftarrow scan\_toks(false, true)$; $old\_setting \leftarrow selector$;
  $selector \leftarrow new\_string$; *token_show*(*def_ref*); $selector \leftarrow old\_setting$; *flush_list*(*def_ref*); *str_room*(1);
  $s \leftarrow make\_string$;
  if $c = 0$ **then** ⟨Print string *s* on the terminal 1283⟩
  else ⟨Print string *s* as an error message 1286⟩;
  flush_string;
  end;

1283. ⟨Print string *s* on the terminal 1283⟩ ≡
  begin if $term\_offset + length(s) > max\_print\_line - 2$ **then** *print_ln*
  else if $(term\_offset > 0) \lor (file\_offset > 0)$ **then** *print_char*(`"␣"`);
  slow_print(*s*); *update_terminal*;
  end
This code is used in section 1282.

1284. 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? }

1285. ⟨ Set initial values of key variables 21 ⟩ +≡
long_help_seen ← *false*;

1286. ⟨ Print string s as an error message 1286 ⟩ ≡
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 1282.

1287. 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;

1288. 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* 1213 ⟩ +≡
any_mode(*case_shift*): *shift_case*;

1289. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("lowercase", *case_shift*, *lc_code_base*); *primitive*("uppercase", *case_shift*, *uc_code_base*);

1290. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
case_shift: **if** *chr_code* = *lc_code_base* **then** *print_esc*("lowercase")
else *print_esc*("uppercase");

1291. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
procedure *shift_case*;
  var *b*: *pointer*; { *lc_code_base* or *uc_code_base* }
    p: *pointer*; { runs through the token list }
    t: *halfword*; { token }
    c: *quarterword*; { character code }
  begin $b \leftarrow cur\_chr$; $p \leftarrow scan\_toks(false, false)$; $p \leftarrow link(def\_ref)$;
  while $p \neq null$ **do**
    begin ⟨Change the case of the token in *p*, if a change is appropriate 1292⟩;
    $p \leftarrow link(p)$;
    end;
  back_list(*link*(*def_ref*)); *free_avail*(*def_ref*); { omit reference count }
  end;

1292. 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.

⟨Change the case of the token in *p*, if a change is appropriate 1292⟩ ≡
  $t \leftarrow info(p)$;
  if $t < cs\_token\_flag + single\_base$ **then**
    begin $c \leftarrow t$ **mod** 65536;
    if $c < 256$ **then** { only convert the single-byte char }
      if $equiv(b + c) \neq 0$ **then** $info(p) \leftarrow t - c + equiv(b + c)$;
    end

This code is used in section 1291.

1293. We come finally to the last pieces missing from *main_control*, namely the '`\show`' commands that are useful when debugging.

⟨Cases of *main_control* that don't depend on *mode* 1213⟩ +≡
any_mode(*xray*): *show_whatever*;

1294. **define** *show_code* = 0 { `\show` }
  define *show_box_code* = 1 { `\showbox` }
  define *show_the_code* = 2 { `\showthe` }
  define *show_lists* = 3 { `\showlists` }

⟨Put each of TEX's primitives into the hash table 226⟩ +≡
  primitive("`show`", *xray*, *show_code*); *primitive*("`showbox`", *xray*, *show_box_code*);
  primitive("`showthe`", *xray*, *show_the_code*); *primitive*("`showlists`", *xray*, *show_lists*);

1295. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
xray: **case** *chr_code* **of**
  show_box_code: *print_esc*("`showbox`");
  show_the_code: *print_esc*("`showthe`");
  show_lists: *print_esc*("`showlists`");
  othercases *print_esc*("`show`")
  endcases;

1296. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
procedure *show_whatever*;
  label *common_ending*;
  var *p*: *pointer*; { tail of a token list to show }
  begin case *cur_chr* **of**
  show_lists: **begin** *begin_diagnostic*; *show_activities*;
    end;
  show_box_code: ⟨ Show the current contents of a box 1299 ⟩;
  show_code: ⟨ Show the current meaning of a token, then **goto** *common_ending* 1297 ⟩;
  othercases ⟨ Show the current value of some parameter or register, then **goto** *common_ending* 1300 ⟩
  endcases;
  ⟨ Complete a potentially long `\show` command 1301 ⟩;
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;

1297. ⟨ Show the current meaning of a token, then **goto** *common_ending* 1297 ⟩ ≡
  begin *get_token*;
  if *interaction* = *error_stop_mode* **then** *wake_up_terminal*;
  print_nl(`">␣"`);
  if *cur_cs* $\neq 0$ **then**
    begin *sprint_cs*(*cur_cs*); *print_char*(`"="`);
    end;
  print_meaning; **goto** *common_ending*;
  end

This code is used in section 1296.

1298. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
undefined_cs: *print*(`"undefined"`);
call: *print*(`"macro"`);
long_call: *print_esc*(`"long␣macro"`);
outer_call: *print_esc*(`"outer␣macro"`);
long_outer_call: **begin** *print_esc*(`"long"`); *print_esc*(`"outer␣macro"`);
  end;
end_template: *print_esc*(`"outer␣endtemplate"`);

1299. ⟨Show the current contents of a box 1299⟩ ≡
begin *scan_eight_bit_int*; *begin_diagnostic*; *print_nl*(`">␣\box"`); *print_int*(*cur_val*); *print_char*(`"="`);
if *box*(*cur_val*) = *null* **then** *print*(`"void"`)
else *show_box*(*box*(*cur_val*));
end

This code is used in section 1296.

1300. ⟨Show the current value of some parameter or register, then **goto** *common_ending* 1300⟩ ≡
begin $p \leftarrow$ *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 1296.

1301. ⟨Complete a potentially long `\show` command 1301⟩ ≡
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 1296.

1302. 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 TEX 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 TEX is ready to start. For `INITEX` this string says simply '`(INITEX)`'; for other versions of TEX 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 TEX's tables are loaded.

⟨Global variables 13⟩ +≡
format_ident: *str_number*;

1303. ⟨Set initial values of key variables 21⟩ +≡
format_ident ← 0;

1304. ⟨Initialize table entries (done by `INITEX` only) 164⟩ +≡
if *ini_version* **then** *format_ident* ← "␣(INITEX)";

1305. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
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 }
format_engine: ↑*text_char*;
begin ⟨If dumping is not allowed, abort 1307⟩;
⟨Create the *format_ident*, open the format file, and inform the user that dumping has begun 1331⟩;
⟨Dump constants for consistency check 1310⟩;
⟨Dump MLTEX-specific data 1403⟩;
⟨Dump the string pool 1312⟩;
⟨Dump the dynamic memory 1314⟩;
⟨Dump the table of equivalents 1316⟩;
⟨Dump the font information 1323⟩;
⟨Dump the CJK font face information 1576⟩;
⟨Dump the face matching table 1578⟩;
⟨Dump the CJK font information 1580⟩;
⟨Dump the hyphenation tables 1327⟩;
⟨Dump a couple more things and the closing check word 1329⟩;
⟨Close the format file 1332⟩;
end;
tini

1306. 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 TEX 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* 527 ⟩
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 }
    format_engine: ↑*text_char*; *dummy_xord*: *ASCII_code*; *dummy_xchr*: *text_char*;
    dummy_xprn: *ASCII_code*;
  begin ⟨ Undump constants for consistency check 1311 ⟩;
  ⟨ Undump MLTEX-specific data 1404 ⟩;
  ⟨ Undump the string pool 1313 ⟩;
  ⟨ Undump the dynamic memory 1315 ⟩;
  ⟨ Undump the table of equivalents 1317 ⟩;
  ⟨ Undump the font information 1324 ⟩;
  ⟨ Undump the CJK font face information 1577 ⟩;
  ⟨ Unump the face matching table 1579 ⟩;
  ⟨ Undump the CJK font information 1581 ⟩;
  ⟨ Undump the hyphenation tables 1328 ⟩;
  ⟨ Undump a couple more things and the closing check word 1330 ⟩;
  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**;

1307. 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 1307 ⟩ ≡
  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 1305.

1308. Format files consist of *memory_word* items, and we use the following macros to dump words of different types:

⟨ Global variables 13 ⟩ +≡
fmt_file: *word_file*; { for input or output of format information }

1309. 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 $a \le x \le b$.

```
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 format_debug_end(#) ≡ write_ln(stderr, ´␣=␣´, #);
        end ;
define format_debug(#) ≡
        if debug_format_file then
          begin write(stderr, ´fmtdebug:´, #); format_debug_end
define undump_size_end_end(#) ≡ too_small(#) else format_debug(#)(x); 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
```

1310. The next few sections of the program should make it clear how we use the dump/undump macros.

```
⟨ Dump constants for consistency check 1310 ⟩ ≡
  dump_int("57325458); { Web2C TEX's magic constant: "W2TX" }
    { Align engine to 4 bytes with one or more trailing NUL }
  x ← strlen(engine_name); format_engine ← xmalloc_array(text_char, x + 4);
  strcpy(stringcast(format_engine), engine_name);
  for k ← x to x + 3 do format_engine[k] ← 0;
  x ← x + 4 − (x mod 4); dump_int(x); dump_things(format_engine[0], x); libc_free(format_engine);
  dump_int(@$);
  ⟨ Dump xord, xchr, and xprn 1389 ⟩;
  dump_int(max_halfword);
  dump_int(hash_high); dump_int(mem_bot);
  dump_int(mem_top);
  dump_int(eqtb_size);
  dump_int(hash_prime);
  dump_int(hyph_prime)
```

This code is used in section 1305.

1311. 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 1311⟩ ≡ Init libc_free(font_info); libc_free(str_pool);
  libc_free(str_start); libc_free(yhash); libc_free(zeqtb); libc_free(yzmem); Tiniundump_int(x);
  format_debug('format␣magic␣number')(x);
  if x ≠ "57325458 then goto bad_fmt;   { not a format file }
  undump_int(x); format_debug('engine␣name␣size')(x);
  if (x < 0) ∨ (x > 256) then goto bad_fmt;   { corrupted format file }
  format_engine ← xmalloc_array(text_char, x); undump_things(format_engine[0], x);
  format_engine[x − 1] ← 0;   { force string termination, just in case }
  if strcmp(engine_name, stringcast(format_engine)) then
    begin wake_up_terminal;
    wterm_ln('---!␣', stringcast(name_of_file + 1), '␣was␣written␣by␣', format_engine);
    libc_free(format_engine); goto bad_fmt;
    end;
  libc_free(format_engine); undump_int(x); format_debug('string␣pool␣checksum')(x);
  if x ≠ @$ then
    begin   { check that strings are the same }
    wake_up_terminal; wterm_ln('---!␣', stringcast(name_of_file + 1), '␣doesn''t␣match␣', pool_name);
    goto bad_fmt;
    end;
  ⟨Undump xord, xchr, and xprn 1390⟩;
  undump_int(x);
  if x ≠ max_halfword then goto bad_fmt;   { check max_halfword }
  undump_int(hash_high);
  if (hash_high < 0) ∨ (hash_high > sup_hash_extra) then goto bad_fmt;
  if hash_extra < hash_high then hash_extra ← hash_high;
  eqtb_top ← eqtb_size + hash_extra;
  if hash_extra = 0 then hash_top ← undefined_control_sequence
  else hash_top ← eqtb_top;
  yhash ← xmalloc_array(two_halves, 1 + hash_top − hash_offset); hash ← yhash − hash_offset;
  next(hash_base) ← 0; text(hash_base) ← 0;
  for x ← hash_base + 1 to hash_top do hash[x] ← hash[hash_base];
  zeqtb ← xmalloc_array(memory_word, eqtb_top + 1); eqtb ← zeqtb;
  eq_type(undefined_control_sequence) ← undefined_cs; equiv(undefined_control_sequence) ← null;
  eq_level(undefined_control_sequence) ← level_zero;
  for x ← eqtb_size + 1 to eqtb_top do eqtb[x] ← eqtb[undefined_control_sequence];
  undump_int(x); format_debug('mem_bot')(x);
  if x ≠ mem_bot then goto bad_fmt;
  undump_int(mem_top); format_debug('mem_top')(mem_top);
  if mem_bot + 1100 > mem_top then goto bad_fmt;
  head ← contrib_head; tail ← contrib_head; page_tail ← page_head;   { page initialization }
  mem_min ← mem_bot − extra_mem_bot; mem_max ← mem_top + extra_mem_top;
  yzmem ← xmalloc_array(memory_word, mem_max − mem_min + 1); zmem ← yzmem − mem_min;
      { this pointer arithmetic fails with some compilers }
  mem ← zmem; undump_int(x);
  if x ≠ eqtb_size then goto bad_fmt;
  undump_int(x);
  if x ≠ hash_prime then goto bad_fmt;
  undump_int(x);
  if x ≠ hyph_prime then goto bad_fmt
```

This code is used in section 1306.

1312. **define** $dump\_four\_ASCII \equiv w.b0 \leftarrow qi(so(str\_pool[k]))$; $w.b1 \leftarrow qi(so(str\_pool[k+1]))$; $w.b2 \leftarrow qi(so(str\_pool[k+2]))$; $w.b3 \leftarrow qi(so(str\_pool[k+3]))$; $dump\_qqqq(w)$

⟨ Dump the string pool 1312 ⟩ ≡
$dump\_int(pool\_ptr)$; $dump\_int(str\_ptr)$; $dump\_things(str\_start[0], str\_ptr + 1)$;
$dump\_things(str\_pool[0], pool\_ptr)$; $print\_ln$; $print\_int(str\_ptr)$; $print($"␣strings␣of␣total␣length␣"$)$;
$print\_int(pool\_ptr)$

This code is used in section 1305.

1313. **define** $undump\_four\_ASCII \equiv undump\_qqqq(w)$; $str\_pool[k] \leftarrow si(qo(w.b0))$; $str\_pool[k+1] \leftarrow si(qo(w.b1))$; $str\_pool[k+2] \leftarrow si(qo(w.b2))$; $str\_pool[k+3] \leftarrow si(qo(w.b3))$

⟨ Undump the string pool 1313 ⟩ ≡
$undump\_size(0)(sup\_pool\_size - pool\_free)($´string␣pool␣size´$)(pool\_ptr)$;
if $pool\_size < pool\_ptr + pool\_free$ **then** $pool\_size \leftarrow pool\_ptr + pool\_free$;
$undump\_size(0)(sup\_max\_strings - strings\_free)($´sup␣strings´$)(str\_ptr)$;
if $max\_strings < str\_ptr + strings\_free$ **then** $max\_strings \leftarrow str\_ptr + strings\_free$;
$str\_start \leftarrow xmalloc\_array(pool\_pointer, max\_strings)$;
$undump\_checked\_things(0, pool\_ptr, str\_start[0], str\_ptr + 1)$;
$str\_pool \leftarrow xmalloc\_array(packed\_ASCII\_code, pool\_size)$; $undump\_things(str\_pool[0], pool\_ptr)$;
$init\_str\_ptr \leftarrow str\_ptr$; $init\_pool\_ptr \leftarrow pool\_ptr$

This code is used in section 1306.

1314. 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 1314 ⟩ ≡
$sort\_avail$; $var\_used \leftarrow 0$; $dump\_int(lo\_mem\_max)$; $dump\_int(rover)$; $p \leftarrow mem\_bot$; $q \leftarrow rover$; $x \leftarrow 0$;
repeat $dump\_things(mem[p], q + 2 - p)$; $x \leftarrow x + q + 2 - p$; $var\_used \leftarrow var\_used + q - p$;
  $p \leftarrow q + node\_size(q)$; $q \leftarrow rlink(q)$;
until $q = rover$;
$var\_used \leftarrow var\_used + lo\_mem\_max - p$; $dyn\_used \leftarrow mem\_end + 1 - hi\_mem\_min$;
$dump\_things(mem[p], lo\_mem\_max + 1 - p)$; $x \leftarrow x + lo\_mem\_max + 1 - p$; $dump\_int(hi\_mem\_min)$;
$dump\_int(avail)$; $dump\_things(mem[hi\_mem\_min], mem\_end + 1 - hi\_mem\_min)$;
$x \leftarrow x + mem\_end + 1 - hi\_mem\_min$; $p \leftarrow avail$;
while $p \neq null$ **do**
  begin $decr(dyn\_used)$; $p \leftarrow link(p)$;
  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 1305.

1315. ⟨Undump the dynamic memory 1315⟩ ≡
undump(*lo_mem_stat_max* + 1000)(*hi_mem_stat_min* − 1)(*lo_mem_max*);
undump(*lo_mem_stat_max* + 1)(*lo_mem_max*)(*rover*); $p \leftarrow$ *mem_bot*; $q \leftarrow$ *rover*;
repeat *undump_things*(*mem*[*p*], $q + 2 - p$); $p \leftarrow q +$ *node_size*(*q*);
  if ($p >$ *lo_mem_max*) ∨ (($q \geq$ *rlink*(*q*)) ∧ (*rlink*(*q*) ≠ *rover*)) **then goto** *bad_fmt*;
  $q \leftarrow$ *rlink*(*q*);
until $q =$ *rover*;
undump_things(*mem*[*p*], *lo_mem_max* + 1 − *p*);
if *mem_min* < *mem_bot* − 2 **then** { make more low memory available }
  begin $p \leftarrow$ *llink*(*rover*); $q \leftarrow$ *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*; *undump_things*(*mem*[*hi_mem_min*], *mem_end* + 1 − *hi_mem_min*);
undump_int(*var_used*); *undump_int*(*dyn_used*)

This code is used in section 1306.

1316. ⟨Dump the table of equivalents 1316⟩ ≡
⟨Dump regions 1 to 4 of *eqtb* 1318⟩;
⟨Dump regions 5 and 6 of *eqtb* 1319⟩;
dump_int(*par_loc*); *dump_int*(*write_loc*);
⟨Dump the hash table 1321⟩

This code is used in section 1305.

1317. ⟨Undump the table of equivalents 1317⟩ ≡
⟨Undump regions 1 to 6 of *eqtb* 1320⟩;
undump(*hash_base*)(*hash_top*)(*par_loc*); *par_token* ← *cs_token_flag* + *par_loc*;
undump(*hash_base*)(*hash_top*)(*write_loc*);
⟨Undump the hash table 1322⟩

This code is used in section 1306.

1318. The table of equivalents usually contains repeated information, so we dump it in compressed form: The sequence of $n+2$ values $(n, x_1, \ldots, x_n, m)$ in the format file represents $n+m$ consecutive entries of *eqtb*, with m extra copies of x_n, namely $(x_1, \ldots, x_n, x_n, \ldots, x_n)$.

⟨Dump regions 1 to 4 of *eqtb* 1318⟩ ≡
```
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); dump_things(eqtb[k], l − k); k ← j + 1; dump_int(k − l);
until k = int_base
```
This code is used in section 1316.

1319. ⟨Dump regions 5 and 6 of *eqtb* 1319⟩ ≡
```
repeat j ← k;
  while j < eqtb_size do
    begin if eqtb[j].int = eqtb[j + 1].int then goto found2;
    incr(j);
    end;
  l ← eqtb_size + 1; goto done2;  { j = eqtb_size }
found2: incr(j); l ← j;
  while j < eqtb_size do
    begin if eqtb[j].int ≠ eqtb[j + 1].int then goto done2;
    incr(j);
    end;
done2: dump_int(l − k); dump_things(eqtb[k], l − k); k ← j + 1; dump_int(k − l);
until k > eqtb_size;
if hash_high > 0 then dump_things(eqtb[eqtb_size + 1], hash_high);  { dump hash_extra part }
```
This code is used in section 1316.

1320. ⟨Undump regions 1 to 6 of *eqtb* 1320⟩ ≡
```
k ← active_base;
repeat undump_int(x);
  if (x < 1) ∨ (k + x > eqtb_size + 1) then goto bad_fmt;
  undump_things(eqtb[k], x); k ← k + x; undump_int(x);
  if (x < 0) ∨ (k + x > eqtb_size + 1) then goto bad_fmt;
  for j ← k to k + x − 1 do eqtb[j] ← eqtb[k − 1];
  k ← k + x;
until k > eqtb_size;
if hash_high > 0 then undump_things(eqtb[eqtb_size + 1], hash_high);  { undump hash_extra part }
```
This code is used in section 1317.

1321. A different scheme is used to compress the hash table, since its lower region is usually sparse. When $text(p) \neq 0$ for $p \le hash\_used$, we output two words, p and $hash[p]$. The hash table is, of course, densely packed for $p \ge hash\_used$, so the remaining entries are output in a block.

⟨Dump the hash table 1321⟩ ≡
dump_int(*hash_used*); *cs_count* ← *frozen_control_sequence* − 1 − *hash_used* + *hash_high*;
for *p* ← *hash_base* **to** *hash_used* **do**
  if *text*(*p*) ≠ 0 **then**
    begin *dump_int*(*p*); *dump_hh*(*hash*[*p*]); *incr*(*cs_count*);
    end;
dump_things(*hash*[*hash_used* + 1], *undefined_control_sequence* − 1 − *hash_used*);
if *hash_high* > 0 **then** *dump_things*(*hash*[*eqtb_size* + 1], *hash_high*);
dump_int(*cs_count*);
print_ln; *print_int*(*cs_count*); *print*("␣multiletter␣control␣sequences")

This code is used in section 1316.

1322. ⟨Undump the hash table 1322⟩ ≡
undump(*hash_base*)(*frozen_control_sequence*)(*hash_used*); *p* ← *hash_base* − 1;
repeat *undump*(*p* + 1)(*hash_used*)(*p*); *undump_hh*(*hash*[*p*]);
until *p* = *hash_used*;
undump_things(*hash*[*hash_used* + 1], *undefined_control_sequence* − 1 − *hash_used*);
if *debug_format_file* **then**
  begin *print_csnames*(*hash_base*, *undefined_control_sequence* − 1);
  end;
if *hash_high* > 0 **then**
  begin *undump_things*(*hash*[*eqtb_size* + 1], *hash_high*);
  if *debug_format_file* **then**
    begin *print_csnames*(*eqtb_size* + 1, *hash_high* − (*eqtb_size* + 1));
    end;
  end;
undump_int(*cs_count*)

This code is used in section 1317.

1323. ⟨Dump the font information 1323⟩ ≡
dump_int(*fmem_ptr*); *dump_things*(*font_info*[0], *fmem_ptr*); *dump_int*(*font_ptr*);
⟨Dump the array info for internal font number *k* 1325⟩;
print_ln; *print_int*(*fmem_ptr* − 7); *print*("␣words␣of␣font␣info␣for␣");
print_int(*font_ptr* − *font_base*);
if *font_ptr* ≠ *font_base* + 1 **then** *print*("␣preloaded␣fonts")
else *print*("␣preloaded␣font")

This code is used in section 1305.

1324. ⟨Undump the font information 1324⟩ ≡
undump_size(7)(*sup_font_mem_size*)('font␣mem␣size')(*fmem_ptr*);
if *fmem_ptr* > *font_mem_size* **then** *font_mem_size* ← *fmem_ptr*;
font_info ← *xmalloc_array*(*fmemory_word*, *font_mem_size*); *undump_things*(*font_info*[0], *fmem_ptr*);
undump_size(*font_base*)(*font_base* + *max_font_max*)('font␣max')(*font_ptr*);
    { This undumps all of the font info, despite the name. }
⟨Undump the array info for internal font number *k* 1326⟩;

This code is used in section 1306.

1325. ⟨Dump the array info for internal font number k 1325⟩ ≡

begin *dump_things*(*font_check*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_size*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_dsize*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_params*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*hyphen_char*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*skew_char*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_name*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_area*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_bc*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_ec*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*char_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*width_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*height_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*depth_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*italic_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*lig_kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*exten_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*param_base*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_glue*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*bchar_label*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
dump_things(*font_false_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
for $k \gets$ *null_font* **to** *font_ptr* **do**
  begin *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_dsize*[k] **then**
    begin *print*("␣at␣"); *print_scaled*(*font_size*[k]); *print*("pt");
    end;
  end;
end

This code is used in section 1323.

1326. This module should now be named 'Undump all the font arrays'.

⟨ Undump the array info for internal font number k 1326 ⟩ ≡
begin { Allocate the font arrays }
font_check ← *xmalloc_array*(*four_quarters*, *font_max*); *font_size* ← *xmalloc_array*(*scaled*, *font_max*);
font_dsize ← *xmalloc_array*(*scaled*, *font_max*); *font_params* ← *xmalloc_array*(*font_index*, *font_max*);
font_name ← *xmalloc_array*(*str_number*, *font_max*); *font_area* ← *xmalloc_array*(*str_number*, *font_max*);
font_bc ← *xmalloc_array*(*eight_bits*, *font_max*); *font_ec* ← *xmalloc_array*(*eight_bits*, *font_max*);
font_glue ← *xmalloc_array*(*halfword*, *font_max*); *hyphen_char* ← *xmalloc_array*(*integer*, *font_max*);
skew_char ← *xmalloc_array*(*integer*, *font_max*); *bchar_label* ← *xmalloc_array*(*font_index*, *font_max*);
font_bchar ← *xmalloc_array*(*nine_bits*, *font_max*); *font_false_bchar* ← *xmalloc_array*(*nine_bits*, *font_max*);
char_base ← *xmalloc_array*(*integer*, *font_max*); *width_base* ← *xmalloc_array*(*integer*, *font_max*);
height_base ← *xmalloc_array*(*integer*, *font_max*); *depth_base* ← *xmalloc_array*(*integer*, *font_max*);
italic_base ← *xmalloc_array*(*integer*, *font_max*); *lig_kern_base* ← *xmalloc_array*(*integer*, *font_max*);
kern_base ← *xmalloc_array*(*integer*, *font_max*); *exten_base* ← *xmalloc_array*(*integer*, *font_max*);
param_base ← *xmalloc_array*(*integer*, *font_max*);
undump_things(*font_check*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*font_size*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*font_dsize*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(*min_halfword*, *max_halfword*, *font_params*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*hyphen_char*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*skew_char*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_upper_check_things(*str_ptr*, *font_name*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_upper_check_things(*str_ptr*, *font_area*[*null_font*], *font_ptr* + 1 − *null_font*); { There's no point in checking these values against the range [0, 255], since the data type is *unsigned char*, and all values of that type are in that range by definition. }
undump_things(*font_bc*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*font_ec*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*char_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*width_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*height_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*depth_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*italic_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*lig_kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*kern_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*exten_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_things(*param_base*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(*min_halfword*, *lo_mem_max*, *font_glue*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(0, *fmem_ptr* − 1, *bchar_label*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(*min_quarterword*, *non_char*, *font_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
undump_checked_things(*min_quarterword*, *non_char*, *font_false_bchar*[*null_font*], *font_ptr* + 1 − *null_font*);
end

This code is used in section 1324.

1327. ⟨ Dump the hyphenation tables 1327 ⟩ ≡

```
dump_int(hyph_count);
if hyph_next ≤ hyph_prime then hyph_next ← hyph_size;
dump_int(hyph_next);  { minumum value of hyphen_size needed }
for k ← 0 to hyph_size do
  if hyph_word[k] ≠ 0 then
    begin dump_int(k + 65536 ∗ hyph_link[k]);
        { assumes number of hyphen exceptions does not exceed 65535 }
    dump_int(hyph_word[k]); dump_int(hyph_list[k]);
    end;
print_ln; print_int(hyph_count);
if hyph_count ≠ 1 then print("␣hyphenation␣exceptions")
else print("␣hyphenation␣exception");
if trie_not_ready then init_trie;
dump_int(trie_max); dump_things(trie_trl[0], trie_max + 1); dump_things(trie_tro[0], trie_max + 1);
dump_things(trie_trc[0], trie_max + 1); dump_int(trie_op_ptr); dump_things(hyf_distance[1], trie_op_ptr);
dump_things(hyf_num[1], trie_op_ptr); dump_things(hyf_next[1], trie_op_ptr);
print_nl("Hyphenation␣trie␣of␣length␣"); print_int(trie_max); print("␣has␣");
print_int(trie_op_ptr);
if trie_op_ptr ≠ 1 then print("␣ops")
else print("␣op");
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 1305.

1328. Only "nonempty" parts of *op_start* need to be restored.

⟨ Undump the hyphenation tables 1328 ⟩ ≡
  undump_size(0)(*hyph_size*)(´hyph_size´)(*hyph_count*);
  undump_size(*hyph_prime*)(*hyph_size*)(´hyph_size´)(*hyph_next*); $j \leftarrow 0$;
  for $k \leftarrow 1$ **to** *hyph_count* **do**
    begin *undump_int*(j);
    if $j < 0$ **then goto** *bad_fmt*;
    if $j > 65535$ **then**
      begin *hyph_next* $\leftarrow j$ **div** 65536; $j \leftarrow j - hyph\_next * 65536$;
      end
    else *hyph_next* $\leftarrow 0$;
    if $(j \geq hyph\_size) \vee (hyph\_next > hyph\_size)$ **then goto** *bad_fmt*;
    hyph_link[j] $\leftarrow$ *hyph_next*; *undump*(0)(*str_ptr*)(*hyph_word*[j]);
    undump(*min_halfword*)(*max_halfword*)(*hyph_list*[j]);
    end; { j is now the largest occupied location in *hyph_word* }
  incr(j);
  if $j < hyph\_prime$ **then** $j \leftarrow hyph\_prime$;
  hyph_next $\leftarrow j$;
  if *hyph_next* $\geq$ *hyph_size* **then** *hyph_next* $\leftarrow$ *hyph_prime*
  else if *hyph_next* $\geq$ *hyph_prime* **then** *incr*(*hyph_next*);
  undump_size(0)(*trie_size*)(´trie␣size´)(j); **init** *trie_max* $\leftarrow j$; **tini**
    { These first three haven't been allocated yet unless we're INITEX; we do that precisely so we don't allocate more space than necessary. }
  if ¬*trie_trl* **then** *trie_trl* $\leftarrow$ *xmalloc_array*(*trie_pointer*, $j + 1$);
  undump_things(*trie_trl*[0], $j + 1$);
  if ¬*trie_tro* **then** *trie_tro* $\leftarrow$ *xmalloc_array*(*trie_pointer*, $j + 1$);
  undump_things(*trie_tro*[0], $j + 1$);
  if ¬*trie_trc* **then** *trie_trc* $\leftarrow$ *xmalloc_array*(*quarterword*, $j + 1$);
  undump_things(*trie_trc*[0], $j + 1$);
  undump_size(0)(*trie_op_size*)(´trie␣op␣size´)(j); **init** *trie_op_ptr* $\leftarrow j$; **tini**
    { I'm not sure we have such a strict limitation (64) on these values, so let's leave them unchecked. }
  undump_things(*hyf_distance*[1], j); *undump_things*(*hyf_num*[1], j);
  undump_upper_check_things(*max_trie_op*, *hyf_next*[1], j);
  init for $k \leftarrow 0$ **to** 255 **do** *trie_used*[k] $\leftarrow$ *min_quarterword*;
  tini
  $k \leftarrow 256$;
  while $j > 0$ **do**
    begin *undump*(0)($k - 1$)(k); *undump*(1)(j)(x); **init** *trie_used*[k] $\leftarrow$ *qi*(x); **tini**
    $j \leftarrow j - x$; *op_start*[k] $\leftarrow$ *qo*(j);
    end;
  init *trie_not_ready* $\leftarrow$ *false* **tini**

This code is used in section 1306.

1329. We have already printed a lot of statistics, so we set *tracing_stats* $\leftarrow 0$ to prevent them from appearing again.

⟨ Dump a couple more things and the closing check word 1329 ⟩ ≡
  dump_int(*interaction*); *dump_int*(*format_ident*); *dump_int*(69069); *tracing_stats* $\leftarrow 0$

This code is used in section 1305.

1330. ⟨ Undump a couple more things and the closing check word 1330 ⟩ ≡
undump(*batch_mode*)(*error_stop_mode*)(*interaction*);
if *interaction_option* $\neq$ *unspecified_mode* **then** *interaction* $\leftarrow$ *interaction_option*;
undump(0)(*str_ptr*)(*format_ident*); *undump_int*(x);
if $x \neq 69069$ **then goto** *bad_fmt*

This code is used in section 1306.

1331. ⟨ Create the *format_ident*, open the format file, and inform the user that dumping has begun 1331 ⟩ ≡
selector $\leftarrow$ *new_string*; *print*("␣(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* $\leftarrow$ *log_only*
else *selector* $\leftarrow$ *term_and_log*;
str_room(1); *format_ident* $\leftarrow$ *make_string*; *pack_job_name*(*format_extension*);
while $\neg$*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 1305.

1332. ⟨ Close the format file 1332 ⟩ ≡
w_close(*fmt_file*)

This code is used in section 1305.

1333. 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 1336 ⟩

1334. We have noted that there are two versions of TEX82. 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 *ready_already* ← 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 *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 *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 }

1335. 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."

define *const_chk*(#) ≡
  begin if # < *inf*@&# **then** # ← *inf*@&#
  else if # > *sup*@&# **then** # ← *sup*@&#
  end { *setup_bound_var* stuff duplicated in `mf.ch`. }
define *setup_bound_var*(#) ≡ *bound_default* ← #; *setup_bound_var_end*
define *setup_bound_var_end*(#) ≡ *bound_name* ← #; *setup_bound_var_end_end*
define *setup_bound_var_end_end*(#) ≡ *setup_bound_variable*(*addressof*(#), *bound_name*, *bound_default*)

procedure *main_body*;
begin { *start_here* }
{ Bounds that may be set from the configuration file. We want the user to be able to specify the names with underscores, but `TANGLE` removes underscores, so we're stuck giving the names twice, once as a string, once as the identifier. How ugly. }
setup_bound_var(0)(´`mem_bot`´)(*mem_bot*); *setup_bound_var*(250000)(´`main_memory`´)(*main_memory*);
  { *memory_word*s for *mem* in `INITEX` }
setup_bound_var(0)(´`extra_mem_top`´)(*extra_mem_top*); { increase high mem in `VIRTEX` }
setup_bound_var(0)(´`extra_mem_bot`´)(*extra_mem_bot*); { increase low mem in `VIRTEX` }
setup_bound_var(200000)(´`pool_size`´)(*pool_size*);
setup_bound_var(75000)(´`string_vacancies`´)(*string_vacancies*);
setup_bound_var(5000)(´`pool_free`´)(*pool_free*); { min pool avail after fmt }
setup_bound_var(15000)(´`max_strings`´)(*max_strings*);
setup_bound_var(100)(´`strings_free`´)(*strings_free*);
setup_bound_var(100000)(´`font_mem_size`´)(*font_mem_size*);
setup_bound_var(500)(´`font_max`´)(*font_max*); *setup_bound_var*(20000)(´`trie_size`´)(*trie_size*);
  { if *ssup_trie_size* increases, recompile }
setup_bound_var(659)(´`hyph_size`´)(*hyph_size*); *setup_bound_var*(3000)(´`buf_size`´)(*buf_size*);
setup_bound_var(50)(´`nest_size`´)(*nest_size*); *setup_bound_var*(15)(´`max_in_open`´)(*max_in_open*);
setup_bound_var(60)(´`param_size`´)(*param_size*); *setup_bound_var*(4000)(´`save_size`´)(*save_size*);
setup_bound_var(300)(´`stack_size`´)(*stack_size*);
setup_bound_var(16384)(´`dvi_buf_size`´)(*dvi_buf_size*); *setup_bound_var*(79)(´`error_line`´)(*error_line*);
setup_bound_var(50)(´`half_error_line`´)(*half_error_line*);
setup_bound_var(79)(´`max_print_line`´)(*max_print_line*);
setup_bound_var(0)(´`hash_extra`´)(*hash_extra*);
setup_bound_var(10000)(´`expand_depth`´)(*expand_depth*); *const_chk*(*mem_bot*);
const_chk(*main_memory*); **Init** *extra_mem_top* ← 0; *extra_mem_bot* ← 0; **Tini**
if *extra_mem_bot* > *sup_main_memory* **then** *extra_mem_bot* ← *sup_main_memory*;
if *extra_mem_top* > *sup_main_memory* **then** *extra_mem_top* ← *sup_main_memory*;
  { *mem_top* is an index, *main_memory* a size }
mem_top ← *mem_bot* + *main_memory* − 1; *mem_min* ← *mem_bot*; *mem_max* ← *mem_top*;
  { Check other constants against their sup and inf. }
const_chk(*trie_size*); *const_chk*(*hyph_size*); *const_chk*(*buf_size*); *const_chk*(*nest_size*);
const_chk(*max_in_open*); *const_chk*(*param_size*); *const_chk*(*save_size*); *const_chk*(*stack_size*);
const_chk(*dvi_buf_size*); *const_chk*(*pool_size*); *const_chk*(*string_vacancies*); *const_chk*(*pool_free*);
const_chk(*max_strings*); *const_chk*(*strings_free*); *const_chk*(*font_mem_size*); *const_chk*(*font_max*);
const_chk(*hash_extra*);
if *error_line* > *ssup_error_line* **then** *error_line* ← *ssup_error_line*; { array memory allocation }
buffer ← *xmalloc_array*(*ASCII_code*, *buf_size*); *nest* ← *xmalloc_array*(*list_state_record*, *nest_size*);
save_stack ← *xmalloc_array*(*memory_word*, *save_size*);
input_stack ← *xmalloc_array*(*in_state_record*, *stack_size*);
input_file ← *xmalloc_array*(*alpha_file*, *max_in_open*); *line_stack* ← *xmalloc_array*(*integer*, *max_in_open*);

```
source_filename_stack ← xmalloc_array(str_number, max_in_open);
full_source_filename_stack ← xmalloc_array(str_number, max_in_open);
param_stack ← xmalloc_array(halfword, param_size); dvi_buf ← xmalloc_array(eight_bits, dvi_buf_size);
hyph_word ← xmalloc_array(str_number, hyph_size);
hyph_list ← xmalloc_array(halfword, hyph_size); hyph_link ← xmalloc_array(hyph_pointer, hyph_size);
    Init yzmem ← xmalloc_array(memory_word, mem_top − mem_bot + 1);
zmem ← yzmem − mem_bot; { Some compilers require mem_bot = 0 }
eqtb_top ← eqtb_size + hash_extra;
if hash_extra = 0 then hash_top ← undefined_control_sequence
else hash_top ← eqtb_top;
yhash ← xmalloc_array(two_halves, 1 + hash_top − hash_offset); hash ← yhash − hash_offset;
    { Some compilers require hash_offset = 0 }
next(hash_base) ← 0; text(hash_base) ← 0;
for hash_used ← hash_base + 1 to hash_top do hash[hash_used] ← hash[hash_base];
zeqtb ← xmalloc_array(memory_word, eqtb_top); eqtb ← zeqtb;
str_start ← xmalloc_array(pool_pointer, max_strings);
str_pool ← xmalloc_array(packed_ASCII_code, pool_size);
font_info ← xmalloc_array(fmemory_word, font_mem_size); Tinihistory ← 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 1340 ⟩;
history ← spotless; { ready to go! }
main_control; { come to life }
final_cleanup; { prepare for death }
close_files_and_terminate;
final_end: do_final_end;
end { main_body }
;
```

1336. Here we do whatever is needed to complete TeX'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.

This program doesn't bother to close the input files that may still be open.

```
⟨Last-minute procedures 1336⟩ ≡
procedure close_files_and_terminate;
  var k: integer;  { all-purpose index }
  begin ⟨Finish the extensions 1381⟩;
  stat if tracing_stats > 0 then ⟨Output statistics about this job 1337⟩; tats
  wake_up_terminal; ⟨Finish the DVI file 645⟩;
  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␣"); print_file_name(0, log_name, 0); print_char(".");
      end;
    end;
  print_ln;
  if (edit_name_start ≠ 0) ∧ (interaction > batch_mode) then
    call_edit(str_pool, edit_name_start, edit_name_length, edit_line);
  end;
```

See also sections 1338, 1339, and 1341.

This code is used in section 1333.

1337. 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 TeX is being used.

```
⟨Output statistics about this job 1337⟩ ≡
  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(´␣out␣of␣´, 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, ´+´,
        hash_extra : 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 1336.

1338. We get to the *final_cleanup* routine when `\end` or `\dump` has been scanned and *its_all_over*.

```
⟨ Last-minute procedures 1336 ⟩ +≡
procedure final_cleanup;
  label exit;
  var c: small_number; { 0 for \end, 1 for \dump }
  begin c ← cur_chr;
  if job_name = 0 then open_log_file;
  while input_ptr > 0 do
    if state = token_list then end_token_list else end_file_reading;
  while open_parens > 0 do
    begin print("␣)"); decr(open_parens);
    end;
  if cur_level > level_one then
    begin print_nl("("); print_esc("end␣occurred␣"); print("inside␣a␣group␣at␣level␣");
    print_int(cur_level − level_one); print_char(")");
    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 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;
```

1339.

```
⟨ Last-minute procedures 1336 ⟩ +≡
  init procedure init_prim; { initialize all the primitives }
  begin no_new_control_sequence ← false; ⟨ Put each of TEX's primitives into the hash table 226 ⟩;
  no_new_control_sequence ← true;
  end;
  tini
```

1340. 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 1340⟩ ≡
  begin ⟨Initialize the input routines 331⟩;
  if (format_ident = 0) ∨ (buffer[loc] = "&") ∨ dump_line 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); eqtb ← zeqtb;
    while (loc < limit) ∧ (buffer[loc] = "␣") do incr(loc);
    end;
  if end_line_char_inactive then decr(limit)
  else buffer[limit] ← end_line_char;
  if mltex_enabled_p then
    begin wterm_ln(´MLTeX␣v2.2␣enabled´);
    end;
  fix_date_and_time;
  init if trie_not_ready then
    begin {initex without format loaded}
    trie_trl ← xmalloc_array(trie_pointer, trie_size); trie_tro ← xmalloc_array(trie_pointer, trie_size);
    trie_trc ← xmalloc_array(quarterword, trie_size);
    trie_c ← xmalloc_array(packed_ASCII_code, trie_size); trie_o ← xmalloc_array(trie_opcode, trie_size);
    trie_l ← xmalloc_array(trie_pointer, trie_size); trie_r ← xmalloc_array(trie_pointer, trie_size);
    trie_hash ← xmalloc_array(trie_pointer, trie_size); trie_taken ← xmalloc_array(boolean, trie_size);
    trie_root ← 0; trie_c[0] ← si(0); trie_ptr ← 0; {Allocate and initialize font arrays}
    font_check ← xmalloc_array(four_quarters, font_max); font_size ← xmalloc_array(scaled, font_max);
    font_dsize ← xmalloc_array(scaled, font_max); font_params ← xmalloc_array(font_index, font_max);
    font_name ← xmalloc_array(str_number, font_max);
    font_area ← xmalloc_array(str_number, font_max); font_bc ← xmalloc_array(eight_bits, font_max);
    font_ec ← xmalloc_array(eight_bits, font_max); font_glue ← xmalloc_array(halfword, font_max);
    hyphen_char ← xmalloc_array(integer, font_max); skew_char ← xmalloc_array(integer, font_max);
    bchar_label ← xmalloc_array(font_index, font_max); font_bchar ← xmalloc_array(nine_bits, font_max);
    font_false_bchar ← xmalloc_array(nine_bits, font_max); char_base ← xmalloc_array(integer, font_max);
    width_base ← xmalloc_array(integer, font_max); height_base ← xmalloc_array(integer, font_max);
    depth_base ← xmalloc_array(integer, font_max); italic_base ← xmalloc_array(integer, font_max);
    lig_kern_base ← xmalloc_array(integer, font_max); kern_base ← xmalloc_array(integer, font_max);
    exten_base ← xmalloc_array(integer, font_max); param_base ← xmalloc_array(integer, font_max);
    font_ptr ← null_font; fmem_ptr ← 7; font_name[null_font] ← "nullfont"; font_area[null_font] ← "";
    hyphen_char[null_font] ← "-"; skew_char[null_font] ← −1; bchar_label[null_font] ← non_address;
    font_bchar[null_font] ← non_char; font_false_bchar[null_font] ← non_char; font_bc[null_font] ← 1;
    font_ec[null_font] ← 0; font_size[null_font] ← 0; font_dsize[null_font] ← 0; char_base[null_font] ← 0;
    width_base[null_font] ← 0; height_base[null_font] ← 0; depth_base[null_font] ← 0;
    italic_base[null_font] ← 0; lig_kern_base[null_font] ← 0; kern_base[null_font] ← 0;
    exten_base[null_font] ← 0; font_glue[null_font] ← null; font_params[null_font] ← 7;
    param_base[null_font] ← −1;
    for font_k ← 0 to 6 do font_info[font_k].sc ← 0;
    end;
  tini
```

$font\_used \gets xmalloc\_array(boolean, font\_max)$;
for $font\_k \gets font\_base$ **to** $font\_max$ **do** $font\_used[font\_k] \gets false$;
⟨ Compute the magic offset 768 ⟩;
⟨ Initialize the print *selector* based on *interaction* 75 ⟩;
if $(loc < limit) \land (cat\_code(buffer[loc]) \neq escape)$ **then** *start_input*; { `\input` assumed }
end

This code is used in section 1335.

1341. 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 1336 ⟩ +≡
debug procedure *debug_help*; { routine to display various things }
label *breakpoint*, *exit*;
var k, l, m, n: *integer*;
begin 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** *dump_core* { do something to cause a core dump }
    else begin *read*(*term_in*, n);
      case m **of**
      ⟨ Numbered cases for *debug_help* 1342 ⟩
      othercases *print*("`?`")
      endcases;
      end;
  end;
exit: **end**;
gubed

1342. ⟨ Numbered cases for *debug_help* 1342 ⟩ ≡

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: **begin** *print_scaled*(*font_info*[n].*sc*); *print_char*("␣");
   print_int(*font_info*[n].*qqqq*.*b0*); *print_char*(":");
   print_int(*font_info*[n].*qqqq*.*b1*); *print_char*(":");
   print_int(*font_info*[n].*qqqq*.*b2*); *print_char*(":");
   print_int(*font_info*[n].*qqqq*.*b3*);
   end;
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* ← 10000; *depth_threshold* ← *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* ← *null_font*; *cfont_in_short_display* ← *null_cfont*; *short_display*(n);
   end;
16: *panicking* ← ¬*panicking*;

This code is used in section 1341.

1343. 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.

1344. 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 *language_node* = 4 { *subtype* in whatsits that change the current language }
define $what\_lang(\#) \equiv link(\# + 1)$ { language number, in the range 0 .. 255 }
define $what\_lhm(\#) \equiv type(\# + 1)$ { minimum left fragment, in the range 1 .. 63 }
define $what\_rhm(\#) \equiv subtype(\# + 1)$ { minimum right fragment, in the range 1 .. 63 }
define $write\_tokens(\#) \equiv link(\# + 1)$ { reference count of token list to write }
define $write\_stream(\#) \equiv info(\# + 1)$ { stream number (0 to 17) }
define $open\_name(\#) \equiv link(\# + 1)$ { string number of file name to open }
define $open\_area(\#) \equiv info(\# + 2)$ { string number of file area for *open_name* }
define $open\_ext(\#) \equiv link(\# + 2)$ { string number of file extension for *open_name* }

1345. The sixteen possible `\write` streams are represented by the *write_file* array. The jth 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*;

1346. ⟨ Set initial values of key variables 21 ⟩ +≡
for $k \gets 0$ **to** 17 **do** *write_open*[k] ← *false*;

1347. 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* = 4 { command modifier for `\immediate` }
define *set_language_code* = 5 { command modifier for `\setlanguage` }

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive(`"openout"`, *extension*, *open_node*);
primitive(`"write"`, *extension*, *write_node*); *write_loc* ← *cur_val*;
primitive(`"closeout"`, *extension*, *close_node*);
primitive(`"special"`, *extension*, *special_node*);
text(*frozen_special*) ← `"special"`; *eqtb*[*frozen_special*] ← *eqtb*[*cur_val*];
primitive(`"immediate"`, *extension*, *immediate_code*);
primitive(`"setlanguage"`, *extension*, *set_language_code*);

1348. 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` }

1349. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
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"`);
othercases *print*(`"[unknown␣extension!]"`)
endcases;

1350. 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 TEX 1350 ⟩ ≡
any_mode(*extension*): *do_extension*;

This code is used in section 1048.

1351. ⟨ Declare action procedures for use by *main_control* 1046 ⟩ +≡
⟨ Declare procedures needed in *do_extension* 1352 ⟩
procedure *do_extension*;
  var *k*: *integer*; { all-purpose integers }
    p: *pointer*; { all-purpose pointers }
  begin case *cur_chr* **of**
  open_node: ⟨ Implement `\openout` 1354 ⟩;
  write_node: ⟨ Implement `\write` 1355 ⟩;
  close_node: ⟨ Implement `\closeout` 1356 ⟩;
  special_node: ⟨ Implement `\special` 1357 ⟩;
  immediate_code: ⟨ Implement `\immediate` 1378 ⟩;
  set_language_code: ⟨ Implement `\setlanguage` 1380 ⟩;
  othercases *confusion*("ext1")
  endcases;
  end;

1352. 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* 1352 ⟩ ≡
procedure *new_whatsit*(*s* : *small_number*; *w* : *small_number*);
  var *p*: *pointer*; { the new node }
  begin $p \leftarrow$ *get_node*(*w*); *type*(*p*) ← *whatsit_node*; *subtype*(*p*) ← *s*; *link*(*tail*) ← *p*; *tail* ← *p*;
  end;

See also section 1353.

This code is used in section 1351.

1353. 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* 1352 ⟩ +≡
procedure *new_write_whatsit*(*w* : *small_number*);
  begin *new_whatsit*(*cur_chr*, *w*);
  if $w \neq$ *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) ∧ (*cur_val* $\neq 18$) **then** *cur_val* ← 16;
    end;
  write_stream(*tail*) ← *cur_val*;
  end;

1354. ⟨ Implement `\openout` 1354 ⟩ ≡
  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 1351.

1355. 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` 1355 ⟩ ≡
  begin $k \leftarrow cur\_cs$; $new\_write\_whatsit(write\_node\_size)$;
  $cur\_cs \leftarrow k$; $p \leftarrow scan\_toks(false, false)$; $write\_tokens(tail) \leftarrow def\_ref$;
  end

This code is used in section 1351.

1356. ⟨ Implement `\closeout` 1356 ⟩ ≡
  begin $new\_write\_whatsit(write\_node\_size)$; $write\_tokens(tail) \leftarrow null$;
  end

This code is used in section 1351.

1357. When '`\special{...}`' appears, we expand the macros in the token list as in `\xdef` and `\mark`.

⟨ Implement `\special` 1357 ⟩ ≡
  begin $new\_whatsit(special\_node, write\_node\_size)$; $write\_stream(tail) \leftarrow null$; $p \leftarrow scan\_toks(false, true)$;
  $write\_tokens(tail) \leftarrow def\_ref$;
  end

This code is used in section 1351.

1358. 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;

1359. ⟨ Display the whatsit node p 1359 ⟩ ≡
  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;
  $language\_node$: **begin** $print\_esc($"`setlanguage`"$)$; $print\_int(what\_lang(p))$; $print($"`␣(hyphenmin␣`"$)$;
    $print\_int(what\_lhm(p))$; $print\_char($"`,`"$)$; $print\_int(what\_rhm(p))$; $print\_char($"`)`"$)$;
    end;
  othercases $print($"`whatsit?`"$)$
  endcases

This code is used in section 183.

1360. ⟨ 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 1360 ⟩ ≡

case *subtype*(p) **of**
open_node: **begin** $r \leftarrow$ *get_node*(*open_node_size*); *words* $\leftarrow$ *open_node_size*;
end;
write_node, *special_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;
othercases *confusion*("ext2")
endcases

This code is used in section 206.

1361. ⟨ Wipe out the whatsit node p and **goto** *done* 1361 ⟩ ≡

begin case *subtype*(p) **of**
open_node: *free_node*(p, *open_node_size*);
write_node, *special_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*);
othercases *confusion*("ext3")
endcases;
goto *done*;
end

This code is used in section 202.

1362. ⟨ Incorporate a whatsit node into a vbox 1362 ⟩ ≡
do_nothing

This code is used in section 672.

1363. ⟨ Incorporate a whatsit node into an hbox 1363 ⟩ ≡
do_nothing

This code is used in section 654.

1364. ⟨ Let d be the width of the whatsit p 1364 ⟩ ≡
$d \leftarrow 0$

This code is used in section 1150.

1365. **define** *adv_past*(#) ≡ **if** *subtype*(#) = *language_node* **then**
begin *cur_lang* $\leftarrow$ *what_lang*(#); *l_hyf* $\leftarrow$ *what_lhm*(#); *r_hyf* $\leftarrow$ *what_rhm*(#); **end**

⟨ Advance past a whatsit node in the *line_break* loop 1365 ⟩ ≡ *adv_past*(*cur_p*)

This code is used in section 869.

1366. ⟨ Advance past a whatsit node in the pre-hyphenation loop 1366 ⟩ ≡ *adv_past*(s)

This code is used in section 899.

1367. ⟨ Prepare to move whatsit p to the current page, then **goto** *contribute* 1367 ⟩ ≡
goto *contribute*

This code is used in section 1003.

1368. ⟨Process whatsit p in *vert_break* loop, **goto** *not_found* 1368⟩ ≡
goto *not_found*

This code is used in section 976.

1369. ⟨Output the whatsit node p in a vlist 1369⟩ ≡
out_what(p)

This code is used in section 634.

1370. ⟨Output the whatsit node p in an hlist 1370⟩ ≡
out_what(p)

This code is used in section 625.

1371. 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).

⟨Declare procedures needed in *hlist_out*, *vlist_out* 1371⟩ ≡
procedure *special_out*(p : *pointer*);
  var *old_setting*: 0 .. *max_selector*; { holds print *selector* }
    k: *pool_pointer*; { index into *str_pool* }
  begin *synch_h*; *synch_v*;
  old_setting ← *selector*; *selector* ← *new_string*;
  show_token_list(*link*(*write_tokens*(p)), *null*, *pool_size* − *pool_ptr*); *selector* ← *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 ← *str_start*[*str_ptr*] **to** *pool_ptr* − 1 **do** *dvi_out*(*so*(*str_pool*[k]));
  pool_ptr ← *str_start*[*str_ptr*]; { erase the string }
  end;

See also sections 1373 and 1376.

This code is used in section 622.

1372. 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`.

⟨Initialize table entries (done by `INITEX` only) 164⟩ +≡
  text(*end_write*) ← `"endwrite"`; *eq_level*(*end_write*) ← *level_one*; *eq_type*(*end_write*) ← *outer_call*;
  equiv(*end_write*) ← *null*;

1373. ⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1371 ⟩ +≡

```
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 }
    d: integer; { number of characters in incomplete current string }
    clobbered: boolean; { system string is ok? }
    runsystem_ret: integer; { return value from runsystem }
  begin ⟨ Expand macros in the token list and make link(def_ref) point to the result 1374 ⟩;
  old_setting ← selector; j ← write_stream(p);
  if j = 18 then selector ← new_string
  else 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);
  if j = 18 then
    begin if (tracing_online ≤ 0) then selector ← log_only { Show what we're doing in the log file. }
    else selector ← term_and_log; { Show what we're doing. }
        { If the log file isn't open yet, we can only send output to the terminal. Calling open_log_file from
          here seems to result in bad data in the log. }
    if ¬log_opened then selector ← term_only;
    print_nl("runsystem(");
    for d ← 0 to cur_length − 1 do
      begin { print gives up if passed str_ptr, so do it by hand. }
      print(so(str_pool[str_start[str_ptr] + d])); { N.B.: not print_char }
      end;
    print(")...");
    if shellenabledp then
      begin str_room(1); append_char(0); { Append a null byte to the expansion. }
      clobbered ← false;
      for d ← 0 to cur_length − 1 do { Convert to external character set. }
        begin str_pool[str_start[str_ptr] + d] ← xchr[str_pool[str_start[str_ptr] + d]];
        if (str_pool[str_start[str_ptr] + d] = null_code) ∧ (d < cur_length − 1) then clobbered ← true;
              { minimal checking: NUL not allowed in argument string of system() }
        end;
      if clobbered then print("clobbered")
      else begin { We have the command. See if we're allowed to execute it, and report in the log. We
            don't check the actual exit status of the command, or do anything with the output. }
        runsystem_ret ← runsystem(conststringcast(addressof(str_pool[str_start[str_ptr]])));
        if runsystem_ret = −1 then print("quotation␣error␣in␣system␣command")
        else if runsystem_ret = 0 then print("disabled␣(restricted)")
          else if runsystem_ret = 1 then print("executed")
            else if runsystem_ret = 2 then print("executed␣safely␣(allowed)")
        end;
      end
    else begin print("disabled"); { shellenabledp false }
      end;
    print_char("."); print_nl(""); print_ln; pool_ptr ← str_start[str_ptr]; { erase the string }
    end;
```

selector ← *old_setting*;
end;

1374. 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 1374⟩ ≡
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 1375⟩;
mode ← *old_mode*; *end_token_list* { conserve stack space }

This code is used in section 1373.

1375. ⟨Recover from an unbalanced write command 1375⟩ ≡
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 1374.

1376. The *out_what* procedure takes care of outputting whatsit nodes for *vlist_out* and *hlist_out*.

⟨Declare procedures needed in *hlist_out*, *vlist_out* 1371⟩ +≡
procedure *out_what*(*p* : *pointer*);
var *j*: *small_number*; { write stream number }
old_setting: 0 . . *max_selector*;
begin case *subtype*(*p*) **of**
open_node, *write_node*, *close_node*: ⟨Do some work that has been queued up for `\write` 1377⟩;
special_node: *special_out*(*p*);
language_node: *do_nothing*;
othercases *confusion*(`"ext4"`)
endcases;
end;

1377. 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 1377⟩ ≡
  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 ¬kpse_out_name_ok(stringcast(name_of_file + 1)) ∨ ¬a_open_out(write_file[j]) do
            prompt_file_name("output␣file␣name", ".tex");
          write_open[j] ← true;  { If on first line of input, log file is not ready yet, so don't log. }
          if log_opened then
            begin old_setting ← selector;
            if (tracing_online ≤ 0) then selector ← log_only  { Show what we're doing in the log file. }
            else selector ← term_and_log;  { Show what we're doing. }
            print_nl("\openout"); print_int(j); print("␣=␣`");
            print_file_name(cur_name, cur_area, cur_ext); print("´."); print_nl(""); print_ln;
            selector ← old_setting;
            end;
          end;
      end;
    end
```

This code is used in section 1376.

1378. 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 1378⟩ ≡
  begin get_x_token;
  if (cur_cmd = extension) ∧ (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 back_input;
  end
```

This code is used in section 1351.

1379. 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* 1046 ⟩ +≡
procedure *fix_language*;
  var *l*: *ASCII_code*; { the new current language }
  begin if *language* ≤ 0 **then** $l \leftarrow 0$
  else if *language* > 255 **then** $l \leftarrow 0$
    else $l \leftarrow$ *language*;
  if $l \neq$ *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;

1380. ⟨ Implement `\setlanguage` 1380 ⟩ ≡
  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 1351.

1381. ⟨ Finish the extensions 1381 ⟩ ≡
  for $k \leftarrow 0$ **to** 15 **do**
    if *write_open*[*k*] **then** *a_close*(*write_file*[*k*])

This code is used in section 1336.

1382. System-dependent changes for Web2c. Here are extra variables for Web2c. (This numbering of the system-dependent section allows easy integration of Web2c and e-TEX, etc.)

⟨ Global variables 13 ⟩ +≡
edit_name_start: *pool_pointer*; { where the filename to switch to starts }
edit_name_length, *edit_line*: *integer*; { what line to start editing at }
ipc_on: *cinttype*; { level of IPC action, 0 for none [default] }
stop_at_space: *boolean*; { whether *more_name* returns false for space }

1383. The *edit_name_start* will be set to point into *str_pool* somewhere after its beginning if TEX is supposed to switch to an editor on exit.

⟨ Set initial values of key variables 21 ⟩ +≡
edit_name_start ← 0; *stop_at_space* ← *true*;

1384. These are used when we regenerate the representation of the first 256 strings.

⟨ Global variables 13 ⟩ +≡
save_str_ptr: *str_number*;
save_pool_ptr: *pool_pointer*;
shellenabledp: *cinttype*;
restrictedshell: *cinttype*;
output_comment: ↑*char*;
k, *l*: 0 . . 255; { used by 'Make the first 256 strings', etc. }

1385. When debugging a macro package, it can be useful to see the exact control sequence names in the format file. For example, if ten new csnames appear, it's nice to know what they are, to help pinpoint where they came from. (This isn't a truly "basic" printing procedure, but that's a convenient module in which to put it.)

⟨ Basic printing procedures 57 ⟩ +≡
```
procedure print_csnames(hstart : integer; hfinish : integer);
  var c, h: integer;
  begin write_ln(stderr, ´fmtdebug:csnames␣from␣´, hstart, ´␣to␣´, hfinish, ´:´);
  for h ← hstart to hfinish do
    begin if text(h) > 0 then
      begin { if have anything at this position }
      for c ← str_start[text(h)] to str_start[text(h) + 1] − 1 do
        begin put_byte(str_pool[c], stderr); { print the characters }
        end;
      write_ln(stderr, ´|´);
      end;
    end;
  end;
```

1386. Are we printing extra info as we read the format file?

⟨ Global variables 13 ⟩ +≡
debug_format_file: *boolean*;

1387. A helper for printing file:line:error style messages. Look for a filename in *full_source_filename_stack*, and if we fail to find one fall back on the non-file:line:error style.

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_file_line*;
  var *level*: 0 . . *max_in_open*;
  begin *level* ← *in_open*;
  while (*level* > 0) ∧ (*full_source_filename_stack*[*level*] = 0) **do** *decr*(*level*);
  if *level* = 0 **then** *print_nl*("!␣")
  else begin *print_nl*(""); *print*(*full_source_filename_stack*[*level*]); *print*(":");
    if *level* = *in_open* **then** *print_int*(*line*)
    else *print_int*(*line_stack*[*level* + 1]);
    print(":␣");
    end;
  end;

1388. To be able to determine whether `\write18` is enabled from within TEX we also implement `\eof18`. We sort of cheat by having an additional route *scan_four_bit_int_or_18* which is the same as *scan_four_bit_int* except it also accepts the value 18.

⟨ Declare procedures that scan restricted classes of integers 436 ⟩ +≡
procedure *scan_four_bit_int_or_18*;
  begin *scan_int*;
  if (*cur_val* < 0) ∨ ((*cur_val* > 15) ∧ (*cur_val* ≠ 18)) **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;

1389. Dumping the *xord*, *xchr*, and *xprn* arrays. We dump these always in the format, so a TCX file loaded during format creation can set a default for users of the format.

⟨ Dump *xord*, *xchr*, and *xprn* 1389 ⟩ ≡
  dump_things(*xord*[0], 256); *dump_things*(*xchr*[0], 256); *dump_things*(*xprn*[0], 256);

This code is used in section 1310.

1390. Undumping the *xord*, *xchr*, and *xprn* arrays. This code is more complicated, because we want to ensure that a TCX file specified on the command line will override whatever is in the format. Since the tcx file has already been loaded, that implies throwing away the data in the format. Also, if no *translate_filename* is given, but *eight_bit_p* is set we have to make all characters printable.

⟨ Undump *xord*, *xchr*, and *xprn* 1390 ⟩ ≡
  if *translate_filename* **then**
    begin for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xord*, 1);
    for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xchr*, 1);
    for *k* ← 0 **to** 255 **do** *undump_things*(*dummy_xprn*, 1);
    end
  else begin *undump_things*(*xord*[0], 256); *undump_things*(*xchr*[0], 256); *undump_things*(*xprn*[0], 256);
    if *eight_bit_p* **then**
      for *k* ← 0 **to** 255 **do** *xprn*[*k*] ← 1;
    end;

This code is used in section 1311.

1391. The string recycling routines. TEX uses 2 upto 4 *new* strings when scanning a filename in an `\input`, `\openin`, or `\openout` operation. These strings are normally lost because the reference to them are not saved after finishing the operation. *search_string* searches through the string pool for the given string and returns either 0 or the found string number.

⟨ Declare additional routines for string recycling 1391 ⟩ ≡
function *search_string*(*search* : *str_number*): *str_number*;
  label *found*;
  var *result*: *str_number*; *s*: *str_number*; { running index }
    len: *integer*; { length of searched string }
  begin *result* ← 0; *len* ← *length*(*search*);
  if *len* = 0 **then** { trivial case }
    begin *result* ← ""; **goto** *found*;
    end
  else begin *s* ← *search* − 1; { start search with newest string below *s*; *search* > 1! }
    while *s* > 255 **do** { first 256 strings depend on implementation!! }
      begin if *length*(*s*) = *len* **then**
        if *str_eq_str*(*s*, *search*) **then**
          begin *result* ← *s*; **goto** *found*;
          end;
      decr(*s*);
      end;
    end;
found: *search_string* ← *result*;
  end;

See also section 1392.

This code is used in section 47.

1392. The following routine is a variant of *make_string*. It searches the whole string pool for a string equal to the string currently built and returns a found string. Otherwise a new string is created and returned. Be cautious, you can not apply *flush_string* to a replaced string!

⟨ Declare additional routines for string recycling 1391 ⟩ +≡
function *slow_make_string*: *str_number*;
  label *exit*;
  var *s*: *str_number*; { result of *search_string* }
    t: *str_number*; { new string }
  begin *t* ← *make_string*; *s* ← *search_string*(*t*);
  if *s* > 0 **then**
    begin *flush_string*; *slow_make_string* ← *s*; **return**;
    end;
  slow_make_string ← *t*;
exit: **end**;

1393. System-dependent changes for MLTEX.

The boolean variable *mltex_p* is set by web2c according to the given command line option (or an entry in the configuration file) before any TEX function is called.

⟨ Global variables 13 ⟩ +≡
mltex_p: *boolean*;

1394. The boolean variable *mltex_enabled_p* is used to enable MLTEX's character substitution. It is initialised to *false*. When loading a `FMT` it is set to the value of the boolean *mltex_p* saved in the `FMT` file. Additionally it is set to the value of *mltex_p* in IniTEX.

⟨ Global variables 13 ⟩ +≡
mltex_enabled_p: *boolean*; { enable character substitution }

1395. ⟨ Set initial values of key variables 21 ⟩ +≡
mltex_enabled_p ← *false*;

1396. The function *effective_char* computes the effective character with respect to font information. The effective character is either the base character part of a character substitution definition, if the character does not exist in the font or the character itself.

Inside *effective_char* we can not use *char_info* because the macro *char_info* uses *effective_char* calling this function a second time with the same arguments.

If neither the character c exists in font f nor a character substitution for c was defined, you can not use the function value as a character offset in *char_info* because it will access an undefined or invalid *font_info* entry! Therefore inside *char_info* and in other places, *effective_char*'s boolean parameter *err_p* is set to *true* to issue a warning and return the incorrect replacement, but always existing character *font_bc*[f].

⟨ Declare additional functions for MLTEX 1396 ⟩ ≡
function *effective_char*(*err_p* : *boolean*; f : *internal_font_number*; c : *quarterword*): *integer*;
  label *found*;
  var *base_c*: *integer*; { or *eightbits*: replacement base character }
    result: *integer*; { or *quarterword* }
  begin *result* ← c; { return c unless it does not exist in the font }
  if ¬*mltex_enabled_p* **then goto** *found*;
  if *font_ec*[f] ≥ *qo*(c) **then**
    if *font_bc*[f] ≤ *qo*(c) **then**
      if *char_exists*(*orig_char_info*(f)(c)) **then** { N.B.: not *char_info*(f)(c) }
        goto *found*;
  if *qo*(c) ≥ *char_sub_def_min* **then**
    if *qo*(c) ≤ *char_sub_def_max* **then**
      if *char_list_exists*(*qo*(c)) **then**
        begin *base_c* ← *char_list_char*(*qo*(c)); *result* ← *qi*(*base_c*); { return *base_c* }
        if ¬*err_p* **then goto** *found*;
        if *font_ec*[f] ≥ *base_c* **then**
          if *font_bc*[f] ≤ *base_c* **then**
            if *char_exists*(*orig_char_info*(f)(*qi*(*base_c*))) **then goto** *found*;
        end;
  if *err_p* **then** { print error and return existing character? }
    begin *begin_diagnostic*; *print_nl*("Missing␣character:␣There␣is␣no␣");
    print("substitution␣for␣"); *print_ASCII*(*qo*(c)); *print*("␣in␣font␣"); *slow_print*(*font_name*[f]);
    print_char("!"); *end_diagnostic*(*false*); *result* ← *qi*(*font_bc*[f]);
      { N.B.: not non-existing character c! }
    end;
found: *effective_char* ← *result*;
  end;

See also section 1397.

This code is used in section 563.

1397. The function *effective_char_info* is equivalent to *char_info*, except it will return *null_character* if neither the character *c* exists in font *f* nor is there a substitution definition for *c*. (For these cases *char_info* using *effective_char* will access an undefined or invalid *font_info* entry. See the documentation of *effective_char* for more information.)

```
⟨Declare additional functions for MLTeX 1396⟩ +≡
function effective_char_info(f : internal_font_number; c : quarterword): four_quarters;
  label exit;
  var ci: four_quarters; {character information bytes for c}
    base_c: integer; {or eightbits: replacement base character}
  begin if ¬mltex_enabled_p then
    begin effective_char_info ← orig_char_info(f)(c); return;
    end;
  if font_ec[f] ≥ qo(c) then
    if font_bc[f] ≤ qo(c) then
      begin ci ← orig_char_info(f)(c); {N.B.: not char_info(f)(c)}
      if char_exists(ci) then
        begin effective_char_info ← ci; return;
        end;
      end;
  if qo(c) ≥ char_sub_def_min then
    if qo(c) ≤ char_sub_def_max then
      if char_list_exists(qo(c)) then
        begin {effective_char_info ← char_info(f)(qi(char_list_char(qo(c))));}
        base_c ← char_list_char(qo(c));
        if font_ec[f] ≥ base_c then
          if font_bc[f] ≤ base_c then
            begin ci ← orig_char_info(f)(qi(base_c)); {N.B.: not char_info(f)(c)}
            if char_exists(ci) then
              begin effective_char_info ← ci; return;
              end;
            end;
        end;
  effective_char_info ← null_character;
exit: end;
```

1398. This code is called for a virtual character *c* in *hlist_out* during *ship_out*. It tries to built a character substitution construct for *c* generating appropriate DVI code using the character substitution definition for this character. If a valid character substitution exists DVI code is created as if *make_accent* was used. In all other cases the status of the substituion for this character has been changed between the creation of the character node in the hlist and the output of the page—the created DVI code will be correct but the visual result will be undefined.

Former MLTEX versions have replaced the character node by a sequence of character, box, and accent kern nodes splicing them into the original horizontal list. This version does not do this to avoid a) a memory overflow at this processing stage, b) additional code to add a pointer to the previous node needed for the replacement, and c) to avoid wrong code resulting in anomalies because of the use within a \leaders box.

```
⟨Output a substitution, goto continue if not possible 1398⟩ ≡
  begin ⟨Get substitution information, check it, goto found if all is ok, otherwise goto continue 1400⟩;
found: ⟨Print character substition tracing log 1401⟩;
  ⟨Rebuild character using substitution information 1402⟩;
  end
```

This code is used in section 623.

1399. The global variables for the code to substitute a virtual character can be declared as local. Nonetheless we declare them as global to avoid stack overflows because *hlist_out* can be called recursivly.

⟨Global variables 13⟩ +≡
accent_c, *base_c*, *replace_c*: *integer*;
ia_c, *ib_c*: *four_quarters*; { accent and base character information }
base_slant, *accent_slant*: *real*; { amount of slant }
base_x_height: *scaled*; { accent is designed for characters of this height }
base_width, *base_height*: *scaled*; { height and width for base character }
accent_width, *accent_height*: *scaled*; { height and width for accent }
delta: *scaled*; { amount of right shift }

1400. Get the character substitution information in *char_sub_code* for the character *c*. The current code checks that the substition exists and is valid and all substitution characters exist in the font, so we can *not* substitute a character used in a substitution. This simplifies the code because we have not to check for cycles in all character substitution definitions.

⟨Get substitution information, check it, goto *found* if all is ok, otherwise goto *continue* 1400⟩ ≡

```
if qo(c) ≥ char_sub_def_min then
  if qo(c) ≤ char_sub_def_max then
    if char_list_exists(qo(c)) then
      begin base_c ← char_list_char(qo(c)); accent_c ← char_list_accent(qo(c));
      if (font_ec[f] ≥ base_c) then
        if (font_bc[f] ≤ base_c) then
          if (font_ec[f] ≥ accent_c) then
            if (font_bc[f] ≤ accent_c) then
              begin ia_c ← char_info(f)(qi(accent_c)); ib_c ← char_info(f)(qi(base_c));
              if char_exists(ib_c) then
                if char_exists(ia_c) then goto found;
              end;
      begin_diagnostic; print_nl("Missing␣character:␣Incomplete␣substitution␣");
      print_ASCII(qo(c)); print("␣=␣"); print_ASCII(accent_c); print("␣"); print_ASCII(base_c);
      print("␣in␣font␣"); slow_print(font_name[f]); print_char("!"); end_diagnostic(false);
      goto continue;
      end;
  begin_diagnostic; print_nl("Missing␣character:␣There␣is␣no␣"); print("substitution␣for␣");
  print_ASCII(qo(c)); print("␣in␣font␣"); slow_print(font_name[f]); print_char("!");
  end_diagnostic(false); goto continue
```

This code is used in section 1398.

1401. For *tracinglostchars* > 99 the substitution is shown in the log file.

⟨Print character substition tracing log 1401⟩ ≡

```
if tracing_lost_chars > 99 then
  begin begin_diagnostic; print_nl("Using␣character␣substitution:␣"); print_ASCII(qo(c));
  print("␣=␣"); print_ASCII(accent_c); print("␣"); print_ASCII(base_c); print("␣in␣font␣");
  slow_print(font_name[f]); print_char("."); end_diagnostic(false);
  end
```

This code is used in section 1398.

1402. This outputs the accent and the base character given in the substitution. It uses code virtually identical to the *make_accent* procedure, but without the node creation steps.

Additionally if the accent character has to be shifted vertically it does *not* create the same code. The original routine in *make_accent* and former versions of MLTEX creates a box node resulting in *push* and *pop* operations, whereas this code simply produces vertical positioning operations. This can influence the pixel rounding algorithm in some DVI drivers—and therefore will probably be changed in one of the next MLTEX versions.

⟨Rebuild character using substitution information 1402⟩ ≡

base_x_height ← *x_height*(*f*); *base_slant* ← *slant*(*f*)/*float_constant*(65536); *accent_slant* ← *base_slant*;
{ slant of accent character font }
base_width ← *char_width*(*f*)(*ib_c*); *base_height* ← *char_height*(*f*)(*height_depth*(*ib_c*));
accent_width ← *char_width*(*f*)(*ia_c*); *accent_height* ← *char_height*(*f*)(*height_depth*(*ia_c*));
{ compute necessary horizontal shift (don't forget slant) }
delta ← *round*((*base_width* − *accent_width*)/*float_constant*(2) + *base_height* ∗ *base_slant* − *base_x_height* ∗ *accent_slant*); *dvi_h* ← *cur_h*; { update *dvi_h*, similar to the last statement in module 620 }
{ 1. For centering/horizontal shifting insert a kern node. }
cur_h ← *cur_h* + *delta*; *synch_h*;
{ 2. Then insert the accent character possibly shifted up or down. }
if ((*base_height* ≠ *base_x_height*) ∧ (*accent_height* > 0)) **then**
begin { the accent must be shifted up or down }
cur_v ← *base_line* + (*base_x_height* − *base_height*); *synch_v*;
if *accent_c* ≥ 128 **then** *dvi_out*(*set1*);
dvi_out(*accent_c*);
cur_v ← *base_line*;
end
else begin *synch_v*;
if *accent_c* ≥ 128 **then** *dvi_out*(*set1*);
dvi_out(*accent_c*);
end;
cur_h ← *cur_h* + *accent_width*; *dvi_h* ← *cur_h*;
{ 3. For centering/horizontal shifting insert another kern node. }
cur_h ← *cur_h* + (−*accent_width* − *delta*);
{ 4. Output the base character. }
synch_h; *synch_v*;
if *base_c* ≥ 128 **then** *dvi_out*(*set1*);
dvi_out(*base_c*);
cur_h ← *cur_h* + *base_width*; *dvi_h* ← *cur_h* { update of *dvi_h* is unnecessary, will be set in module 620 }

This code is used in section 1398.

1403. Dumping MLTEX-related material. This is just the flag in the format that tells us whether MLTEX is enabled.

⟨Dump MLTEX-specific data 1403⟩ ≡

dump_int("4D4C5458); { MLTEX's magic constant: "MLTX" }
if *mltex_p* **then** *dump_int*(1)
else *dump_int*(0);

This code is used in section 1305.

1404. Undump MLTEX-related material, which is just a flag in the format that tells us whether MLTEX is enabled.

⟨ Undump MLTEX-specific data 1404 ⟩ ≡
  undump_int(x); { check magic constant of MLTEX }
  if $x \neq$ ″4D4C5458 **then goto** *bad_fmt*;
  undump_int(x); { undump *mltex_p* flag into *mltex_enabled_p* }
  if $x = 1$ **then** *mltex_enabled_p* ← *true*
  else if $x \neq 0$ **then goto** *bad_fmt*;

This code is used in section 1306.

1405. System-dependent changes. This section should be replaced, if necessary, by any special modifications of the program that are necessary to make TEX work at a particular installation. It is usually best to design your change file so that all changes to previous sections preserve the section numbering; then everybody's version will be consistent with the published program. More extensive changes, which introduce new sections, can be inserted here; then only the index itself will get a new section number.

1406. ⟨Declare action procedures for use by *main_control* 1046⟩ +≡
procedure *insert_src_special*;
  var *toklist*, *p*, *q*: *pointer*;
  begin if (*source_filename_stack*[*in_open*] > 0 ∧ *is_new_source*(*source_filename_stack*[*in_open*], *line*)) **then**
    begin *toklist* ← *get_avail*; *p* ← *toklist*; *info*(*p*) ← *cs_token_flag* + *frozen_special*; *link*(*p*) ← *get_avail*;
    p ← *link*(*p*); *info*(*p*) ← *left_brace_token* + "{";
    q ← *str_toks*(*make_src_special*(*source_filename_stack*[*in_open*], *line*)); *link*(*p*) ← *link*(*temp_head*);
    p ← *q*; *link*(*p*) ← *get_avail*; *p* ← *link*(*p*); *info*(*p*) ← *right_brace_token* + "}"; *ins_list*(*toklist*);
    remember_source_info(*source_filename_stack*[*in_open*], *line*);
    end;
  end;
procedure *append_src_special*;
  var *q*: *pointer*;
  begin if (*source_filename_stack*[*in_open*] > 0 ∧ *is_new_source*(*source_filename_stack*[*in_open*], *line*)) **then**
    begin *new_whatsit*(*special_node*, *write_node_size*); *write_stream*(*tail*) ← 0; *def_ref* ← *get_avail*;
    token_ref_count(*def_ref*) ← *null*; *q* ← *str_toks*(*make_src_special*(*source_filename_stack*[*in_open*], *line*));
    link(*def_ref*) ← *link*(*temp_head*); *write_tokens*(*tail*) ← *def_ref*;
    remember_source_info(*source_filename_stack*[*in_open*], *line*);
    end;
  end;

1407. This function used to be in pdftex, but is useful in tex too.
function *get_nullstr*: *str_number*;
  begin *get_nullstr* ← "";
  end;

1408. Introduction to PUTEX. PUTEXis an extension of TEXto handle CJK character sets.

1409. ⟨ Global variables 13 ⟩ +≡
hi_byte, *lo_byte*: *ASCII_code*; { temp var for storing high byte and low byte of a double-byte character }
db_char: *quarterword*; { temp var for storing a double-byte character }
expand_char: *boolean*;
doc_charset: *eight_bits*;
char_val_flag: *boolean*;

1410. ⟨ Set initial values of key variables 21 ⟩ +≡
expand_char ← *false*;

1411. The default catcode for CJK characters is 'letter'.

⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
for k ← 256 **to** 65535 **do**
begin *cat_code*(k) ← *letter*;
end;

1412. Initially, PUTEX just set type codes for OT1 encoding.

define *set_tail_forbidden*(#) ≡ *set_type_code*(#)(*tail_forbidden*)
define *set_head_forbidden*(#) ≡ *set_type_code*(#)(*head_forbidden*)

⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
set_tail_forbidden("("); *set_tail_forbidden*("["); *set_tail_forbidden*("{");
set_head_forbidden("!"); *set_head_forbidden*(")"); *set_head_forbidden*(",");
set_head_forbidden("."); *set_head_forbidden*(":"); *set_head_forbidden*(";");
set_head_forbidden("?"); *set_head_forbidden*("]"); *set_head_forbidden*("}");

1413. ⟨ PUTeX routines that will be used by TeX routines 1413 ⟩ ≡
function *get_cat_code*(*ch* : *halfword*): *halfword*;
var *cat*: *halfword*; { catcode }
begin if *pux_wcharother* ≠ 0 **then**
if *ch* > 255 **then** *cat* ← *other_char*
else *cat* ← *cat_code*(*ch*)
else *cat* ← *cat_code*(*ch*);
get_cat_code ← *cat*;
end;

See also sections 1431, 1478, and 1479.

This code is used in section 4.

1414. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("`PUXrangecatcode`", *pux_range_catcode*, 0);
primitive("`PUXrangetypecode`", *pux_range_type_code*, 0);

1415. ⟨ Other variables used by the procedure *prefixed_command* 1415 ⟩ ≡
bc, *ec*: *halfword*; { the begin char and end char of code range }

See also sections 1536, 1540, 1551, and 1564.

This code is used in section 1214.

1416. ⟨ Assignments 1220 ⟩ +≡

```
pux_range_catcode, pux_range_type_code: begin p ← cur_chr;
  if cur_cmd = pux_range_catcode then
    begin n ← max_char_code; p ← cat_code_base;
    end
  else begin n ← max_type_code; p ← pux_type_code_base;
    end;
  scan_wchar_num; bc ← cur_val;
  scan_keyword("to");
  scan_wchar_num; ec ← cur_val;
  scan_optional_equals;
  scan_int;
  if (bc = 0) ∨ (ec = 0) ∨ (ec < bc) then
    begin if ec < bc then
      begin print_err("Invalid␣range␣setting,␣ec␣<␣bc");
      end;
    help1("I´m␣going␣to␣ignore␣this␣command.");
    error; goto exit;
    end;
  if (cur_val < 0) ∨ (cur_val > n) then
    begin print_err("Invalid␣catcode␣("); print_int(cur_val);
    print("),␣should␣be␣in␣the␣range␣0..15");
    help1("I´m␣going␣to␣ignore␣this␣command.");
    error; goto exit;
    end;
  for k ← bc to ec do define(p + k, data, cur_val);
  end;
```

1417. ⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡

```
  for k ← 0 to 255 do local_names(k) ← "?";
```

1418. ⟨ PUTeX basic scanning routines 1418 ⟩ ≡

```
function scan_name: str_number;
  begin ⟨ Get the next non-blank non-call token 409 ⟩;
  while cur_cmd = letter do
    begin if (is_wchar(cur_chr)) then append_wchar(cur_chr)
    else append_char(cur_chr);
    get_x_token;
    end;
  if pool_ptr ≠ str_start[str_ptr] then scan_name ← make_string
  else scan_name ← 0;
  end;
```

This code is used in section 466.

1419. ⟨ Declare procedures that scan restricted classes of integers 436 ⟩ +≡

```
procedure scan_wchar_num;
  begin scan_int;
  if (cur_val < 257) ∨ (cur_val > 65535) then
    begin print_err("Bad␣wide␣character␣code");
    help2("A␣wide␣character␣number␣must␣be␣between␣256␣and␣65536.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

1420. CJK Numbers.

⟨ Global variables 13 ⟩ +≡
cnum_one_flag: *boolean*;

1421.

define *ten_wchar_offset* = 10
define *hundred_wchar_offset* = 11
define *thousand_wchar_offset* = 12
define *ten_thousand_wchar_offset* = 13
define *hundred_million_wchar_offset* = 14
define *arabic_wchar_offset* = 40
define *negative_wchar_offset* = 50
define *negative_wsym_offset* = 51
define *twenty_wchar_offset* = 52
define *thirty_wchar_offset* = 53
define *CJK_digit_offset* = 0
define *C_simple_digit_offset* = 10
define *C_formal_digit_offset* = 25
define *C_arabic_digit_offset* = 40

1422.

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_chinese_int*(n, *digit_base* : *integer*; *simple*, *formal* : *boolean*);
  var m: *integer*;
  begin *cnum_one_flag* ← *false*;
  if $n < 0$ **then**
    begin { *print_dbchar* is replaced by the following 2 *print_char* calls. }
    print_wchar(*local_names*(*negative_wchar_offset*)); *negate*(n);
    end;
  if $n < 100$ **then** *print_small_chinese_int*(n, *digit_base*, *simple*, *formal*)
  else begin if $n > 99999999$ **then**
      begin *print_small_chinese_int*(n **div** 100000000, *digit_base*, *simple*, *formal*);
      print_wchar(*local_names*(*digit_base* + *hundred_million_wchar_offset*)); *cnum_one_flag* ← *true*;
      $n \leftarrow n$ **mod** 100000000;
      if $n > 0 \land n < 10000000$ **then** *print_wchar*(*local_names*(*digit_base*)); { zero character in Chinese }
      end;
    if $n > 9999$ **then**
      begin *print_medium_chinese_int*(n **div** 10000, *digit_base*, *simple*, *formal*);
      print_wchar(*local_names*(*digit_base* + *ten_thousand_wchar_offset*)); *cnum_one_flag* ← *true*;
      $n \leftarrow n$ **mod** 10000;
      if $n > 0 \land n < 1000$ **then** *print_wchar*(*local_names*(*digit_base*)); { zero character in Chinese }
      end;
    print_medium_chinese_int(n, *digit_base*, *simple*, *formal*);
    end;
  end;

1423. The following procedure prints a number n, $0 \le n \le 99$.

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_small_chinese_int*(*n*, *digit_base* : *integer*; *simple*, *formal* : *boolean*);
  label *done1*;
  begin if $n < 10$ **then** *print_wchar*(*local_names*(*n* + *digit_base*))
  else begin if $n < 20$ **then**
      begin if *formal* ∨ *cnum_one_flag* **then** *print_wchar*(*local_names*(*digit_base* + 1));
      print_wchar(*local_names*(*digit_base* + 10));
      goto *done1*;
      end;
    if $n < 30$ ∧ *simple* **then**
      begin *print_wchar*(*local_names*(*twenty_wchar_offset*));
      goto *done1*;
      end;
    if $n < 40$ ∧ *simple* **then**
      begin *print_wchar*(*local_names*(*thirty_wchar_offset*));
      goto *done1*;
      end;
    print_wchar(*local_names*(*digit_base* + *n* **div** 10)); *print_wchar*(*local_names*(*digit_base* + 10));
  done1: *n* ← *n* **mod** 10;
    if $n > 0$ **then** *print_wchar*(*local_names*(*n* + *digit_base*));
    end
  end;

1424. Print a chinese number of medium size.

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_medium_chinese_int*(*n*, *digit_base* : *integer*; *simple*, *formal* : *boolean*);
  begin if $n > 999$ **then**
    begin *print_wchar*(*local_names*(*digit_base* + *n* **div** 1000));
    print_wchar(*local_names*(*digit_base* + *thousand_wchar_offset*)); *n* ← *n* **mod** 1000;
    if $n > 0 \land n < 99$ **then** *print_wchar*(*local_names*(*digit_base*)); { zero character in Chinese }
    end;
  if $n > 99$ **then**
    begin *print_wchar*(*local_names*(*digit_base* + *n* **div** 100));
    print_wchar(*local_names*(*digit_base* + *hundred_wchar_offset*)); *n* ← *n* **mod** 100;
    if $n > 0 \land n < 9$ **then** *print_wchar*(*local_names*(*digit_base*)); { zero character in Chinese }
    end;
  cnum_one_flag ← *true*;
  if $n > 0$ **then** *print_small_chinese_int*(*n*, *digit_base*, *simple*, *formal*);
  end;

1425. ⟨ Put each of TeX's primitives into the hash table 226 ⟩ +≡
  primitive("puxnumdigits", *pux_get_int*, *int_base* + *pux_digit_num_code*);
  primitive("puxsign", *pux_get_int*, *int_base* + *pux_sign_code*);
  primitive("puxdigit", *pux_get_int*, *int_base* + *pux_digit_base*);

1426. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
pux_get_int: **if** *chr_code* = *pux_digit_num_code* + *int_base* **then** *print_esc*("puxnumdigits")
  else if *chr_code* = *pux_sign_code* + *int_base* **then** *print_esc*("puxsign")
    else if *chr_code* = *pux_digit_base* + *int_base* **then** *print_esc*("puxdigit");

1427. ⟨ Assignments 1220 ⟩ +≡
pux_get_int: **begin** *print_err*("You␣can´t␣assign␣values␣to␣internal␣read-only␣parameters.");
  error;
  end;

1428. ⟨ scan PUTEX internal values 1428 ⟩ ≡
  begin if $m = pux\_digit\_base + int\_base$ **then**
    begin *scan_int*;
    if $cur\_val < 0 \lor cur\_val > 9$ **then**
      begin *print_err*("Improper␣digit␣place␣specified␣("); *print_int*(*cur_val*);
      print("),␣replaced␣by␣0"); $cur\_val \leftarrow 0$;
      end;
    $m \leftarrow m + cur\_val$;
    end;
  scanned_result(*eqtb*[*m*].*int*)(*int_val*);
  end

This code is used in section 416.

1429. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive("PUXsplitnumber", *pux_split_number*, 0);

1430. ⟨ Assignments 1220 ⟩ +≡
pux_split_number: **begin** *scan_int*; *split_number*(*cur_val*);
  end;

1431. The following procedure splits the integer parameter n to digit list and stores the number of digits into *pux_digit_num*, the sign (1: positive or -1: negative) into *pux_num_sign*, and the digits into the array *pux_nth_digit*. Since the largest n is 2^{31}, n contains at most 10 digits.

⟨ PUTeX routines that will be used by TeX routines 1413 ⟩ +≡
procedure *split_number*(*n* : *integer*);
  var *k*: 0 . . 10;
  begin if $n < 0$ **then**
    begin $pux\_num\_sign \leftarrow -1$; *negate*(*n*)
    end
  else $pux\_num\_sign \leftarrow 1$;
  $k \leftarrow 0$;
  repeat $pux\_nth\_digit(k) \leftarrow n$ **mod** 10; $n \leftarrow n$ **div** 10; *incr*(*k*);
  until $n = 0$;
  $pux\_digit\_num \leftarrow k$;
  while $k < 10$ **do**
    begin $pux\_nth\_digit(k) \leftarrow 0$; *incr*(*k*);
    end;
  end;

1432.

⟨ scan and split the number 1432 ⟩ ≡
  begin *scan_int*; *split_number*(*cur_val*);
  end

This code is used in section 474.

1433. ⟨ scan a CJK number with a possible selector and then split it 1433 ⟩ ≡
begin *scan_int*; *saved_val* ← *cur_val*; *split_number*(*cur_val*);
if *scan_keyword*("offset") **then**
begin *scan_eight_bit_int*; *digit_base* ← *cur_val*;
if *scan_keyword*("sign") **then**
begin *scan_eight_bit_int*; *sign* ← *cur_val*;
end
else *sign* ← *negative_wchar_offset*;
end
else *digit_base* ← 0;
end

This code is used in section 474.

1434. Using full-width arabic characters to show chinese numbers.

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_cjk_int*(*n* : *integer*; *digit_base*, *sign* : *integer*);
var *k*: 0 . . 9; { index to current digit }
begin if *pux_num_sign* $= -1$ **then** *print_wchar*(*local_names*(*sign*));
for *k* ← *pux_digit_num* $- 1$ **downto** 0 **do** *print_wchar*(*local_names*(*digit_base* + *pux_nth_digit*(*k*)));
end;

1435. ⟨ using full-width arabic characters to print a CJK number 1435 ⟩ ≡
print_cjk_int(*cur_val*, *C_arabic_digit_offset*, *negative_wsym_offset*)

This code is used in section 475.

1436. ⟨ print a CJK number with specified format 1436 ⟩ ≡
print_cjk_int(*saved_val*, *digit_base*, *sign*)

This code is used in section 475.

1437. ⟨ scan a CJK name sequence number 1437 ⟩ ≡
begin *scan_eight_bit_int*; *saved_val* ← *cur_val*;
if *scan_keyword*("min") **then**
begin *scan_optional_equals*; *scan_eight_bit_int*; *min_val* ← *cur_val*;
end
else begin *print_err*("Missing␣`min'␣part␣("); *print*("min␣0␣inserted)"); *error*;
end;
if *scan_keyword*("max") **then**
begin *scan_optional_equals*; *scan_eight_bit_int*; *max_val* ← *cur_val*;
end
else begin *print_err*("Missing␣`max'␣part␣("); *print*("max␣255␣inserted)"); *error*;
end;
if *scan_keyword*("offset") **then**
begin *scan_optional_equals*; *scan_eight_bit_int*; *offset* ← *cur_val*;
end
else begin *print_err*("Missing␣`offset'␣part␣("); *print*("offset␣0␣inserted)"); *error*;
end;
if *min_val* ≤ *saved_val* ∧ *saved_val* ≤ *max_val* **then** *cur_val* ← *offset* + *saved_val* − *min_val*
else begin *print_err*("Number␣is␣out␣of␣the␣range␣("); *print*("replaced␣with␣the␣min␣value)");
cur_val ← *offset*; *error*;
end;
end

This code is used in section 474.

1438. ⟨ print a CJK name sequence member 1438 ⟩ ≡
print_wchar(*local_names*(*cur_val*))

This code is used in section 475.

1439. A fix_word is a *scaled integers* that are multiples of 2^{-20}. In other words, a binary point is assumed to be twenty bit positions from the right end of a binary computer word.

define *fw_unity* ≡ "100000 { 2^{20}, represents 1.00000 }
define *fw_two* ≡ "200000 { 2^{21}, represents 2.00000 }
define *fw_one_fifth* ≡ "33333 { 0.2 }
define *convfix*(#) ≡ (#) ∗ *fw_unity* **div** 1000

⟨ Types in the outer block 18 ⟩ +≡
fixword = *integer*; { this type is used for fixword (12.20) integers }

1440. ⟨ Declare the function called *print_fixword* 1440 ⟩ ≡

```
procedure print_fixword(s : fixword); { prints fixword real, rounded to five digits }
  var delta: fixword; { amount of allowable inaccuracy }
  begin if s < 0 then
    begin print_char("-"); negate(s); { print the sign, if negative }
    end;
  print_int(s div fw_unity); { print the integer part }
  print_char("."); s ← 10 ∗ (s mod fw_unity) + 5; delta ← 10;
  repeat if delta > fw_unity then s ← s + '200000000 − 50000; { round the last digit }
    print_char("0" + (s div fw_unity)); s ← 10 ∗ (s mod fw_unity); delta ← delta ∗ 10;
  until s ≤ delta;
  end;
```

This code is used in section 1482.

1441. The function *fw_times_sd* do the multiplication of a fixword and a scaled number. The value of fixword is assumed between 16 and −16. The function returns the result as a scaled number. (See also Sec. 571, 572 and 600.)

⟨ Declare the function called *fw_times_sd* 1441 ⟩ ≡

```
function fw_times_sd(x : fixword; z : scaled): scaled; { compute f times s }
  var sw: scaled; a, b, c, d: eight_bits; { byte variables }
    alpha: integer; beta: 1 . . 16;
  begin ⟨ Replace z by z′ and compute α, β 575 ⟩;
  if x ≥ 0 then a ← x div '100000000
  else begin x ← x + '10000000000; x ← x + '10000000000; a ← (x div '100000000) + 128;
    end;
  x ← x mod '100000000; b ← x div '200000; x ← x mod '200000; c ← x div '400; d ← x mod '400;
  sw ← (((((d ∗ z) div '400) + (c ∗ z)) div '400) + (b ∗ z)) div beta;
  if a = 0 then fw_times_sd ← sw
  else if a = 255 then fw_times_sd ← sw − alpha
    else fw_times_sd ← unity;
  end;
```

This code is used in section 1260.

1442. ⟨ Put each of $\TeX$'s primitives into the hash table 226 ⟩ +≡
primitive("PUXchar", *pux_char_num*, 0);

1443. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
pux_char_num: *print_esc*("PUXchar");

1444. ⟨ Give improper hyphenation error for Chinese characters inside 1444 ⟩ ≡
  begin *print_err*("Improper␣"); *print_esc*("hyphenation"); *print*("␣will␣be␣flushed");
  help2("Hyphenation␣exceptions␣can´t␣contain␣Chinese␣characters")
  ("But␣continue;␣I´ll␣forgive␣and␣forget."); *error*;
  end

This code is used in section 938.

1445. ⟨ Cases of *main_control* that build boxes and lists 1059 ⟩ +≡
mmode + *pux_char_num*: **begin** *scan_wchar_num*; *cur_chr* ← *cur_val*;
  print_err("Chinese␣character␣is␣ignored␣in␣math␣mode");
  help1("Did␣you␣forget␣putting␣it␣into␣an␣\hbox?"); *error*;
  end;
mmode + *pux_char_given*: **begin** *print_err*("Chinese␣character␣is␣ignored␣in␣math␣mode");
  help1("Did␣you␣forget␣putting␣it␣into␣an␣\hbox?"); *error*;
  end;

1446. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
pux_char_given: **begin** *print_esc*("PUXchar"); *print_hex*(*chr_code*);
  end;

1447. All about spaces.

define $\mathit{is\_tail\_forbidden}(\#) \equiv \mathit{type\_code}(\#) = \mathit{tail\_forbidden}$
define $\mathit{is\_head\_forbidden}(\#) \equiv \mathit{type\_code}(\#) = \mathit{head\_forbidden}$
define $\mathit{is\_head\_forbidden\_wchar}(\#) \equiv ((\# > 255) \land (\mathit{type\_code}(\#) = \mathit{head\_forbidden}))$
define $\mathit{is\_punc\_wchar}(\#) \equiv ((\# > 255) \land (\mathit{type\_code}(\#) \neq 0))$

1448.

⟨Global variables 13⟩ +≡
main_cf: *internal_cfont_number*; { the current chinese font }
math_mode_save: $-\mathit{mmode} \,..\, \mathit{mmode}$;
prev_main_cf: *internal_cfont_number*; { the current chinese font }
pre_undet_glue_ptr: *pointer*; { point to the node just before a undetermined glue }
undet_glue_ptr: *pointer*; { point to the undetermined glue }
cglue_ptr: *pointer*;
cglue_spec: *pointer*;
pre_glue_char_ptr: *pointer*;
outer_tail: *pointer*;
hbox_tail: *pointer*;
in_set_box: *boolean*;

1449. ⟨Initialization of global variables done in the *main_control* procedure 1449⟩ ≡
pre_undet_glue_ptr ← *null*; *pre_glue_char_ptr* ← *null*;

See also section 1507.

This code is used in section 1033.

1450.

define *tail_append_glue*(#) ≡
begin *cglue_ptr* ← *get_node*(*small_node_size*); *cglue_spec* ← #; *type*(*cglue_ptr*) ← *glue_node*;
subtype(*cglue_ptr*) ← *normal*; *leader_ptr*(*cglue_ptr*) ← *null*; *glue_ptr*(*cglue_ptr*) ← *cglue_spec*;
incr(*glue_ref_count*(*cglue_spec*)); *tail_append*(*cglue_ptr*);
end

1451. Here is the check done before switching to regular character string.

⟨ If the preceding node is wchar node, then append a cespace 1451 ⟩ ≡

```
if tail = head then
  begin if mode = −hmode then
    begin { beginning of a restricted hlist }
    outer_tail ← nest[nest_ptr − 1].tail_field;
    if pre_undet_glue_ptr ≠ null then
      begin if outer_tail = link(pre_undet_glue_ptr) ∧ pre_glue_char_ptr ≠
            null ∧ is_wchar_node(pre_glue_char_ptr) then
        begin decr(glue_ref_count(glue_ptr(outer_tail)));
        glue_ptr(outer_tail) ← cfont_ceglue_spec[prev_main_cf];
        incr(glue_ref_count(cfont_ceglue_spec[prev_main_cf]));
        end;
      pre_undet_glue_ptr ← null;
      end;
    end;
  end
else if is_char_node(tail) ∧ is_wchar_node(tail) then
    begin if is_head_forbidden(cur_chr) then tail_append(new_penalty(inf_penalty));
    tail_append_glue(cfont_ceglue_spec[main_cf]);
    end
  else if pre_undet_glue_ptr ≠ null ∧ link(pre_undet_glue_ptr) = tail ∧ pre_glue_char_ptr ≠
          null ∧ is_wchar_node(pre_glue_char_ptr) then
      begin decr(glue_ref_count(glue_ptr(tail))); glue_ptr(tail) ← cfont_ceglue_spec[prev_main_cf];
      incr(glue_ref_count(cfont_ceglue_spec[prev_main_cf]));
      end;
pre_undet_glue_ptr ← null; pre_glue_char_ptr ← null;
```

This code is used in section 1037.

1452. If the next token come after the math shift $ is a wide character, then a cespace is appended first.

⟨ If the token is a wide character, then append a cspace 1452 ⟩ ≡

```
if cur_cmd = pux_char_num then
  begin scan_wchar_num; cur_chr ← cur_val; cur_cmd ← pux_char_given;
  end;
if cur_cmd = letter ∨ cur_cmd = other_char ∨ cur_cmd = pux_char_given then
  if is_wchar(cur_chr) then
    if is_punc_wchar(cur_chr) then
      begin if is_head_forbidden(cur_chr) then tail_append(new_penalty(inf_penalty));
      tail_append_glue(zero_glue);
      end
    else tail_append_glue(cfont_ceglue_spec[main_cf])
```

This code is used in section 1196.

1453.

⟨ Append double-byte character *cur_chr* and the following double-byte characters (if any) to the current hlist in the current font; **goto** *main_loop* when a single-byte character has been fetched; **goto** *reswitch* when a non-character has been fetched 1453 ⟩ ≡

main_cf ← *cur_cfont*;
⟨ If the current wchar is at the beginning of a restricted hlist that is after a undetermined spacer, then we have to determine that space. When it is done **goto** *save_cur_wchar* 1454 ⟩;
⟨ If the previous node is an undetermined glue, then make it certain and **goto** *save_cur_wchar* 1456 ⟩;
if ¬*is_char_node*(*tail*) **then goto** *save_cur_wchar*;
main_loop_wchar + 1: ⟨ the previous node is a character node, so we have to append a glue first 1457 ⟩;
save_cur_wchar: *fast_get_avail*(*lig_stack*); *font*(*lig_stack*) ← *main_cf*; *character*(*lig_stack*) ← *cur_chr*;
tail_append(*lig_stack*);
⟨ Prepare a nonbreak space if the current wide character is not allowed to appear at the end of line 1458 ⟩;
fetch_next_tok: *get_next*; { set only *cur_cmd* and *cur_chr*, for speed }
⟨ Check the lookahead character 1459 ⟩;
x_token; { now expand and set *cur_cmd*, *cur_chr*, *cur_tok* }
⟨ Check the lookahead character 1459 ⟩;
if *cur_cmd* = *char_num* **then**
begin *scan_char_num*; *cur_chr* ← *cur_val*; *cur_cmd* ← *char_given*; **goto** *next_is_a_char*;
end;
if *cur_cmd* = *pux_char_num* **then**
begin *scan_wchar_num*; *cur_chr* ← *cur_val*;
if *is_punc_wchar*(*cur_chr*) **then**
if *is_head_forbidden*(*cur_chr*) **then** *tail_append*(*new_penalty*(*inf_penalty*));
tail_append_glue(*cfont_glue_spec*[*main_cf*]); **goto** *save_cur_wchar*;
end; { next token is not a character token }
if *cur_cmd* = *math_shift* **then**
if *is_punc_wchar*(*character*(*lig_stack*)) **then** *tail_append_glue*(*zero_glue*)
else *tail_append_glue*(*cfont_ceglue_spec*[*main_cf*]);
goto *reswitch*;
next_is_a_char: **begin if** *cur_chr* < 256 **then**
if *is_head_forbidden*(*cur_chr*) **then** *tail_append*(*new_penalty*(*inf_penalty*));
if *is_punc_wchar*(*character*(*lig_stack*)) **then** *tail_append_glue*(*zero_glue*)
else *tail_append_glue*(*cfont_ceglue_spec*[*main_cf*]);
goto *main_loop* + 1;
end

This code is used in section 1033.

1454. ⟨ If the current wchar is at the beginning of a restricted hlist that is after a undetermined spacer, then we have to determine that space. When it is done **goto** *save_cur_wchar* 1454 ⟩ ≡

if *tail* = *head* **then**
  begin { beginning of a restricted hlist }
  if *mode* = −*hmode* **then**
    begin *outer_tail* ← *nest*[*nest_ptr* − 1].*tail_field*;
    if *pre_undet_glue_ptr* ≠ *null* **then**
      begin if *outer_tail* = *link*(*pre_undet_glue_ptr*) **then**
        begin *undet_glue_ptr* ← *outer_tail*;
        ⟨ Modify the undetermined glue according the type of pre-glue character 1455 ⟩;
        end;
      pre_undet_glue_ptr ← *null*;
      end;
    end;
  goto *save_cur_wchar*;
  end

This code is used in section 1453.

1455. ⟨ Modify the undetermined glue according the type of pre-glue character 1455 ⟩ ≡

decr(*glue_ref_count*(*glue_ptr*(*undet_glue_ptr*)));
if *pre_glue_char_ptr* ≠ *null* ∧ *is_wchar_node*(*pre_glue_char_ptr*) **then**
  begin *glue_ptr*(*undet_glue_ptr*) ← *cfont_glue_spec*[*prev_main_cf*];
  incr(*glue_ref_count*(*cfont_glue_spec*[*prev_main_cf*])); *pre_glue_char_ptr* ← *null*;
  end
else begin *glue_ptr*(*undet_glue_ptr*) ← *cfont_ceglue_spec*[*prev_main_cf*];
  incr(*glue_ref_count*(*cfont_ceglue_spec*[*prev_main_cf*]));
  end

This code is used in sections 1454 and 1456.

1456. ⟨ If the previous node is an undetermined glue, then make it certain and **goto** *save_cur_wchar* 1456 ⟩ ≡

if *pre_undet_glue_ptr* ≠ *null* **then**
  begin if *link*(*pre_undet_glue_ptr*) = *tail* **then**
    begin *undet_glue_ptr* ← *tail*;
    ⟨ Modify the undetermined glue according the type of pre-glue character 1455 ⟩;
    pre_undet_glue_ptr ← *null*; **goto** *save_cur_wchar*;
    end;
  pre_undet_glue_ptr ← *null*;
  end

This code is used in section 1453.

1457. ⟨the previous node is a character node, so we have to append a glue first 1457⟩ ≡

if *is_wchar_node*(*tail*) **then**
  begin if *is_head_forbidden_wchar*(*cur_chr*) **then** *tail_append*(*new_penalty*(*inf_penalty*));
  tail_append(*new_glue*(*cfont_glue_spec*[*main_cf*]));
  end
else begin { previous node is a single byte character }
  if *is_punc_wchar*(*cur_chr*) **then**
    begin if *is_head_forbidden*(*cur_chr*) **then** *tail_append*(*new_penalty*(*inf_penalty*));
    tail_append_glue(*zero_glue*);
    end
  else begin if *is_head_forbidden*(*character*(*tail*)) **then** *tail_append*(*new_penalty*(*inf_penalty*));
    tail_append_glue(*cfont_ceglue_spec*[*main_cf*]);
    end;
  end

This code is used in section 1453.

1458. For those Chinese puncuations that shoudn't appear in the line end, we append a penalty node to prevent line boken after it.

⟨Prepare a nonbreak space if the current wide character is not allowed to appear at the end of line 1458⟩ ≡

if *is_punc_wchar*(*cur_chr*) **then**
  if *is_tail_forbidden*(*cur_chr*) **then** *tail_append*(*new_penalty*(*inf_penalty*))

This code is used in section 1453.

1459. ⟨Check the lookahead character 1459⟩ ≡

if $cur\_cmd = letter \lor cur\_cmd = other\_char \lor cur\_cmd = pux\_char\_given \lor cur\_cmd = char\_given$ **then**
  if *is_wchar*(*cur_chr*) **then**
    begin if *is_punc_wchar*(*cur_chr*) **then**
      if *is_head_forbidden*(*cur_chr*) **then** *tail_append*(*new_penalty*(*inf_penalty*));
    tail_append_glue(*cfont_glue_spec*[*main_cf*]); **goto** *save_cur_wchar*;
    end
  else goto *next_is_a_char*

This code is used in sections 1453 and 1453.

1460. ⟨Look ahead for next character. If it is a wide character then append a cespace, or leave *lig_stack* empty if there's no character there 1460⟩ ≡

get_next; {set only *cur_cmd* and *cur_chr*, for speed}
if *cur_cmd* = *letter* ∨ *cur_cmd* = *other_char* **then**
  if *is_wchar*(*cur_chr*) **then goto** *main_loop_lookahead* + 2
  else goto *main_loop_lookahead* + 1;
if *cur_cmd* = *char_given* **then goto** *main_loop_lookahead* + 1;
if *cur_cmd* = *pux_char_given* **then goto** *main_loop_lookahead* + 2;
x_token; {now expand and set *cur_cmd*, *cur_chr*, *cur_tok*}
if *cur_cmd* = *letter* ∨ *cur_cmd* = *other_char* **then**
  if *is_wchar*(*cur_chr*) **then goto** *main_loop_lookahead* + 2
  else 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* = *pux_char_num* **then**
  begin *scan_wchar_num*; *cur_chr* ← *cur_val*; **goto** *main_loop_lookahead* + 2;
  end;
if *cur_cmd* = *no_boundary* **then** *bchar* ← *non_char*;
main_loop_lookahead + 2: *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 1041.

1461. ⟨Cases of *main_control* that handle spacer 1461⟩ ≡

hmode + *spacer*: ⟨Lookahead and determine the type of spacer to append 1463⟩;
hmode + *ex_space*: ⟨Lookahead and determine the type of *ex_spacer* to append 1464⟩;
mmode + *ex_space*: **begin if** *pux_xspace* = 0 **then** *get_x_token*; {lookahead}
  goto *append_normal_space*;
  end;
hmode + *pux_space*: ⟨Handle PUTEXspace command 1467⟩;
mmode + *pux_space*: **begin** *print_err*("This␣space␣command␣is␣ignored␣in␣math␣mode");
  help1("Did␣you␣forget␣putting␣it␣into␣an␣\hbox?"); *error*;
  end;

This code is used in section 1033.

1462. ⟨Setup *hbox_tail* and package 1462⟩ ≡

if *in_set_box* **then** *package*(0)
else begin if *tail* ≠ *head* ∧ *is_char_node*(*tail*) **then** *hbox_tail* ← *tail*
  else *hbox_tail* ← *null*;
  package(0); *get_x_token*;
  if *cur_cmd* ≠ *spacer* **then** *hbox_tail* ← *null*;
  back_input;
  end

This code is used in sections 1088 and 1088.

1463. ⟨Lookahead and determine the type of spacer to append 1463⟩ ≡

begin if $pux\_xspace = 0$ **then**
  begin if $tail \neq head \land is\_char\_node(tail)$ **then** $pre\_glue\_char\_ptr \leftarrow tail$
  else $pre\_glue\_char\_ptr \leftarrow null$;
  get_x_token; { lookahead }
  if $cur\_cmd = char\_num$ **then**
    begin $scan\_char\_num$; $cur\_chr \leftarrow cur\_val$; $cur\_cmd \leftarrow char\_given$;
    end
  else if $cur\_cmd = pux\_char\_num$ **then**
      begin $scan\_wchar\_num$; $cur\_chr \leftarrow cur\_val$; $cur\_cmd \leftarrow pux\_char\_given$;
      end;
  if $cur\_cmd = letter \lor cur\_cmd = other\_char \lor cur\_cmd = char\_given \lor cur\_cmd = pux\_char\_given$ **then**
    if $is\_wchar(cur\_chr)$ **then**
      begin $main\_cf \leftarrow cur\_cfont$;
      if $pre\_glue\_char\_ptr \neq null$ **then goto** $main\_loop\_wchar + 1$;
      if $hbox\_tail \neq null \land is\_wchar\_node(hbox\_tail)$ **then**
        begin $tail\_append\_glue(cfont\_glue\_spec[main\_cf])$; $hbox\_tail \leftarrow null$;
        end
      else begin $tail\_append\_glue(cfont\_ceglue\_spec[main\_cf])$;
        if $is\_punc\_wchar(cur\_chr)$ **then**
          if $is\_head\_forbidden(cur\_chr)$ **then** $tail\_append(new\_penalty(inf\_penalty))$;
        if $hbox\_tail \neq null$ **then** $hbox\_tail \leftarrow null$;
        end;
      goto $save\_cur\_wchar$;
      end
    else if $(pre\_glue\_char\_ptr \neq null \land is\_wchar\_node(tail)) \lor (hbox\_tail \neq null \land is\_wchar\_node(hbox\_tail))$
          then
        begin $tail\_append\_glue(cfont\_ceglue\_spec[cur\_cfont])$; $hbox\_tail \leftarrow null$; **goto** $main\_loop$;
        end;
  $prev\_main\_cf \leftarrow cur\_cfont$; $pre\_undet\_glue\_ptr \leftarrow tail$;
  if $pre\_glue\_char\_ptr \neq null \land is\_wchar\_node(pre\_glue\_char\_ptr)$ **then**
    begin $tail\_append\_glue(cfont\_ceglue\_spec[cur\_cfont])$; **goto** $reswitch$;
    end;
  end;
if $space\_factor = 1000$ **then goto** $append\_normal\_space$
else begin app_space;
  if $pux\_xspace = 0$ **then goto** $reswitch$
  else goto big_switch;
  end;
end

This code is used in section 1461.

1464. ⟨Lookahead and determine the type of *ex_spacer* to append 1464⟩ ≡

```
begin if pux_xspace = 0 then
  begin get_x_token;   { lookahead }
  if cur_cmd = char_num then
    begin scan_char_num; cur_chr ← cur_val; cur_cmd ← char_given;
    end;
  if cur_cmd = pux_char_num then
    begin scan_wchar_num; cur_chr ← cur_val; cur_cmd ← pux_char_given;
    end;
  if cur_cmd = letter ∨ cur_cmd = other_char ∨ cur_cmd = char_given ∨ cur_cmd = pux_char_given then
    if is_wchar(cur_chr) then
      begin main_cf ← cur_cfont;
      if tail ≠ head ∧ is_char_node(tail) then
        if is_wchar_node(tail) then goto append_normal_space
        else goto main_loop_wchar + 1;
      tail_append_glue(cfont_glue_spec[main_cf]); goto save_cur_wchar;
      end
    else if tail ≠ head ∧ is_char_node(tail) then
        if is_wchar_node(tail) then
          begin tail_append_glue(cfont_ceglue_spec[cur_cfont]); goto main_loop;
          end;
  if tail ≠ head ∧ is_char_node(tail) then
    if is_wchar_node(tail) then
      begin tail_append_glue(cfont_glue_spec[cur_cfont]); goto reswitch;
      end;
  prev_main_cf ← cur_cfont; pre_undet_glue_ptr ← tail;
  end;
goto append_normal_space;
end
```

This code is used in section 1461.

1465.

```
define pux_space_code = 0
define pux_exspace_code = 1
define pux_cspace_code = 2
define pux_cespace_code = 3
```

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

```
primitive("PUXspace", pux_space, pux_space_code);
primitive("PUXexspace", pux_space, pux_exspace_code);
primitive("PUXcspace", pux_space, pux_cspace_code);
primitive("PUXcespace", pux_space, pux_cespace_code);
```

1466. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡

```
pux_space: case chr_code of
  pux_space_code: print_esc("PUXspace");
  pux_exspace_code: print_esc("PUXexspace");
  pux_cspace_code: print_esc("PUXcspace");
  othercases print_esc("PUXcespace")
  endcases;
```

1467. ⟨ Handle PUTEXspace command 1467 ⟩ ≡
case *cur_chr* **of**
pux_space_code: **begin** *get_x_token*;
if *space_factor* = 1000 **then goto** *append_normal_space*;
app_space;
if *pux_xspace* = 0 **then goto** *reswitch*
else goto *big_switch*;
end;
pux_exspace_code: **begin** *get_x_token*; **goto** *append_normal_space*;
end;
pux_cspace_code: *tail_append*(*new_glue*(*cfont_glue_spec*[*cur_cfont*]));
othercases *tail_append*(*new_glue*(*cfont_ceglue_spec*[*cur_cfont*]))
endcases

This code is used in section 1461.

1468. CJK font face definition table.

1469. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("PUXcfacedef", *pux_cface_def*, 0);

1470. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
pux_cface_def: *print_esc*("PUXcfacedef"); { TCW }

1471. ⟨ Assignments 1220 ⟩ +≡
pux_cface_def: *new_cface*(*a*);

1472. ⟨ Constants in the outer block 11 ⟩ +≡
cface_base = 0; { CJK font face base }
null_cface = 0; { null CJK font faces }

1473.
⟨ Types in the outer block 18 ⟩ +≡
internal_cface_number = *cface_base* . . *max_cface*;

1474. The CJK font face definition table is implemented by parallel arrays as follows.

define *regular* = 0
define *italic* = "40 { bit 6: italic flag }
define *underline* = "20 { bit 5: underline flag }
define *strikeout* = "10 { bit 4: strikeout flag }
define *inverse* = "08 { bit 3: inverse flag }
define *rotated* = "01 { bit 0: rotation flag }
define *default_cface_weight* ≡ 400
define *default_cface_style* = *regular*
define *default_cface_fw_width* ≡ *fw_unity*
define *default_cface_fw_height* ≡ *fw_unity*
define *cface_id_text*(#) ≡ *text*(*cface_id_base* + #)

⟨ Global variables 13 ⟩ +≡
cface_ptr: *internal_cface_number*; { index of the first unused entry }
cface: **array** [*internal_cface_number*] **of** *str_number*; { CJK font face identifier }
cface_name: **array** [*internal_cface_number*] **of** *str_number*; { CJK font face name }
cface_charset: **array** [*internal_cface_number*] **of** *eight_bits*; { CJK font charset }
cface_weight: **array** [*internal_cface_number*] **of** 1 . . 1000; { CJK font weight }
cface_style: **array** [*internal_cface_number*] **of** *eight_bits*; { CJK font style }
cface_fw_width: **array** [*internal_cface_number*] **of** *fixword*; { CJK font width ratio }
cface_fw_height: **array** [*internal_cface_number*] **of** *fixword*; { CJK font heigh ratio }
cface_fw_depth: **array** [*internal_cface_number*] **of** *fixword*; { CJK font depth ratio }
cface_csp_width: **array** [*internal_cface_number*] **of** *integer*; { CJK font c-space width }
cface_csp_shrink: **array** [*internal_cface_number*] **of** *integer*; { CJK font c-space shrink }
cface_csp_stretch: **array** [*internal_cface_number*] **of** *integer*; { CJK font c-space stretch }
cface_cesp_width: **array** [*internal_cface_number*] **of** *integer*; { CJK font ce-space width }
cface_cesp_shrink: **array** [*internal_cface_number*] **of** *integer*; { CJK font ce-space shrink }
cface_cesp_stretch: **array** [*internal_cface_number*] **of** *integer*; { CJK font ce-space stretch }
cface_fw_default_depth: *fixword*;

1475. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("PUXsetdefaultcface", *pux_set_default_cface*, *int_base* + *pux_default_cface_code*);

1476. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
pux_set_default_cface: *print_esc*("PUXsetdefaultcface"); { TCW }

1477. ⟨ Assignments 1220 ⟩ +≡
pux_set_default_cface: **begin** $p \leftarrow$ *cur_chr*; ⟨ Get the next non-blank non-call token 409 ⟩;
  if *cur_cmd* = *pux_set_cface* **then** *word_define*(*p*, *cur_chr*)
  else begin *print_err*("Here␣should␣put␣a␣CJK␣font␣face␣command.␣");
    print("The␣dafault␣CJK␣font␣face␣remains␣unchanged"); *error*;
    end;
  end;

1478. ⟨ PUTeX routines that will be used by TeX routines 1413 ⟩ +≡
procedure *reset_cface_cspace*(*face_num* : *integer*);
  begin *cface_csp_width*[*face_num*] ← *g_cspace_width*;
  cface_csp_shrink[*face_num*] ← *g_cspace_shrink*;
  cface_csp_stretch[*face_num*] ← *g_cspace_stretch*;
  end;

1479. ⟨ PUTeX routines that will be used by TeX routines 1413 ⟩ +≡
procedure *reset_cface_cespace*(*face_num* : *integer*);
  begin *cface_cesp_width*[*face_num*] ← *g_cespace_width*;
  cface_cesp_shrink[*face_num*] ← *g_cespace_shrink*;
  cface_cesp_stretch[*face_num*] ← *g_cespace_stretch*;
  end;

1480. Setup default and null CJK font faces.

⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
  cur_cface ← *null_cface*; *eq_type*(*cur_cface_loc*) ← *data*; *eq_level*(*cur_cface_loc*) ← *level_one*;
  cface_fw_default_depth ← *convfix*(*puxg_cface_depth*); *cface_ptr* ← *cface_base* + 1;
  cface[*null_cface*] ← "nullcface";
  cface_name[*null_cface*] ← "nullcjkface";
  cface_charset[*null_cface*] ← 0;
  cface_weight[*null_cface*] ← 400; { normal weight }
  cface_style[*null_cface*] ← 0;
  cface_fw_width[*null_cface*] ← 0;
  cface_fw_height[*null_cface*] ← 0;
  cface_fw_depth[*null_cface*] ← 0;
  reset_cface_cspace(*null_cface*);
  reset_cface_cespace(*null_cface*);

1481. The function *find_cface_num* searches the CJK font face definition table for the entry with the same identifier as *id*. The entry index is return if found; otherwise, the current value of *cface_ptr* is return.

define $cface\_found(\#) \equiv ((\#) < cface\_ptr)$

⟨ Declare the function called *find_cface_num* 1481 ⟩ ≡
function *find_cface_num*(*id* : *str_number*): *internal_cface_number*;
  label *done*;
  var *f*: *internal_cface_number*; { runs through existing faces }
  begin $f \leftarrow cface\_base$;
  while $(f < cface\_ptr)$ **do**
    begin if *str_eq_str*(*id*, *cface*[*f*]) **then goto** *done*;
    incr(*f*);
    end;
done: $find\_cface\_num \leftarrow f$;
  end;

This code is used in section 1260.

1482. ⟨ Declare subprocedures for *prefixed_command* 1218 ⟩ +≡
⟨ Declare the function called *print_fixword* 1440 ⟩
procedure *new_cface*(*a* : *small_number*);
label *done*, *done1*, *common_ending*;
var *u*: *pointer*; { user's chinese face identifier }
t: *str_number*; { name for the frozen font identifier }
id: *str_number*; { CJK font face identifier }
face_name: *str_number*; { CJK font face name }
charset: *integer*; { CJK font charset }
weight: *integer*; { CJK font weight }
style: *integer*; { CJK font style }
w: *integer*; { CJK font width ratio }
h: *integer*; { CJK font height ratio }
d: *integer*; { CJK font depth ratio }
fix_w: *fixword*; { CJK font width ratio }
fix_h: *fixword*; { CJK font height ratio }
fix_d: *fixword*; { CJK font depth ratio }
f: *internal_cface_number*; { runs through existing faces }
k: *integer*;
⟨ Other variables used by *new_cface* 1486 ⟩
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* ← `"CFACE"` **else** *t* ← *u* − *single_base*
else begin *old_setting* ← *selector*; *selector* ← *new_string*; *print*(`"CFACE"`); *print*(*u* − *active_base*);
selector ← *old_setting*; *str_room*(1); *t* ← *make_string*;
end;
define(*u*, *pux_set_cface*, *null_cface*); *scan_optional_equals*;
⟨ Setup variables before scanning CJK font face parameters 1483 ⟩;
⟨ Scan CJK font face identifier 1484 ⟩;
⟨ Scan CJK font face name 1485 ⟩;
⟨ Scan optional CJK font face definition parameters 1487 ⟩;
⟨ If the face name is missing, then ignore this face deinition 1495 ⟩;
⟨ If this Chinese face has already been loaded, then **goto** *common_ending* 1496 ⟩;
⟨ Setup this new Chinese face 1497 ⟩;
common_ending: *equiv*(*u*) ← *f*; *eqtb*[*cface_id_base* + *f*] ← *eqtb*[*u*]; *cface_id_text*(*f*) ← *t*;
end;

1483. ⟨ Setup variables before scanning CJK font face parameters 1483 ⟩ ≡
charset ← *pux_charset*; { set to the base charset of document }
w ← 1000; *h* ← 1000; *d* ← *puxg_cface_depth*;
weight ← 400; { normal weight }
style ← 0; { regular style }
if *puxg_rotate_ctext* ≠ 0 **then** *style* ← *style* + *rotated*;
f ← *null_cface*

This code is used in section 1482.

1484. ⟨ Scan CJK font face identifier 1484 ⟩ ≡
id ← *scan_name*;
if *id* > 0 **then**
  begin *f* ← *find_cface_num*(*id*);
  if (*f* < *cface_ptr*) **then**
    begin *flush_string*; *id* ← *cface*[*f*]; { for saving string pool sapce }
    f ← *null_cface*; *print_err*("The␣Chinese␣face␣id␣("); *print*(*id*); *print*(")␣is␣already␣used");
    error;
    end;
  end
else begin *print_err*("Missing␣CJK␣font␣face␣identifier"); *error*;
  end

This code is used in section 1482.

1485. ⟨ Scan CJK font face name 1485 ⟩ ≡
begin *face_name* ← *scan_name*;
if *face_name* > 0 **then**
  begin *k* ← *cface_base*;
  while (*k* < *cface_ptr*) **do**
    begin if *str_eq_str*(*face_name*, *cface_name*[*k*]) **then**
      begin *flush_string*; *face_name* ← *cface_name*[*k*]; *f* ← *k*; **goto** *done1*;
      end;
    incr(*k*);
    end;
  end
else begin *print_err*("Missing␣CJK␣font␣face␣name"); *error*; *face_name* ← *cface_name*[*null_cface*];
  f ← *null_cface*;
  end;
done1: **end**

This code is used in section 1482.

1486. ⟨ Other variables used by *new_cface* 1486 ⟩ ≡
i_flag: *boolean*; { italic flag }
u_flag: *boolean*; { underline flag }
s_flag: *boolean*; { strikeout flag }
r_flag: *boolean*; { rotation flag }
v_flag: *boolean*; { inverse flag }
more_param: *boolean*; { have more parameters to come }

This code is used in section 1482.

1487. ⟨Scan optional CJK font face definition parameters 1487⟩ ≡

```
i_flag ← false; u_flag ← false; s_flag ← false;
r_flag ← false; v_flag ← false;
more_param ← true;
while more_param do
  begin ⟨Get the next non-blank non-call token 409⟩;
  if cur_cmd = letter then
    case cur_chr of
    "c", "C": ⟨Scan the CJK font charset 1488⟩;
    "w", "W": ⟨Scan the CJK font width 1489⟩;
    "h", "H": ⟨Scan the CJK font height 1490⟩;
    "d", "D": ⟨Scan the CJK font depth 1491⟩;
    "t", "T": ⟨Scan the CJK font weight 1492⟩;
    "s", "S": ⟨Scan the CJK font style 1493⟩;
    othercases more_param ← false;
    endcases
  else more_param ← false;
  end;
back_input
```

This code is used in section 1482.

1488.

⟨Scan the CJK font charset 1488⟩ ≡

```
begin scan_optional_equals;
scan_int;
if (cur_val < 0) ∨ (cur_val > 255) then
  begin print_err("Improper␣`charset'␣value␣("); print_int(charset);
  print("),␣replaced␣by␣default␣charset");
  help2("I␣can␣only␣handle␣nonnegative␣charset␣value␣up␣to␣255,")
  ("so␣I've␣changed␣what␣you␣said␣to␣default␣charset."); error;
  end
else charset ← cur_val;
end
```

This code is used in section 1487.

1489.

⟨Scan the CJK font width 1489⟩ ≡

```
begin scan_optional_equals;
scan_int; w ← cur_val;
if (w ≤ 0) ∨ (w > 1000) then
  begin print_err("Improper␣`width'␣value␣("); print_int(w); print("),␣replaced␣by␣1000");
  help2("I␣can␣only␣handle␣fonts␣at␣positive␣width␣ratio␣that␣are␣less")
  ("than␣or␣equal␣to␣1000,␣so␣I've␣changed␣what␣you␣said␣to␣1000."); error; w ← 1000;
  end;
end
```

This code is used in section 1487.

1490. ⟨Scan the CJK font height 1490⟩ ≡
begin *scan_optional_equals*;
scan_int; $h \leftarrow cur\_val$;
if $(h \le 0) \lor (h > 1000)$ **then**
begin *print_err*("Improper␣`height␣value␣("); *print_int*(*h*); *print*("),␣replaced␣by␣1000");
help2("I␣can␣only␣handle␣fonts␣at␣positive␣height␣ratio␣that␣are␣less")
("than␣or␣equal␣to␣1000,␣so␣I´ve␣changed␣what␣you␣said␣to␣1000."); *error*; $h \leftarrow 1000$;
end;
end

This code is used in section 1487.

1491. ⟨Scan the CJK font depth 1491⟩ ≡
begin *scan_optional_equals*;
scan_int; $d \leftarrow cur\_val$;
if $(d < 0) \lor (d > 1000)$ **then**
begin *print_err*("Improper␣`depth´␣value␣("); *print_int*(*d*); *print*("),␣replaced␣by␣0.2");
help3("I␣can␣only␣handle␣fonts␣at␣nonegative␣depth␣ratio␣that␣are␣less")
("than␣or␣equal␣to␣1000,␣so␣I´ve␣changed␣what␣you␣said␣to")
("the␣current␣\puxgCfaceDepth␣value."); *error*; $d \leftarrow puxg\_cface\_depth$;
end;
end

This code is used in section 1487.

1492. ⟨Scan the CJK font weight 1492⟩ ≡
begin *scan_optional_equals*;
scan_int; $weight \leftarrow cur\_val$;
if $(weight < 0) \lor (weight > 1000)$ **then**
begin *print_err*("Illegal␣CJK␣font␣weight␣has␣been␣changed␣to␣400");
help1("The␣font␣weight␣must␣be␣between␣1␣and␣1000."); *int_error*(*cur_val*); $weight \leftarrow 400$;
{ normal weight }
end;
end

This code is used in section 1487.

1493.

⟨ Scan the CJK font style 1493 ⟩ ≡
begin *scan_optional_equals*;
⟨ Get the next non-blank non-call token 409 ⟩;
if *cur_cmd* = *letter* **then**
case *cur_chr* **of**
"i", "I": **if** ¬*i_flag* **then**
begin *style* ← *style* + *italic*; *i_flag* ← *true*;
end;
"u", "U": **if** ¬*u_flag* **then**
begin *style* ← *style* + *underline*; *u_flag* ← *true*;
end;
"s", "S": **if** ¬*s_flag* **then**
begin *style* ← *style* + *strikeout*; *s_flag* ← *true*;
end;
"r", "R": **if** ¬*r_flag* **then** ⟨ Set CJK font rotation style 1494 ⟩;
"v", "V": **if** ¬*v_flag* **then**
begin *style* ← *style* + *inverse*; *v_flag* ← *true*;
end;
othercases
begin *print_err*("Illegal␣CJK␣font␣style␣setting␣has␣been␣ignored");
print("␣("); *print*(*cur_chr*); *print*(")"); *back_error*;
{ fix the case when cur_chr is a double-byte char }
help2("The␣CJK␣font␣style␣setting␣should␣use␣characters:")
("i:italic,␣u:underline,␣s:strikeout,␣r:rotated,␣v:reversed");
end;
endcases;
end

This code is used in section 1487.

1494. ⟨ Set CJK font rotation style 1494 ⟩ ≡
begin if *puxg_rotate_ctext* ≠ 0 **then** *style* ← *style* − *rotated*
else *style* ← *style* + *rotated*;
r_flag ← *true*;
end

This code is used in section 1493.

1495. ⟨ If the face name is missing, then ignore this face deinition 1495 ⟩ ≡
if f = *null_cface* **then goto** *common_ending*

This code is used in section 1482.

1496. ⟨ If this Chinese face has already been loaded, then **goto** *common_ending* 1496 ⟩ ≡
fix_w ← *convfix*(w); *fix_h* ← *convfix*(h); *fix_d* ← *convfix*(d);
if f ≠ *null_cface* **then**
if *weight* = *cface_weight*[f] ∧ *style* = *cface_style*[f] **then**
if *fix_w* = *cface_fw_width*[f] ∧ *fix_h* = *cface_fw_height*[f] ∧ *fix_d* = *cface_fw_depth*[f] **then**
goto *common_ending*

This code is used in section 1482.

1497.

⟨ Setup this new Chinese face 1497 ⟩ ≡

if $cface\_ptr \le max\_cface$ **then**
begin $f \leftarrow cface\_ptr$; $cface[f] \leftarrow id$; $cface\_name[f] \leftarrow face\_name$;
$cface\_charset[f] \leftarrow charset$;
$cface\_weight[f] \leftarrow weight$;
$cface\_style[f] \leftarrow style$;
if $style$ **mod** $2 = 1$ **then**
begin $cface\_fw\_width[f] \leftarrow fix\_w$; $cface\_fw\_height[f] \leftarrow fix\_h$;
end
else begin $cface\_fw\_width[f] \leftarrow fix\_h$; $cface\_fw\_height[f] \leftarrow fix\_w$;
end;
$cface\_fw\_depth[f] \leftarrow fix\_d$;
$reset\_cface\_cspace(f)$;
$reset\_cface\_cespace(f)$;
$incr(cface\_ptr)$;
end
else begin $f \leftarrow null\_cface$; $print\_err$("CJK␣font␣Face␣definition␣table␣overflow"); $error$;
end

This code is used in section 1482.

1498. CJK font definition table.

1499. ⟨ Constants in the outer block 11 ⟩ +≡
cfont_base = *font_max_limit* + 1; { CJK font base }
cfont_max = *font_max_limit* + 1 + *cfont_max_limit*; { maximum internal chinese font number }

1500.
⟨ Types in the outer block 18 ⟩ +≡
internal_cfont_number = *cfont_base* . . *cfont_max*;

1501. ⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
cur_cfont ← *default_cfont*; *eq_type*(*cur_cfont_loc*) ← *data*; *eq_level*(*cur_cfont_loc*) ← *level_one*;

1502. ⟨ Global variables 13 ⟩ +≡
cfont_ptr: *internal_cfont_number*;
cfont_face: **array** [*internal_cfont_number*] **of** *internal_cface_number*; { CJK font face name }
cfont_dsize: **array** [*internal_cfont_number*] **of** *scaled*; { CJK font design size }
cfont_size: **array** [*internal_cfont_number*] **of** *scaled*; { CJK font size }
cfont_width: **array** [*internal_cfont_number*] **of** *scaled*; { CJK font width }
cfont_height: **array** [*internal_cfont_number*] **of** *scaled*; { CJK font heigh }
cfont_depth: **array** [*internal_cfont_number*] **of** *scaled*; { CJK font depth }
cfont_glue_spec: **array** [*internal_cfont_number*] **of** *pointer*; { CJK font inter-character space }
cfont_ceglue_spec: **array** [*internal_cfont_number*] **of** *pointer*; { CJK font inter-character space }
cfont_used: **array** [*internal_cfont_number*] **of** *boolean*;
{ has a character from this chinese font actually appeared in the output? }

1503. ⟨ Set initial values of key variables 21 ⟩ +≡
for *k* ← *cfont_base* **to** *cfont_max* **do** *cfont_used*[*k*] ← *false*;
cfont_face[*null_cfont*] ← *null_cface*; *cfont_dsize*[*null_cfont*] ← 0; *cfont_size*[*null_cfont*] ← 0;
cfont_width[*null_cfont*] ← 0; *cfont_height*[*null_cfont*] ← 0; *cfont_depth*[*null_cfont*] ← 0;

1504. ⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
cfont_ptr ← *default_cfont*;

1505. ⟨ Declare PUTeX subprocedures for *prefixed_command* 1505 ⟩ ≡
procedure *set_cglue_spec*(*n* : *integer*);
var *cface_num*: *integer*;
begin *cface_num* ← *cfont_face*[*n*];
width(*cfont_glue_spec*[*n*]) ← *xn_over_d*(*cfont_size*[*n*], *cface_csp_width*[*cface_num*], 1000);
shrink(*cfont_glue_spec*[*n*]) ← *xn_over_d*(*cfont_size*[*n*], *cface_csp_shrink*[*cface_num*], 1000);
stretch(*cfont_glue_spec*[*n*]) ← *xn_over_d*(*cfont_size*[*n*], *cface_csp_stretch*[*cface_num*], 1000);
end;

See also sections 1506 and 1525.
This code is used in section 1214.

1506. ⟨Declare PUTeX subprocedures for *prefixed_command* 1505⟩ +≡
procedure *set_ceglue_spec*(*n* : *integer*);
  var *cface_num*: *integer*;
  begin *cface_num* ← *cfont_face*[*n*];
  width(*cfont_ceglue_spec*[*n*]) ← *xn_over_d*(*cfont_size*[*n*], *cface_cesp_width*[*cface_num*], 1000);
  shrink(*cfont_ceglue_spec*[*n*]) ← *xn_over_d*(*cfont_size*[*n*], *cface_cesp_shrink*[*cface_num*], 1000);
  stretch(*cfont_ceglue_spec*[*n*]) ← *xn_over_d*(*cfont_size*[*n*], *cface_cesp_stretch*[*cface_num*], 1000);
  end;

1507. ⟨Initialization of global variables done in the *main_control* procedure 1449⟩ +≡
  cfont_glue_spec[*null_cfont*] ← *new_spec*(*zero_glue*); *cfont_ceglue_spec*[*null_cfont*] ← *new_spec*(*zero_glue*);

1508. ⟨Other local variables used by procedure *new_font* 1508⟩ ≡
face_id: *str_number*; { Chinese face name fetched from *font* command }
jj: *internal_cface_number*;
cface_num: *internal_cface_number*;
ds: *integer*;
dsize: *scaled*;
size: *scaled*;
This code is used in section 1260.

1509.
⟨Define a CJK font and then goto *common_ending* 1509⟩ ≡
  begin *define*(*u*, *set_cfont*, *null_cfont*); *cface_num* ← *pux_default_cface*;
  ⟨Fetch the Chinese face name 1510⟩;
  ⟨Fetch the font design size and compute font 'at' size 1511⟩;
  ⟨If this CJK font has already been loaded, set *f* to the internal CJK font number and **goto** common_ending 1513⟩;
  f ← *make_cfont*(*cface_num*, *dsize*, *size*);
  goto *common_ending*;
  end;
This code is used in section 1260.

1510.
  define *is_letter*(#) ≡ ((# ≥ "A" ∧ # ≤ "Z") ∨ (# ≥ "a" ∧ # ≤ "z"))

⟨Fetch the Chinese face name 1510⟩ ≡
  jj ← *j*; *j* ← *j* + 5; { skip the prefix 'CFONT' }
  while *is_letter*(*str_pool*[*j*]) **do** { fixme for wchar }
    begin *append_char*(*str_pool*[*j*]); *incr*(*j*);
    end;
  if *pool_ptr* ≠ *str_start*[*str_ptr*] **then**
    begin *face_id* ← *make_string*;
    cface_num ← *find_cface_num*(*face_id*); *flush_string*;
    end
  else begin *print_err*("Missing␣Chinese␣face␣identifier"); *error*;
    end;
This code is used in section 1509.

1511.

define *is_digit*(#) ≡ (# ≥ "0" ∧ # ≤ "9")

⟨ Fetch the font design size and compute font 'at' size 1511 ⟩ ≡
ds ← 0;
while *is_digit*(*str_pool*[*j*]) **do**
begin *ds* ← *ds* ∗ 10 + (*str_pool*[*j*] − "0"); *incr*(*j*);
end;
if *ds* = 0 **then**
begin *print_err*("Missing␣CJK␣font␣size␣specification,␣replaced␣by␣10pt"); *ds* ← 10;
{ set to default size: 10pt }
error;
end;
dsize ← *mult_integers*(*ds*, *unity*);
if *s* = −1000 **then** *size* ← *dsize*
else if *s* ≥ 0 **then** *size* ← *s*
else *size* ← *xn_over_d*(*dsize*, −*s*, 1000);

This code is used in section 1509.

1512.

define *defined_cfont*(#) ≡ (#) < *cfont_ptr*
define *undefined_cfont*(#) ≡ (#) = *cfont_ptr*

⟨ Declare the procedure called *check_cfont* 1512 ⟩ ≡
function *check_cfont*(*cface_num* : *internal_cface_number*; *size* : *scaled*): *internal_cfont_number*;
label *done*;
var *f*: *internal_cfont_number*;
begin *f* ← *cfont_base* + 1;
while (*f* < *cfont_ptr*) **do**
begin if *cface_num* = *cfont_face*[*f*] ∧ *size* = *cfont_size*[*f*] **then goto** *done*;
incr(*f*);
end;
done: *check_cfont* ← *f*;
end;

This code is used in section 1260.

1513. ⟨ If this CJK font has already been loaded, set *f* to the internal CJK font number and **goto** common_ending 1513 ⟩ ≡
f ← *check_cfont*(*cface_num*, *size*);
if *defined_cfont*(*f*) **then goto** *common_ending*;

This code is used in section 1509.

1514.

⟨ Declare the procedure called *make_cfont* 1514 ⟩ ≡

function *make_cfont*(*cfn* : *internal_cface_number*; *dsize*, *size* : *scaled*): *internal_cfont_number*;
  begin if *cfont_ptr* ≤ *cfont_max* **then**
    begin *cfont_face*[*cfont_ptr*] ← *cfn*;
    cfont_dsize[*cfont_ptr*] ← *dsize*;
    cfont_size[*cfont_ptr*] ← *size*;
    cfont_width[*cfont_ptr*] ← *fw_times_sd*(*cface_fw_width*[*cfn*], *size*);
    cfont_height[*cfont_ptr*] ← *fw_times_sd*(*cface_fw_height*[*cfn*], *size*);
    cfont_depth[*cfont_ptr*] ← *fw_times_sd*(*cface_fw_depth*[*cfn*], *size*);
    cfont_glue_spec[*cfont_ptr*] ← *new_spec*(*zero_glue*); *set_cglue_spec*(*cfont_ptr*);
    cfont_ceglue_spec[*cfont_ptr*] ← *new_spec*(*zero_glue*); *set_ceglue_spec*(*cfont_ptr*);
    make_cfont ← *cfont_ptr*;
    incr(*cfont_ptr*);
    end
  else begin *print_err*("CJK␣font␣table␣overflow"); *error*;
    end
  end;

This code is used in section 1260.

1515. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡

set_cfont: **begin** *print*("select␣CJK␣font␣"); *slow_print*(*cface*[*cfont_face*[*chr_code*]]); *print*("␣at␣(");
  print_scaled(*cfont_size*[*chr_code*]); *print*("pt"); *print*(")");
  end;

1516. Matching faces.

define *min_ectbl* $= 0$
define *max_ectbl* $= 255$

1517. ⟨ Types in the outer block 18 ⟩ +≡
internal_ectbl_number = *min_ectbl* . . *max_ectbl*;

1518. ⟨ Global variables 13 ⟩ +≡
ectbl_eface_name: **array** [*internal_ectbl_number*] **of** *str_number*; { the table of English face names }
ectbl_ptr: *internal_ectbl_number*; { index to the first unused entry }

1519. *ectbl_cface_num* table entries are already initialized in section 232.

⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
ectbl_ptr ← *min_ectbl*; *equiv*(*ectbl_cface_num_base*) ← *null_cface*; *eq_type*(*ectbl_cface_num_base*) ← *data*;
eq_level(*ectbl_cface_num_base*) ← *level_one*;
for k ← *ectbl_cface_num_base* + 1 **to** *font_matching_table_base* − 1 **do** *eqtb*[k] ← *eqtb*[*ectbl_cface_num_base*];

1520. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive(`"PUXfacematch"`, *pux_face_match*, 0);

1521. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
pux_face_match: *print_esc*(`"PUXfacematch"`);

1522. ⟨ Assignments 1220 ⟩ +≡
pux_face_match: *match_ec_face*(a);

1523. The function *find_ec_num* lookup the *ectbl_eface_name* table for the name *eface_name*. It returns the index to the name if the name exits; otherwose, it returns the current value of *ectbl_ptr*.

define *ectbl_found*(#) ≡ ((#) < *ectbl_ptr*)

⟨ Declare the function called *find_ec_num* 1523 ⟩ ≡
function *find_ec_num*(*eface_name* : *str_number*): *internal_ectbl_number*;
  label *done*;
  var k: *integer*;
  begin k ← *min_ectbl*;
  while k < *ectbl_ptr* **do**
    begin if *str_eq_str*(*eface_name*, *ectbl_eface_name*[k]) **then goto** *done*;
    incr(k);
    end;
done: *find_ec_num* ← k;
  end;

This code is used in section 1526.

1524.

⟨ Declare subprocedures for *prefixed_command* 1218 ⟩ +≡
procedure *make_cfont_id*(f : *internal_cfont_number*; a : *small_number*);
  var i: 0 . . 23; m: *integer*; u: *pointer*; t: *str_number*; n: *integer*;
  begin *buffer*[*buf_size* + 1] ← ´C´; *buffer*[*buf_size* + 2] ← ´F´; *buffer*[*buf_size* + 3] ← ´O´;
  buffer[*buf_size* + 4] ← ´N´; *buffer*[*buf_size* + 5] ← ´T´; m ← *buf_size* + 6; n ← f; i ← 0;
  repeat *dig*[i] ← n **mod** 10; n ← n **div** 10; *incr*(i);
  until $n = 0$;
  while $i > 0$ **do** { append design size }
    begin *decr*(i); *buffer*[m] ← "0" + *dig*[i]; *incr*(m);
    end;
  no_new_control_sequence ← *false*; u ← *id_lookup*(*buf_size* + 1, m − *buf_size* − 1);
  no_new_control_sequence ← *true*; t ← *text*(u); *define*(u, *set_cfont*, f); *eqtb*[*font_id_base* + f] ← *eqtb*[u];
  font_id_text(f) ← t;
  end;

1525. ⟨ Declare PUTeX subprocedures for *prefixed_command* 1505 ⟩ +≡
function *fetch_efont_face*(*efont_name* : *str_number*): *str_number*;
  var k: *integer*;
    p: *pool_pointer*; s: *str_number*;
  begin p ← *str_start*[*efont_name* + 1] − 1; { last char position of efont_name }
  while *is_digit*(*str_pool*[p]) **do** *decr*(p); { assumed that the TeX font name has letters }
  k ← *str_start*[*efont_name*];
  while $k \le p$ **do**
    begin *append_char*(*str_pool*[k]); *incr*(k);
    end;
  s ← *make_string*; *fetch_efont_face* ← s;
  end;

1526.

```
⟨Declare subprocedures for prefixed_command 1218⟩ +≡
  ⟨Declare the function called find_ec_num 1523⟩
procedure match_ec_face(a : small_number);
  label done1, done2, exit;
  var k, f: integer; eface_name, efname, efont_name, cface_id: str_number;
    cfont_num: internal_cfont_number; cface_num: internal_cface_number; err: boolean;
  begin err ← false; f ← ectbl_ptr;
  eface_name ← scan_name;
  if cur_cmd = pux_set_cface then eface_name ← fetch_efont_face(font_name[cur_font])
        { should be flushed later }
  else if eface_name = 0 then
      begin print_err("Missing␣a␣TeX␣face␣name"); error; goto exit;
      end;
  f ← find_ec_num(eface_name);
  if ectbl_found(f) then
    begin { it is already in the ectbl_eface_name table }
    flush_string; eface_name ← ectbl_eface_name[f]
    end;
  if cur_cmd = pux_set_cface then
    begin { the second form: match face of current efont }
    cface_num ← cur_chr; ⟨Define the cur_cfont according to cur_font and cface_num 1527⟩;
    end
  else ⟨Fetch a Chinese face id 1528⟩;
  ⟨Add this face matching 1529⟩;
exit: end;
```

1527. ⟨Define the *cur_cfont* according to *cur_font* and *cface_num* 1527⟩ ≡

```
  cfont_num ← check_cfont(cface_num, font_size[cur_font]);
  if undefined_cfont(cfont_num) then
    begin cfont_num ← make_cfont(cface_num, font_dsize[cur_font], font_size[cur_font]);
    make_cfont_id(cfont_num, a);
    end;
  define(cur_cfont_loc, data, cfont_num)
```

This code is used in sections 1526, 1535, and 1541.

1528. ⟨Fetch a Chinese face id 1528⟩ ≡

```
  begin ⟨Get the next non-blank non-call token 409⟩;
  if cur_cmd = pux_set_cface then cface_num ← cur_chr
  else begin print_err("Missing␣a␣CJK␣font␣face␣identifier"); err ← true; error;
    cface_num ← pux_default_cface;
    end
  end
```

This code is used in section 1526.

1529. ⟨Add this face matching 1529⟩ ≡

```
if f > max_ectbl then
  begin print_err("Font␣face␣matching␣table␣overflow"); err ← true; error;
  end;
if ¬err then
  begin define(ectbl_cface_num_base + f, data, cface_num);
  if f = ectbl_ptr then
    begin { add this new eface name the the eface_name table }
    ectbl_eface_name[f] ← eface_name; incr(ectbl_ptr);
    end;
  end
```

This code is used in section 1526.

1530. ⟨Declare subprocedures for *prefixed_command* 1218⟩ +≡

```
function lookup_cface(efont_name : str_number): internal_cface_number;
  var k: integer;
    cface_num: internal_cface_number;
    eface_name: str_number;
  begin eface_name ← fetch_efont_face(efont_name); k ← find_ec_num(eface_name); flush_string;
  if ectbl_found(k) then cface_num ← ectbl_cface_num(k)
  else cface_num ← pux_default_cface;
  lookup_cface ← cface_num;
  end;
```

1531. Font matching.

1532. ⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
equiv(*font_matching_table_base*) ← *null_cfont*; *eq_type*(*font_matching_table_base*) ← *data*;
eq_level(*font_matching_table_base*) ← *level_one*;
for k ← *font_matching_table_base* + 1 **to** *math_font_base* − 1 **do** *eqtb*[k] ← *eqtb*[*font_matching_table_base*];

1533. ⟨ Put each of TeX's primitives into the hash table 226 ⟩ +≡
primitive(`"PUXfontmatch"`, *pux_font_match*, 0);

1534. ⟨ Assignments 1220 ⟩ +≡
pux_font_match: *match_ec_font*(a);

1535. ⟨ Declare subprocedures for *prefixed_command* 1218 ⟩ +≡

```
procedure match_ec_font(a : small_number);
  label done;
  var efont_num: internal_font_number; cfont_num: internal_cfont_number;
    cface_num: internal_cface_number;
  begin ⟨ Get the next non-blank non-call token 409 ⟩;
  if cur_cmd = pux_set_cface then  { the first form }
    begin efont_num ← cur_font; cface_num ← cur_chr;
    ⟨ Define the cur_cfont according to cur_font and cface_num 1527 ⟩;
    goto done;
    end;
  if cur_cmd = set_font then  { the second form }
    efont_num ← cur_chr
  else begin print_err("Missing␣Tex␣font␣identifier");
    help2("I␣was␣looking␣for␣a␣control␣sequence␣whose")
    ("current␣meaning␣has␣been␣defined␣by␣\font."); back_error; efont_num ← null_font;
    end;
  ⟨ Get the next non-blank non-call token 409 ⟩;
  if cur_cmd = set_cfont then cfont_num ← cur_chr
  else begin print_err("Missing␣CJK␣font␣identifier");
    help2("I␣was␣looking␣for␣a␣control␣sequence␣whose")
    ("current␣meaning␣has␣been␣defined␣by␣\cfont."); back_error; cfont_num ← null_cfont;
    end;
done: if efont_num ≠ null_font ∧ cfont_num ≠ null_cfont then
    define(font_matching_table_base + efont_num − font_base, data, cfont_num);
  end;
```

1536. ⟨ Other variables used by the procedure *prefixed_command* 1415 ⟩ +≡
cface_num: *internal_cface_number*;
cfont_num: *internal_cfont_number*;

1537. ⟨ Set the matching CJK font 1537 ⟩ ≡
cfont_num ← *font_matching_table*(*cur_chr*);
if *cfont_num* = *null_cfont* **then**
begin { efont not mapped }
if *cur_cface* = *null_cface* **then** *cface_num* ← *lookup_cface*(*font_name*[*cur_chr*])
else *cface_num* ← *cur_cface*;
⟨ Build a CJK font according to *cur_chr* and *cface_num* if it is not exist 1538 ⟩;
end
else if *cur_cface* ≠ *null_cface* ∧ *cfont_face*[*cfont_num*] ≠ *cur_cface* **then**
begin *cface_num* ← *cur_cface*;
⟨ Build a CJK font according to *cur_chr* and *cface_num* if it is not exist 1538 ⟩;
end;
define(*cur_cfont_loc*, *data*, *cfont_num*)

This code is used in section 1220.

1538. ⟨ Build a CJK font according to *cur_chr* and *cface_num* if it is not exist 1538 ⟩ ≡
cfont_num ← *check_cfont*(*cface_num*, *font_size*[*cur_chr*]);
if *undefined_cfont*(*cfont_num*) **then**
begin *cfont_num* ← *make_cfont*(*cface_num*, *font_dsize*[*cur_chr*], *font_size*[*cur_chr*]);
make_cfont_id(*cfont_num*, *a*);
end

This code is used in sections 1537 and 1537.

1539. ⟨ Assignments 1220 ⟩ +≡
set_cfont: *define*(*cur_cfont_loc*, *data*, *cur_chr*);

1540. ⟨ Other variables used by the procedure *prefixed_command* 1415 ⟩ +≡
cface_id: *str_number*;

1541. ⟨ Assignments 1220 ⟩ +≡
pux_set_cface: **begin** *cface_num* ← *cur_chr*;
if *cface_num* ≠ *cfont_face*[*cur_cfont*] **then**
begin ⟨ Define the *cur_cfont* according to *cur_font* and *cface_num* 1527 ⟩;
end;
define(*cur_cface_loc*, *data*, *cface_num*);
end;

1542. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("puxgRotateCtext", *puxg_assign_flag*, *int_base* + *puxg_rotate_ctext_code*);
primitive("puxXspace", *puxg_assign_int*, *int_base* + *pux_xspace_code*);
primitive("puxCJKcharOther", *puxg_assign_int*, *int_base* + *pux_wcharother_code*);
primitive("puxCJKinput", *puxg_assign_int*, *int_base* + *pux_CJKinput_code*);
primitive("puxCharSet", *puxg_assign_int*, *int_base* + *pux_charset_code*);
primitive("puxgCfaceDepth", *puxg_assign_int*, *int_base* + *puxg_cface_depth_code*);

1543. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
puxg_assign_flag: **if** *chr_code* = *puxg_rotate_ctext_code* + *int_base* **then** *print_esc*("puxgRotateCtext");
puxg_assign_int: **if** *chr_code* = *pux_xspace_code* + *int_base* **then** *print_esc*("puxXspace")
else if *chr_code* = *pux_wcharother_code* + *int_base* **then** *print_esc*("puxCJKcharOther")
else if *chr_code* = *pux_CJKinput_code* + *int_base* **then** *print_esc*("puxCJKinput")
else if *chr_code* = *pux_charset_code* + *int_base* **then** *print_esc*("puxCharSet")
else if *chr_code* = *puxg_cface_depth_code* + *int_base* **then** *print_esc*("puxgCfaceDepth");

1544. ⟨ Assignments 1220 ⟩ +≡

```
puxg_assign_flag: begin p ← cur_chr; scan_optional_equals; scan_int;
  if cur_val = 0 ∧ eqtb[p].int ≠ 0 then
    begin print_err("Reset␣a␣PUTeX␣global␣parameter␣is␣not␣allowed␣here");
    help2("If␣a␣PUTeX␣global␣parameter␣was␣set␣to␣be␣a␣nonzero␣value,")
    ("it␣can´t␣be␣reset␣to␣be␣zero␣again"); error;
    end
  else begin if p = puxg_rotate_ctext_code + int_base then ⟨ Handle the command puxgRotateCtext 1545 ⟩;
    word_define(p, cur_val);
    end;
  end;
```

1545. ⟨ Handle the command puxgRotateCtext 1545 ⟩ ≡

```
  if puxg_rotate_ctext = 0 ∧ cur_val ≠ 0 then
    begin n ← cface_base;
    while n < cface_ptr do
      begin if cface_style[n] mod 2 = 1 then cface_style[n] ← cface_style[n] − rotated
      else cface_style[n] ← cface_style[n] + rotated;
      incr(n);
      end;
    end
```

This code is used in section 1544.

1546. ⟨ Assignments 1220 ⟩ +≡

```
puxg_assign_int: begin p ← cur_chr; q ← p − int_base; scan_optional_equals; scan_int;
  if cur_val < 0 then
    begin print_err("Negative␣"); print_param(p − int_base); print("␣value␣("); print_int(cur_val);
    print("),␣it␣remains␣unchanged"); help1("This␣PUTeX␣parameter␣can´t␣be␣negative."); error;
    end
  else if q = pux_charset_code ∧ cur_val > 255 then
      begin print_err("Too␣large␣"); print_param(q); print("␣value␣("); print_int(cur_val);
      print("),␣it␣remains␣unchanged");
      help1("The␣value␣of␣document␣charset␣should␣be␣in␣the␣range␣0..255."); error;
      end
    else begin case q of
      pux_xspace_code, pux_wcharother_code, pux_CJKinput_code, pux_charset_code: word_define(p, cur_val);
      puxg_cface_depth_code: if cur_val ≠ eqtb[p].int then
          ⟨ Set PUTeX global parameter puxgCfaceDepth 1547 ⟩;
        othercases begin print_err("Unknow␣integer␣parameter!"); error;
          end;
      endcases
      end;
  end;
```

1547. ⟨Set PUTeX global parameter *puxgCfaceDepth* 1547⟩ ≡

begin if *cur_val* > 1000 **then**
  begin *print_err*("Improper␣`depth´␣value␣("); *print_int*(*cur_val*); *print*(").␣It␣is␣ignored");
  error;
  end
else begin *word_define*(*p*, *cur_val*); *cface_fw_default_depth* ← *convfix*(*puxg_cface_depth*); *n* ← *cface_base*;
  while *n* < *cface_ptr* **do**
    begin *cface_fw_depth*[*n*] ← *cface_fw_default_depth*; *incr*(*n*);
    end;
  n ← *cfont_base* + 1;
  while *n* < *cfont_ptr* **do**
    begin *cfont_depth*[*n*] ← *fw_times_sd*(*cface_fw_depth*[*cfont_face*[*n*]], *cfont_size*[*n*]); *incr*(*n*);
    end;
  end;
end

This code is used in section 1546.

1548.

define *pux_set_cface_csp* = 0
define *pux_set_cface_cesp* = 1
define *pux_set_cface_depth* = 2

⟨Put each of TEX's primitives into the hash table 226⟩ +≡
primitive("PUXcfacecspace", *pux_set_cface_attrib*, *pux_set_cface_csp*);
primitive("PUXcfacecespace", *pux_set_cface_attrib*, *pux_set_cface_cesp*);
primitive("PUXcfacedepth", *pux_set_cface_attrib*, *pux_set_cface_depth*);

1549. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡

pux_set_cface_attrib: **begin case** *chr_code* **of**
  pux_set_cface_csp: *print_esc*("PUXcfacecspace");
  pux_set_cface_cesp: *print_esc*("PUXcfacecespace");
  pux_set_cface_depth: *print_esc*("PUXcfacedepth");
  endcases;
  end;

1550. ⟨Assignments 1220⟩ +≡

pux_set_cface_attrib: **begin** *p* ← *cur_chr*; ⟨Get the next non-blank non-call token 409⟩;
  if *cur_cmd* = *pux_set_cface* **then** *cface_num* ← *cur_chr*
  else begin *cface_num* ← *null_cface*; *print_err*("Missing␣a␣CJK␣font␣face␣identifier"); *error*;
    end;
  scan_optional_equals;
  if *p* = *pux_set_cface_csp* ∨ *p* = *pux_set_cface_cesp* **then** ⟨Scan spacing dimension of CJK font face 1552⟩
  else *scan_int*;
  if *cface_num* ≠ *null_cface* **then**
    begin if *p* = *pux_set_cface_csp* **then** ⟨Modify the cspace factor of the specified chinese face 1553⟩
    else if *p* = *pux_set_cface_cesp* **then** ⟨Modify the cespace factor of the specified chinese face 1554⟩
      else if *p* = *pux_set_cface_depth* **then** ⟨Modify the depth factor of the specified chinese face 1555⟩;
    end;
  end;

1551. ⟨ Other variables used by the procedure *prefixed_command* 1415 ⟩ +≡
width_value: *integer*; { width of space }
stretch_value: *integer*; { stretch of space }
shrink_value: *integer*; { shrink of space }

1552.

define *puxg_set_cspace* = 0
define *puxg_set_cespace* = 1

⟨ Scan spacing dimension of CJK font face 1552 ⟩ ≡
begin *scan_optional_equals*; *scan_int*; *width_value* ← *cur_val*;
if *scan_keyword*("plus") **then**
  begin *scan_int*; *stretch_value* ← *cur_val*;
  end
else { make stretch value compatible to PUTEX3 }
if *width_value* $< 250 \land p =$ *puxg_set_cspace* **then** *stretch_value* ← 125
else *stretch_value* ← *width_value*/2;
if *scan_keyword*("minus") **then**
  begin *scan_int*; *shrink_value* ← *cur_val*;
  end
else { make shrink value compatible to PUTEX3 }
if *width_value* > 0 **then** *shrink_value* ← *width_value* **div** 3
else *shrink_value* ← −*width_value* **div** 3;
end

This code is used in sections 1550 and 1563.

1553. ⟨ Modify the cspace factor of the specified chinese face 1553 ⟩ ≡
begin if *cface_csp_width*[*cface_num*] ≠ *width_value* ∨ *cface_csp_stretch*[*cface_num*] ≠ *stretch_value* ∨ *cface_csp_shrink*[*cface_num*] ≠ *shrink_value* **then**
  begin *cface_csp_width*[*cface_num*] ← *width_value*; *cface_csp_stretch*[*cface_num*] ← *stretch_value*;
  cface_csp_shrink[*cface_num*] ← *shrink_value*; *n* ← *cfont_base* + 1;
  while *n* < *cfont_ptr* **do**
    begin if *cface_num* = *cfont_face*[*n*] **then** *set_cglue_spec*(*n*);
    incr(*n*);
    end;
  end;
end

This code is used in section 1550.

1554. ⟨ Modify the cespace factor of the specified chinese face 1554 ⟩ ≡
begin if *cface_cesp_width*[*cface_num*] ≠ *width_value* ∨ *cface_cesp_stretch*[*cface_num*] ≠ *stretch_value* ∨ *cface_cesp_shrink*[*cface_num*] ≠ *shrink_value* **then**
  begin *cface_cesp_width*[*cface_num*] ← *width_value*; *cface_cesp_stretch*[*cface_num*] ← *stretch_value*;
  cface_cesp_shrink[*cface_num*] ← *shrink_value*; *n* ← *cfont_base* + 1;
  while *n* < *cfont_ptr* **do**
    begin if *cface_num* = *cfont_face*[*n*] **then** *set_ceglue_spec*(*n*);
    incr(*n*);
    end;
  end;
end

This code is used in section 1550.

1555. ⟨ Modify the depth factor of the specified chinese face 1555 ⟩ ≡
begin *cur_val* ← *convfix*(*cur_val*);
if *cface_fw_depth*[*cface_num*] ≠ *cur_val* **then**
begin *cface_fw_depth*[*cface_num*] ← *cur_val*; *n* ← *cfont_base* + 1;
while *n* < *cfont_ptr* **do**
begin if *cface_num* = *cfont_face*[*n*] **then**
cfont_depth[*n*] ← *fw_times_sd*(*cface_fw_depth*[*cface_num*], *cfont_size*[*n*]);
incr(*n*);
end;
end;
end

This code is used in section 1550.

1556.

define *pux_set_cfont_csp* = 0
define *pux_set_cfont_cesp* = 1

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("PUXcfontcspace", *pux_set_cfont_attrib*, *pux_set_cfont_csp*);
primitive("PUXcfontcespace", *pux_set_cfont_attrib*, *pux_set_cfont_cesp*);

1557. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
pux_set_cfont_attrib: **begin case** *chr_code* **of**
pux_set_cfont_csp: *print_esc*("PUXcfontcspace");
pux_set_cfont_cesp: *print_esc*("PUXcfontcespace");
endcases;
end;

1558. ⟨ Assignments 1220 ⟩ +≡
pux_set_cfont_attrib: **begin** *p* ← *cur_chr*; ⟨ Get the next non-blank non-call token 409 ⟩;
if *cur_cmd* = *set_cfont* **then** { the first form }
begin *cfont_num* ← *cur_chr*;
end
else if *cur_cmd* = *set_font* ∧ *cur_chr* = *cur_font* **then** *cfont_num* ← *cur_cfont*
else begin *print_err*("Missing␣CJK␣font␣identifier");
help2("I␣was␣looking␣for␣a␣control␣sequence␣whose")
("current␣meaning␣is␣a␣CJK␣font␣command."); *back_error*; *cfont_num* ← *null_cfont*;
end;
scan_optional_equals;
case *p* **of**
pux_set_cfont_csp: **begin** *scan_glue*(*glue_val*); *width*(*cfont_glue_spec*[*cfont_num*]) ← *width*(*cur_val*);
shrink(*cfont_glue_spec*[*cfont_num*]) ← *shrink*(*cur_val*);
stretch(*cfont_glue_spec*[*cfont_num*]) ← *stretch*(*cur_val*); *fast_delete_glue_ref*(*cur_val*);
end;
pux_set_cfont_cesp: **begin** *scan_glue*(*glue_val*); *width*(*cfont_ceglue_spec*[*cfont_num*]) ← *width*(*cur_val*);
shrink(*cfont_ceglue_spec*[*cfont_num*]) ← *shrink*(*cur_val*);
stretch(*cfont_ceglue_spec*[*cfont_num*]) ← *stretch*(*cur_val*); *fast_delete_glue_ref*(*cur_val*);
end;
endcases;
end;

1559. ⟨ Global variables 13 ⟩ +≡
g_cspace_width: *integer*;
g_cspace_shrink: *integer*;
g_cspace_stretch: *integer*;
g_cespace_width: *integer*;
g_cespace_shrink: *integer*;
g_cespace_stretch: *integer*;

1560.

define *default_csp_width* = 50
define *default_cesp_width* = 150

⟨ Set initial values of key variables 21 ⟩ +≡
g_cspace_width ← *default_csp_width*; *g_cspace_shrink* ← *g_cspace_width* **div** 3; *g_cspace_stretch* ← 125;
g_cespace_width ← *default_cesp_width*; *g_cespace_shrink* ← *g_cespace_width* **div** 3;
g_cespace_stretch ← *g_cespace_width* **div** 2;

1561. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("puxgCspace", *puxg_assign_space*, *puxg_set_cspace*);
primitive("puxgCEspace", *puxg_assign_space*, *puxg_set_cespace*);

1562. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
puxg_assign_space: **begin if** *chr_code* = *puxg_set_cspace* **then** *print_esc*("puxgCspace")
else if *chr_code* = *puxg_set_cespace* **then** *print_esc*("puxgCEspace");
end;

1563. ⟨ Assignments 1220 ⟩ +≡
puxg_assign_space: **begin** *p* ← *cur_chr*; ⟨ Scan spacing dimension of CJK font face 1552 ⟩;
if *p* = *puxg_set_cspace* **then**
begin *g_cspace_width* ← *width_value*; *g_cspace_stretch* ← *stretch_value*;
g_cspace_shrink ← *shrink_value*; *n* ← *cface_base*;
while *n* < *cface_ptr* **do**
begin *cface_csp_width*[*n*] ← *width_value*; *cface_csp_shrink*[*n*] ← *shrink_value*;
cface_csp_stretch[*n*] ← *stretch_value*; *incr*(*n*);
end;
n ← *cfont_base* + 1;
while *n* < *cfont_ptr* **do**
begin *set_cglue_spec*(*n*); *incr*(*n*);
end;
end
else if *p* = *puxg_set_cespace* **then**
begin *g_cespace_width* ← *width_value*; *g_cespace_stretch* ← *stretch_value*;
g_cespace_shrink ← *shrink_value*;
end;
n ← *cface_base*;
while *n* < *cface_ptr* **do**
begin *cface_cesp_width*[*n*] ← *width_value*; *cface_cesp_shrink*[*n*] ← *shrink_value*;
cface_cesp_stretch[*n*] ← *stretch_value*; *incr*(*n*);
end;
n ← *cfont_base* + 1;
while *n* < *cfont_ptr* **do**
begin *set_ceglue_spec*(*n*); *incr*(*n*);
end;
end;

1564. Dump Font Info.

⟨ Other variables used by the procedure *prefixed_command* 1415 ⟩ +≡
old_setting: 0 . . *max_selector*; { holds *selector* setting }

1565. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("PUXdumpfontinfo", *pux_dump_font_info*, 0);

1566. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
pux_dump_font_info: *print_esc*("PUXdumpfontinfo"); { TCW }

1567. ⟨ Assignments 1220 ⟩ +≡
pux_dump_font_info: **begin** *old_setting* ← *selector*; *selector* ← *log_only*;
⟨ Print TeX fonts 1568 ⟩;
⟨ Print CJK font faces 1569 ⟩;
⟨ Print CJK fonts 1570 ⟩;
⟨ Print font faces matching table 1571 ⟩;
selector ← *old_setting*;
end;

1568. ⟨ Print TeX fonts 1568 ⟩ ≡
print_ln; *print*("Tex␣fonts"); *print_ln*; *n* ← 0;
while *n* ≤ *font_ptr* **do**
begin *print_int*(*n*); *print*(":␣"); *print*(*font_name*[*n*]);
print("␣dsize=␣"); *print_scaled*(*font_dsize*[*n*]); *print*("pt");
print("␣at␣"); *print_scaled*(*font_size*[*n*]); *print*("pt");
print("␣matched␣CJK␣font="); *print_int*(*font_matching_table*(*n*)); *print_ln*; *incr*(*n*);
end

This code is used in section 1567.

1569. ⟨ Print CJK font faces 1569 ⟩ ≡
print("Chinese␣faces"); *print_ln*; *n* ← 0;
while *n* < *cface_ptr* **do**
begin *print_int*(*n*); *print*(":␣"); *print*("id="); *print*(*cface*[*n*]);
print("␣name="); *print*(*cface_name*[*n*]);
print("␣charset="); *print_int*(*cface_charset*[*n*]);
print("␣weight="); *print_int*(*cface_weight*[*n*]);
print("␣style="); *print_int*(*cface_style*[*n*]);
print("␣w="); *print_fixword*(*cface_fw_width*[*n*]);
print("␣h="); *print_fixword*(*cface_fw_height*[*n*]);
print("␣d="); *print_fixword*(*cface_fw_depth*[*n*]);
print_ln; *incr*(*n*);
end

This code is used in section 1567.

1570. ⟨Print CJK fonts 1570⟩ ≡

```
print("CJK␣fonts"); print_ln; n ← cfont_base;
while n < cfont_ptr do
  begin print_int(n); print(":face=␣"); print(cface[cfont_face[n]]);
  print("␣dsize=␣"); print_scaled(cfont_dsize[n]); print("pt");
  print("␣at␣"); print_scaled(cfont_size[n]); print("pt");
  print_ln; incr(n);
  end
```

This code is used in section 1567.

1571. ⟨Print font faces matching table 1571⟩ ≡

```
print("English/CJK␣font␣faces␣matching␣table"); print_ln; n ← min_ectbl;
while n < ectbl_ptr do
  begin print_int(n); print(":␣"); print("eface="); print(ectbl_eface_name[n]);
  print("␣cface_id="); print(cface[ectbl_cface_num(n)]);
  print("␣cface_num="); print_int(ectbl_cface_num(n));
  print_ln; incr(n);
  end
```

This code is used in section 1567.

1572. ⟨Global variables 13⟩ +≡

```
dvi_cf: internal_cfont_number;  { the current chinese font }
```

1573. ⟨Output the CJK font definitions for all fonts that were used 1573⟩ ≡

```
while cfont_ptr > cfont_base do
  begin if cfont_used[cfont_ptr] then dvi_cfont_def(cfont_ptr);
  decr(cfont_ptr);
  end
```

This code is used in section 645.

1574. ⟨Change font *dvi_cf* to *f* 1574⟩ ≡

```
begin if ¬cfont_used[f] then
  begin dvi_cfont_def(f); cfont_used[f] ← true;
  end;
dvi_out(cfnt); dvi_out((f − cfont_base − 1) div 256); dvi_out((f − cfont_base − 1) mod 256); dvi_cf ← f;
end
```

This code is used in section 623.

1575. Dump/undump PUTEX internal information.

1576. ⟨ Dump the CJK font face information 1576 ⟩ ≡
dump_int(*cface_ptr*); *dump_int*(*cface_fw_default_depth*);
for $k \leftarrow$ *cface_base* **to** *cface_ptr* -1 **do**
begin *dump_int*(*cface*[*k*]); *dump_int*(*cface_name*[*k*]); *dump_int*(*cface_charset*[*k*]);
dump_int(*cface_weight*[*k*]); *dump_int*(*cface_style*[*k*]); *dump_int*(*cface_fw_width*[*k*]);
dump_int(*cface_fw_height*[*k*]); *dump_int*(*cface_fw_depth*[*k*]); *dump_int*(*cface_csp_width*[*k*]);
dump_int(*cface_csp_shrink*[*k*]); *dump_int*(*cface_csp_stretch*[*k*]); *dump_int*(*cface_cesp_width*[*k*]);
dump_int(*cface_cesp_shrink*[*k*]); *dump_int*(*cface_cesp_stretch*[*k*]); *print_ln*; *print_int*(*k*); *print*(":␣");
print("id="); *print*(*cface*[*k*]);
print("␣name="); *print*(*cface_name*[*k*]);
print("␣charset="); *print_int*(*cface_charset*[*k*]);
print("␣weight="); *print_int*(*cface_weight*[*k*]);
print("␣style="); *print_int*(*cface_style*[*k*]);
print("␣w="); *print_fixword*(*cface_fw_width*[*k*]);
print("␣h="); *print_fixword*(*cface_fw_height*[*k*]);
print("␣d="); *print_fixword*(*cface_fw_depth*[*k*]);
end;
print_ln; *print_int*(*cface_ptr* − *cface_base*); *print*("␣preloaded␣CJK␣font␣face");
if *cface_ptr* $\neq$ *cface_base* $+ 1$ **then** *print_char*("s")

This code is used in section 1305.

1577. ⟨ Undump the CJK font face information 1577 ⟩ ≡
undump_size(*cface_base*)(*max_cface*)(´cface␣max´)(*cface_ptr*); *undump_int*(*cface_fw_default_depth*);
for $k \leftarrow$ *cface_base* **to** *cface_ptr* -1 **do**
begin *undump_size*(0)(*pool_size*)(´cface␣id´)(*cface*[*k*]);
undump_size(0)(*pool_size*)(´cface␣name´)(*cface_name*[*k*]);
undump_size(0)(255)(´charset␣size´)(*cface_charset*[*k*]);
undump_size(1)(1000)(´cface␣weight´)(*cface_weight*[*k*]);
undump_size(0)(255)(´cface␣style´)(*cface_style*[*k*]); *undump_int*(*cface_fw_width*[*k*]);
undump_int(*cface_fw_height*[*k*]); *undump_int*(*cface_fw_depth*[*k*]); *undump_int*(*cface_csp_width*[*k*]);
undump_int(*cface_csp_shrink*[*k*]); *undump_int*(*cface_csp_stretch*[*k*]); *undump_int*(*cface_cesp_width*[*k*]);
undump_int(*cface_cesp_shrink*[*k*]); *undump_int*(*cface_cesp_stretch*[*k*]);
end

This code is used in section 1306.

1578. ⟨ Dump the face matching table 1578 ⟩ ≡
dump_int(*ectbl_ptr*);
for $k \leftarrow$ *min_ectbl* **to** *ectbl_ptr* -1 **do** *dump_int*(*ectbl_eface_name*[*k*])

This code is used in section 1305.

1579. ⟨ Unump the face matching table 1579 ⟩ ≡
undump_size(*min_ectbl*)(*max_ectbl*)(´ectbl_ptr´)(*ectbl_ptr*);
for $k \leftarrow$ *min_ectbl* **to** *ectbl_ptr* -1 **do**
undump_size(0)(*pool_size*)(´ectbl␣eface␣name´)(*ectbl_eface_name*[*k*])

This code is used in section 1306.

1580. ⟨Dump the CJK font information 1580⟩ ≡
begin *dump_int*(*cfont_ptr*);
for $k \leftarrow$ *default_cfont* **to** *cfont_ptr* $- 1$ **do**
begin *dump_int*(*cfont_face*[k]); *dump_int*(*cfont_dsize*[k]); *dump_int*(*cfont_size*[k]);
dump_int(*cfont_width*[k]); *dump_int*(*cfont_height*[k]); *dump_int*(*cfont_depth*[k]);
dump_int(*cfont_glue_spec*[k]); *dump_int*(*cfont_ceglue_spec*[k]); *print_ln*; *print_int*(k);
print(":face=␣"); *print*(*cface*[*cfont_face*[k]]);
print("␣dsize=␣"); *print_scaled*(*cfont_dsize*[k]); *print*("pt");
print("␣at␣"); *print_scaled*(*cfont_size*[k]); *print*("pt");
end;
end

This code is used in section 1305.

1581. ⟨Undump the CJK font information 1581⟩ ≡
begin *undump_size*(*cfont_base*)(*cfont_max*)('cfont␣max')(*cfont_ptr*);
for $k \leftarrow$ *default_cfont* **to** *cfont_ptr* $- 1$ **do**
begin *undump_size*(*cface_base*)(*max_cface*)('cface␣max')(*cfont_face*[k]); *undump_int*(*cfont_dsize*[k]);
undump_int(*cfont_size*[k]); *undump_int*(*cfont_width*[k]); *undump_int*(*cfont_height*[k]);
undump_int(*cfont_depth*[k]); *undump_int*(*cfont_glue_spec*[k]); *undump_int*(*cfont_ceglue_spec*[k]);
end;
end

This code is used in section 1306.

1582. 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, 537
`*`: 174, 176, 178, 313, 360, 859, 1009, 1358
`->`: 294
¡system dependencies: 1382
`=>`: 363
`???`: 59
`?`: 83
`@`: 859
`@@`: 849
a: 47, 102, 218, 521, 522, 526, 563, 694, 725, 741, 755, 1126, 1197, 1214, 1239, 1260, 1441, 1482, 1526, 1535
`A <box> was supposed to...`: 1087
a_close: 51, 329, 488, 489, 1278, 1336, 1377, 1381
a_leaders: 149, 189, 628, 630, 637, 639, 659, 674, 1074, 1075, 1076, 1081, 1151
a_make_name_string: 528, 537, 540
a_open_in: 51, 540, 1278
a_open_out: 537, 1377
A_token: 448
abort: 563, 566, 567, 568, 571, 572, 573, 574, 576, 578
above: 208, 1049, 1181, 1182, 1183
`\above` primitive: 1181
above_code: 1181, 1182, 1185, 1186
above_display_short_skip: 224, 817
`\abovedisplayshortskip` primitive: 226
above_display_short_skip_code: 224, 225, 226, 1206
above_display_skip: 224, 817
`\abovedisplayskip` primitive: 226
above_display_skip_code: 224, 225, 226, 1206, 1209
`\abovewithdelims` primitive: 1181
abs: 66, 186, 211, 218, 219, 421, 425, 451, 504, 613, 666, 678, 721, 740, 760, 761, 762, 834, 839, 852, 862, 947, 951, 1032, 1033, 1059, 1079, 1081, 1083, 1086, 1096, 1113, 1123, 1130, 1152, 1246, 1247, 1380
absorbing: 305, 306, 339, 476
acc_kern: 155, 191, 1128
accent: 208, 265, 266, 1093, 1125, 1167, 1168
`\accent` primitive: 265
accent_c: 1399, 1400, 1401, 1402
accent_chr: 690, 699, 741, 1168
accent_height: 1399, 1402
accent_noad: 690, 693, 699, 701, 736, 764, 1168, 1189
accent_noad_size: 690, 701, 764, 1168
accent_slant: 1399, 1402
accent_width: 1399, 1402
`\PUXacnumber` primitive: 471
acnumber_code: 471, 472, 474, 475
act_width: 869, 870, 871, 872, 874
action procedure: 1032
active: 162, 822, 832, 846, 857, 863, 864, 866, 867, 868, 876, 877, 878
active_base: 220, 222, 252, 255, 262, 263, 353, 445, 509, 1155, 1260, 1292, 1318, 1320, 1482
active_char: 207, 344, 509
active_height: 973, 978, 979
active_node_size: 822, 848, 863, 867, 868
active_width: 826, 827, 832, 846, 864, 867, 869, 871, 973
actual_looseness: 875, 876, 878
add_delims_to: 347
add_glue_ref: 203, 206, 433, 805, 884, 999, 1103, 1232
add_token_ref: 203, 206, 323, 982, 1015, 1019, 1224, 1230, 1360
additional: 647, 648, 660, 675
addressof: 1335, 1373
adj_demerits: 236, 839, 862
`\adjdemerits` primitive: 238
adj_demerits_code: 236, 237, 238
adjust: 579
adjust_head: 162, 891, 892, 1079, 1088, 1202, 1208
adjust_node: 142, 148, 175, 183, 202, 206, 650, 654, 658, 733, 764, 869, 902, 1103
adjust_ptr: 142, 197, 202, 206, 658, 1103
adjust_space_factor: 1037, 1460
adjust_tail: 650, 651, 652, 654, 658, 799, 891, 892, 1079, 1088, 1202
adjusted_hbox_group: 269, 1065, 1086, 1088
adv_past: 1365, 1366
advance: 209, 265, 266, 1213, 1238, 1239, 1241
`\advance` primitive: 265
advance_major_tail: 917, 920

after: 147, 869, 1199
after_assignment: 208, 265, 266, 1271
`\afterassignment` primitive: 265
after_group: 208, 265, 266, 1274
`\aftergroup` primitive: 265
after_math: 1196, 1197
after_token: 1269, 1270, 1271, 1272
aire: 563, 564, 566, 579
align_error: 1129, 1130
align_group: 269, 771, 777, 794, 803, 1134, 1135
align_head: 162, 773, 780
align_peek: 776, 777, 788, 802, 1051, 1136
align_ptr: 773, 774, 775
align_stack_node_size: 773, 775
align_state: 88, 309, 324, 325, 331, 339, 342, 347, 357, 397, 398, 399, 406, 445, 478, 485, 486, 489, 773, 774, 775, 777, 780, 786, 787, 788, 791, 792, 794, 1072, 1097, 1129, 1130
aligning: 305, 306, 339, 780, 792
alignment of rules with characters: 592
alpha: 563, 574, 575, 1441
alpha_file: 25, 50, 54, 304, 483, 528, 1335, 1345
alpha_token: 441, 443
alter_aux: 1245, 1246
alter_box_dimen: 1245, 1250
alter_integer: 1245, 1249
alter_page_so_far: 1245, 1248
alter_prev_graf: 1245, 1247
`Ambiguous...`: 1186
Amble, Ole: 928
`AmSTeX`: 1334
any_mode: 1048, 1051, 1060, 1066, 1070, 1076, 1100, 1105, 1107, 1129, 1137, 1213, 1271, 1274, 1277, 1279, 1288, 1293, 1350
any_state_plus: 344, 345, 347
app_lc_hex: 48
app_space: 1046, 1463, 1467
append_char: 42, 48, 52, 58, 180, 195, 260, 519, 528, 695, 698, 942, 1373, 1418, 1510, 1525
append_charnode_to_t: 911, 914
append_choices: 1174, 1175
append_discretionary: 1119, 1120
append_glue: 1060, 1063, 1081
append_italic_correction: 1115, 1116
append_kern: 1060, 1064
append_normal_space: 1033, 1461, 1463, 1464, 1467
append_penalty: 1105, 1106
append_src_special: 1037, 1406
append_to_name: 522, 526
append_to_vlist: 682, 802, 891, 1079, 1206, 1207, 1208
append_wchar: 42, 1418
arabic_wchar_offset: 1421
area_delimiter: 516, 518, 519, 520, 528
`Argument of \x has...`: 398
arith_error: 104, 105, 106, 107, 451, 456, 463, 1239
`Arithmetic overflow`: 1239
artificial_demerits: 833, 854, 857, 858, 859
ASCII code: 17, 506
ASCII_code: 18, 19, 20, 29, 30, 31, 38, 42, 51, 54, 58, 60, 82, 292, 341, 392, 519, 522, 526, 695, 895, 915, 924, 946, 953, 956, 962, 963, 1306, 1335, 1379, 1409
assign_dimen: 209, 248, 249, 416, 1213, 1227, 1231
assign_font_dimen: 209, 265, 266, 416, 1213, 1256
assign_font_int: 209, 416, 1213, 1256, 1257, 1258
assign_glue: 209, 226, 227, 416, 785, 1213, 1227, 1231
assign_int: 209, 238, 239, 416, 1213, 1225, 1227, 1231, 1240
assign_mu_glue: 209, 226, 227, 416, 1213, 1225, 1227, 1231, 1240
assign_toks: 209, 230, 231, 233, 323, 416, 418, 1213, 1227, 1229, 1230
`at`: 1261
`\atop` primitive: 1181
atop_code: 1181, 1182, 1185
`\atopwithdelims` primitive: 1181
attach_fraction: 451, 456, 457, 459
attach_sign: 451, 452, 458
auto_breaking: 865, 866, 869, 871
aux: 212, 213, 216, 803, 815
aux_field: 212, 213, 218, 778
aux_save: 803, 815, 1209
avail: 118, 120, 121, 122, 123, 164, 168, 1314, 1315
`AVAIL list clobbered...`: 168
awful_bad: 836, 837, 838, 839, 857, 877, 973, 977, 978, 990, 1008, 1009, 1010
axis_height: 703, 709, 739, 749, 750, 752, 765
b: 467, 468, 473, 501, 526, 563, 682, 708, 709, 712, 714, 718, 833, 973, 997, 1201, 1250, 1291, 1441
b_close: 563, 645
b_make_name_string: 528, 535
b_open_in: 566
b_open_out: 535
back_error: 327, 376, 399, 406, 418, 445, 449, 479, 482, 506, 580, 786, 1081, 1087, 1164, 1200, 1210, 1215, 1493, 1535, 1558
back_input: 281, 325, 326, 327, 371, 372, 375, 378, 382, 398, 408, 410, 418, 446, 447, 451, 455, 458, 464, 529, 791, 1034, 1050, 1057, 1067, 1093, 1098, 1127, 1130, 1135, 1141, 1153, 1155, 1156, 1218, 1224, 1229, 1272, 1378, 1462, 1487

back_list: 323, 325, 337, 410, 1291
backed_up: 307, 311, 312, 314, 323, 324, 325, 1029
background: 826, 827, 830, 840, 866, 867
backup_backup: 369
backup_head: 162, 369, 410
`BAD`: 293, 294
bad: 13, 14, 111, 290, 525, 1252, 1335
`Bad \patterns`: 964
`Bad \prevgraf`: 1247
`Bad character code`: 437
`Bad delimiter code`: 440
`Bad flag...`: 170
`Bad link...`: 182
`Bad mathchar`: 439
`Bad number`: 438, 1388
`Bad register code`: 436
`Bad space factor`: 1246
`Bad wide character code`: 1419
bad_fmt: 1306, 1309, 1311, 1315, 1320, 1328, 1330, 1404
bad_pool: 51, 52, 53
bad_tfm: 563
badness: 108, 663, 670, 677, 681, 831, 855, 856, 978, 1010
`\badness` primitive: 419
badness_code: 419, 427
banner: 2, 61, 539, 1302
banner_k: 2, 61, 539
base_c: 1396, 1397, 1399, 1400, 1401, 1402
base_height: 1399, 1402
base_line: 622, 626, 627, 631, 1402
base_ptr: 84, 85, 310, 311, 312, 313, 1134
base_slant: 1399, 1402
base_width: 1399, 1402
base_x_height: 1399, 1402
baseline_skip: 224, 247, 682
`\baselineskip` primitive: 226
baseline_skip_code: 149, 224, 225, 226, 682
batch_mode: 73, 75, 86, 90, 92, 93, 538, 1265, 1266, 1268, 1330, 1331, 1336
`\batchmode` primitive: 1265
bc: 543, 544, 546, 548, 563, 568, 569, 573, 579, 1415, 1416
bch_label: 563, 576, 579
bchar: 563, 576, 579, 904, 906, 908, 909, 911, 914, 916, 919, 920, 1035, 1037, 1040, 1043, 1460
bchar_label: 552, 579, 912, 919, 1037, 1043, 1325, 1326, 1340
before: 147, 192, 1199
begin: 7, 8
begin_box: 1076, 1082, 1087
begin_diagnostic: 76, 245, 284, 299, 323, 403, 404, 505, 512, 584, 641, 644, 666, 678, 866, 990, 995, 1009, 1014, 1124, 1227, 1296, 1299, 1396, 1400, 1401
begin_file_reading: 78, 87, 328, 486, 540
begin_group: 208, 265, 266, 1066
`\begingroup` primitive: 265
begin_insert_or_adjust: 1100, 1102
begin_name: 515, 518, 528, 529, 530, 534
begin_pseudoprint: 316, 318, 319
begin_token_list: 323, 359, 389, 393, 777, 791, 792, 802, 1028, 1033, 1086, 1094, 1142, 1148, 1170, 1374
`Beginning to dump...`: 1331
below_display_short_skip: 224
`\belowdisplayshortskip` primitive: 226
below_display_short_skip_code: 224, 225, 226, 1206
below_display_skip: 224
`\belowdisplayskip` primitive: 226
below_display_skip_code: 224, 225, 226, 1206, 1209
best_bet: 875, 877, 878, 880, 881
best_height_plus_depth: 974, 977, 1013, 1014
best_ins_ptr: 984, 1008, 1012, 1021, 1023, 1024
best_line: 875, 877, 878, 880, 893
best_page_break: 983, 1008, 1016, 1017
best_pl_line: 836, 848, 858
best_place: 836, 848, 858, 973, 977, 983
best_size: 983, 1008, 1020
beta: 563, 574, 575, 1441
big_op_spacing1: 704, 754
big_op_spacing2: 704, 754
big_op_spacing3: 704, 754
big_op_spacing4: 704, 754
big_op_spacing5: 704, 754
big_switch: 209, 236, 997, 1032, 1033, 1034, 1039, 1044, 1463, 1467
BigEndian order: 543
billion: 628
bin_noad: 685, 693, 699, 701, 731, 732, 764, 1159, 1160
bin_op_penalty: 236, 764
`\binoppenalty` primitive: 238
bin_op_penalty_code: 236, 237, 238
blank_line: 245
boolean: 32, 37, 45, 46, 47, 76, 79, 96, 104, 106, 107, 165, 167, 245, 256, 311, 361, 410, 416, 443, 451, 464, 476, 501, 519, 520, 521, 527, 528, 530, 552, 563, 581, 595, 622, 632, 648, 709, 722, 729, 794, 828, 831, 832, 833, 865, 880, 903, 910, 946, 953, 963, 992, 1015, 1035, 1054, 1057, 1094, 1163, 1197, 1214, 1284, 1306, 1340, 1345, 1373,

1382, 1386, 1393, 1394, 1396, 1409, 1420, 1422, 1423, 1424, 1448, 1486, 1502, 1526
bop: 586, 588, 589, 591, 593, 595, 641, 643
Bosshard, Hans Rudolf: 461
bot: 549
bot_mark: 385, 386, 1015, 1019
\botmark primitive: 387
bot_mark_code: 385, 387, 388
bottom_level: 269, 272, 281, 1067, 1071
bottom_line: 311
bound_default: 32, 1335
bound_name: 32, 1335
boundary_normal: 207
bowels: 595
box: 230, 232, 423, 508, 980, 995, 996, 1012, 1018, 1020, 1021, 1024, 1026, 1031, 1082, 1113, 1250, 1299
\box primitive: 1074
box_base: 230, 232, 233, 255, 1080
box_code: 1074, 1075, 1082, 1110, 1113
box_context: 1078, 1079, 1080, 1081, 1082, 1086, 1087
box_end: 1078, 1082, 1087, 1089
box_error: 995, 996, 1018, 1031
box_flag: 1074, 1078, 1080, 1086, 1244
box_max_depth: 247, 1089
\boxmaxdepth primitive: 248
box_max_depth_code: 247, 248
box_node_size: 135, 136, 202, 206, 652, 671, 718, 730, 754, 759, 980, 1024, 1103, 1113, 1204
box_ref: 210, 232, 275, 1080
box_there: 983, 990, 1003, 1004
\box255 is not void: 1018
bp: 461
brain: 1032
breadth_max: 181, 182, 198, 233, 236, 1342
break_node: 822, 848, 858, 859, 867, 880, 881
break_penalty: 208, 265, 266, 1105
break_type: 832, 840, 848, 849, 862
break_width: 826, 827, 840, 841, 843, 844, 845, 846, 847, 882
breakpoint: 1341
broken_ins: 984, 989, 1013, 1024
broken_penalty: 236, 893
\brokenpenalty primitive: 238
broken_penalty_code: 236, 237, 238
broken_ptr: 984, 1013, 1024
buf_size: 30, 31, 32, 35, 71, 111, 315, 328, 331, 341, 363, 369, 377, 527, 533, 537, 1335, 1337, 1524
buffer: 30, 31, 36, 37, 45, 71, 83, 87, 88, 259, 260, 261, 264, 302, 303, 315, 318, 331, 341, 343, 352, 354, 355, 360, 362, 363, 369, 377, 486, 487, 526, 527, 533, 534, 537, 541, 1335, 1340, 1342, 1524
build_choices: 1176, 1177
build_discretionary: 1121, 1122
build_page: 803, 815, 991, 997, 1029, 1057, 1063, 1079, 1094, 1097, 1103, 1106, 1148, 1203
by: 1239
bypass_eoln: 31
byte_file: 25, 528, 535, 542
b0: 110, 114, 133, 221, 268, 548, 549, 553, 557, 559, 567, 605, 686, 688, 1312, 1313, 1342
b1: 110, 114, 133, 221, 268, 548, 549, 557, 559, 567, 605, 686, 688, 1312, 1313, 1342
b2: 110, 114, 548, 549, 557, 559, 567, 605, 686, 688, 1312, 1313, 1342
b3: 110, 114, 548, 549, 559, 567, 605, 686, 688, 1312, 1313, 1342
c: 47, 63, 82, 144, 264, 274, 292, 341, 473, 519, 522, 526, 563, 584, 585, 590, 595, 648, 695, 697, 709, 712, 714, 715, 741, 752, 896, 915, 956, 962, 963, 997, 1015, 1089, 1113, 1120, 1139, 1154, 1158, 1184, 1246, 1248, 1249, 1250, 1278, 1282, 1291, 1338, 1385, 1396, 1397, 1441
C_arabic_digit_offset: 1421, 1435
C_formal_digit_offset: 1421
c_leaders: 149, 190, 630, 639, 1074, 1075
\cleaders primitive: 1074
c_loc: 915, 919
C_simple_digit_offset: 1421
call: 210, 223, 275, 296, 366, 369, 383, 390, 398, 399, 510, 1221, 1224, 1228, 1229, 1230, 1298
call_edit: 84, 1336
cancel_boundary: 1033, 1035, 1036, 1037
cannot \read: 487
car_ret: 207, 232, 342, 347, 780, 783, 784, 786, 787, 788, 791, 1129
carriage_return: 22, 49, 207, 232, 240, 363
case_shift: 208, 1288, 1289, 1290
cat: 341, 354, 355, 356, 1413
cat_code: 230, 232, 236, 341, 355, 1340, 1411, 1413
\catcode primitive: 1233
cat_code_base: 230, 232, 233, 235, 1233, 1234, 1235, 1236, 1416
cc: 341, 352, 355
cc: 461
cf: 588
cface: 475, 1474, 1480, 1481, 1484, 1497, 1515, 1569, 1570, 1571, 1576, 1577, 1580
cface_base: 1472, 1473, 1480, 1481, 1485, 1545, 1547, 1563, 1576, 1577, 1581
cface_cesp_shrink: 1474, 1479, 1506, 1554, 1563, 1576, 1577

cface_cesp_stretch: 1474, 1479, 1506, 1554, 1563, 1576, 1577
cface_cesp_width: 1474, 1479, 1506, 1554, 1563, 1576, 1577
cface_charset: 605, 1474, 1480, 1497, 1569, 1576, 1577
cface_csp_shrink: 1474, 1478, 1505, 1553, 1563, 1576, 1577
cface_csp_stretch: 1474, 1478, 1505, 1553, 1563, 1576, 1577
cface_csp_width: 1474, 1478, 1505, 1553, 1563, 1576, 1577
cface_found: 1481
cface_fw_default_depth: 1474, 1480, 1547, 1576, 1577
cface_fw_depth: 605, 1474, 1480, 1496, 1497, 1514, 1547, 1555, 1569, 1576, 1577
cface_fw_height: 605, 1474, 1480, 1496, 1497, 1514, 1569, 1576, 1577
cface_fw_width: 605, 1474, 1480, 1496, 1497, 1514, 1569, 1576, 1577
cface_id: 1526, 1540
cface_id_base: 222, 1474, 1482
cface_id_text: 1474, 1482
cface_name: 605, 1474, 1480, 1485, 1497, 1569, 1576, 1577
cface_num: 1505, 1506, 1508, 1509, 1510, 1512, 1513, 1526, 1527, 1528, 1529, 1530, 1535, 1536, 1537, 1538, 1541, 1550, 1553, 1554, 1555
cface_ptr: 1474, 1480, 1481, 1484, 1485, 1497, 1545, 1547, 1563, 1569, 1576, 1577
cface_style: 605, 1474, 1480, 1496, 1497, 1545, 1569, 1576, 1577
cface_weight: 605, 1474, 1480, 1496, 1497, 1569, 1576, 1577
cfn: 1514
cfnt: 588, 589, 1574
cfnt_def: 588, 589, 605
cfont_base: 232, 605, 1499, 1500, 1503, 1512, 1547, 1553, 1554, 1555, 1563, 1570, 1573, 1574, 1581
cfont_ceglue_spec: 1451, 1452, 1453, 1455, 1457, 1463, 1464, 1467, 1502, 1506, 1507, 1514, 1558, 1580, 1581
cfont_depth: 605, 657, 1502, 1503, 1514, 1547, 1555, 1580, 1581
cfont_dsize: 475, 605, 1502, 1503, 1514, 1570, 1580, 1581
cfont_face: 475, 605, 1502, 1503, 1505, 1506, 1512, 1514, 1515, 1537, 1541, 1547, 1553, 1554, 1555, 1570, 1580, 1581
cfont_glue_spec: 1453, 1455, 1457, 1459, 1463, 1464, 1467, 1502, 1505, 1507, 1514, 1558, 1580, 1581
cfont_height: 605, 657, 1502, 1503, 1514, 1580, 1581
cfont_id_base: 222
cfont_in_short_display: 173, 174, 666, 867, 1342
cfont_max: 174, 176, 1499, 1500, 1503, 1514, 1581
cfont_max_limit: 222, 1499
cfont_num: 1526, 1527, 1535, 1536, 1537, 1538, 1558
cfont_ptr: 1502, 1504, 1512, 1514, 1547, 1553, 1554, 1555, 1563, 1570, 1573, 1580, 1581
cfont_size: 475, 605, 1502, 1503, 1505, 1506, 1512, 1514, 1515, 1547, 1555, 1570, 1580, 1581
cfont_used: 1502, 1503, 1573, 1574
cfont_width: 605, 623, 657, 718, 844, 845, 869, 870, 873, 874, 1150, 1502, 1503, 1514, 1580, 1581
cglue_ptr: 1448, 1450
cglue_spec: 1448, 1450
ch: 1413
change_if_limit: 500, 501, 512
char: 19, 1326, 1384
`\char` primitive: 265
char_base: 553, 557, 569, 573, 579, 1325, 1326, 1340
char_box: 712, 713, 714, 741
`\chardef` primitive: 1225
char_def_code: 1225, 1226, 1227
char_depth: 557, 657, 711, 712, 715
char_depth_end: 557
char_exists: 557, 576, 579, 585, 623, 711, 725, 741, 743, 752, 758, 1039, 1396, 1397, 1400
char_given: 208, 416, 938, 1033, 1093, 1127, 1154, 1157, 1225, 1226, 1227, 1453, 1459, 1460, 1463, 1464
char_height: 557, 657, 711, 712, 715, 1128, 1402
char_height_end: 557
char_info: 546, 553, 557, 558, 560, 585, 623, 657, 712, 715, 717, 718, 727, 741, 844, 845, 869, 870, 873, 874, 912, 1040, 1042, 1043, 1116, 1126, 1128, 1150, 1396, 1397, 1400
char_info_end: 557
char_info_word: 544, 546, 547
char_italic: 557, 712, 717, 752, 758, 1116
char_italic_end: 557
char_kern: 560, 744, 756, 912, 1043
char_kern_end: 560
char_list_accent: 557, 1400
char_list_char: 557, 1396, 1397, 1400
char_list_exists: 557, 1396, 1397, 1400
char_node: 134, 143, 145, 162, 176, 551, 595, 623, 652, 755, 884, 910, 1032, 1116, 1141

char_num: 208, 265, 266, 938, 1033, 1093, 1127, 1154, 1157, 1453, 1460, 1463, 1464
char_sub_code: 230, 557, 585, 1400
char_sub_code_base: 230, 1227
\charsubdef primitive: 1225
char_sub_def_code: 1225, 1226, 1227
char_sub_def_max: 236, 240, 1227, 1396, 1397, 1400
\charsubdefmax primitive: 238
char_sub_def_max_code: 236, 237, 238, 1227
char_sub_def_min: 236, 240, 1227, 1396, 1397, 1400
\charsubdefmin primitive: 238
char_sub_def_min_code: 236, 237, 238, 1227
char_tag: 557, 573, 711, 713, 743, 744, 752, 755, 912, 1042
char_val_flag: 417, 468, 1409
char_warning: 584, 585, 725, 1039
char_width: 557, 623, 657, 712, 717, 718, 743, 844, 845, 869, 870, 873, 874, 1126, 1128, 1150, 1402
char_width_end: 557
character: 134, 143, 144, 174, 176, 206, 585, 623, 657, 684, 685, 686, 690, 694, 712, 718, 725, 727, 752, 755, 756, 844, 845, 869, 870, 873, 874, 899, 900, 901, 906, 910, 911, 913, 914, 1035, 1037, 1038, 1039, 1040, 1043, 1116, 1126, 1128, 1150, 1154, 1158, 1168, 1453, 1457, 1460
character set dependencies: 23, 49
charset: 1482, 1483, 1488, 1497
check sum: 53, 545, 591
check_byte_range: 573, 576
check_cfont: 1512, 1513, 1527, 1538
check_dimensions: 729, 730, 736, 757
check_existence: 576, 577
check_full_save_stack: 273, 274, 276, 280
check_interrupt: 96, 324, 343, 756, 914, 1034, 1043
check_mem: 165, 167, 1034, 1342
check_outer_validity: 336, 351, 353, 354, 357, 362, 378
check_quoted: 521
check_shrinkage: 828, 830, 871
Chinese characters: 134, 588
choice_node: 691, 692, 693, 701, 733
choose_mlist: 734
chr: 19, 20, 23, 24, 1225
chr_cmd: 298, 784
chr_code: 227, 231, 239, 249, 298, 380, 388, 414, 415, 416, 420, 472, 491, 495, 784, 987, 1056, 1062, 1074, 1075, 1092, 1111, 1118, 1146, 1160, 1173, 1182, 1192, 1212, 1223, 1226, 1234, 1254, 1258, 1264, 1266, 1276, 1281, 1290, 1292, 1295, 1349, 1426, 1446, 1466, 1515, 1543, 1549, 1557, 1562
ci: 1397
cinttype: 32, 1382, 1384
CJK Fonts Extension: 174, 176, 209
CJK_digit_offset: 1421
\PUXnameseq primitive: 471
\PUXcjknumber primitive: 471
cjknumber_code: 471, 472, 474, 475
clang: 212, 213, 815, 1037, 1094, 1203, 1379, 1380
clean_box: 723, 737, 738, 740, 741, 745, 747, 752, 753, 760, 761, 762
clear_for_error_prompt: 78, 83, 330, 346
clear_terminal: 34, 330, 533
clear_trie: 961
CLOBBERED: 293
clobbered: 167, 168, 169, 1373
close_files_and_terminate: 78, 81, 1335, 1336
\closein primitive: 1275
close_noad: 685, 693, 699, 701, 731, 764, 765, 1159, 1160
close_node: 1344, 1347, 1349, 1351, 1359, 1360, 1361, 1376, 1377, 1378
\closeout primitive: 1347
closed: 483, 484, 486, 488, 489, 504, 1278
clr: 740, 746, 748, 749, 759, 760, 761, 762
club_penalty: 236, 893
\clubpenalty primitive: 238
club_penalty_code: 236, 237, 238
cm: 461
cmd: 298, 1225, 1292
cnum_one_flag: 1420, 1422, 1423, 1424
\PUXcnumber primitive: 471
cnumber_code: 471, 472, 474, 475
co_backup: 369
combine_two_deltas: 863
comment: 207, 232, 347
common_ending: 15, 501, 503, 512, 652, 663, 669, 670, 671, 677, 680, 681, 898, 906, 1260, 1263, 1296, 1297, 1300, 1482, 1495, 1496, 1509, 1513
Completed box...: 641
compress_trie: 952, 955
cond_math_glue: 149, 189, 735, 1174
cond_ptr: 492, 493, 498, 499, 500, 501, 503, 512, 1338
conditional: 369, 370, 501
confusion: 95, 202, 206, 281, 500, 633, 672, 731, 739, 757, 764, 769, 794, 801, 803, 844, 845, 869, 873, 874, 880, 971, 976, 1003, 1071, 1188, 1203, 1214, 1351, 1360, 1361, 1376
const_chk: 1335
const_cstring: 32, 537
conststringcast: 1373
continental_point_token: 441, 451

continue: 15, 82, 83, 84, 88, 89, 392, 395, 396, 397, 398, 400, 622, 623, 709, 711, 777, 787, 818, 832, 835, 854, 899, 909, 912, 913, 914, 997, 1004, 1400
contrib_head: 162, 215, 218, 991, 997, 998, 1001, 1002, 1004, 1020, 1026, 1029, 1311
contrib_tail: 998, 1020, 1026, 1029
contribute: 997, 1000, 1003, 1005, 1011, 1367
conv_toks: 369, 370, 473
conventions for representing stacks: 300
convert: 210, 366, 370, 471, 472, 473
convert_to_break_width: 846
convfix: 1439, 1480, 1496, 1547, 1555
\copy primitive: 1074
copy_code: 1074, 1075, 1082, 1110, 1111, 1113
copy_node_list: 161, 203, 204, 206, 1082, 1113
copy_to_cur_active: 832, 864
count: 236, 430, 641, 643, 989, 1011, 1012, 1013
\count primitive: 414
count_base: 236, 239, 242, 1227, 1240
\countdef primitive: 1225
count_def_code: 1225, 1226, 1227
\cr primitive: 783
cr_code: 783, 784, 792, 794, 795
\crcr primitive: 783
cr_cr_code: 783, 788, 792
cramped: 691, 705
cramped_style: 705, 737, 740, 741
cs_count: 256, 258, 260, 1321, 1322, 1337
cs_error: 1137, 1138
cs_name: 210, 265, 266, 366, 370
\csname primitive: 265
cs_token_flag: 289, 290, 293, 334, 336, 337, 339, 357, 358, 365, 372, 375, 378, 382, 383, 384, 445, 469, 509, 783, 1068, 1135, 1218, 1292, 1317, 1374, 1406
cstring: 523
cur_active_width: 826, 827, 832, 835, 840, 846, 847, 854, 855, 856, 863
cur_align: 773, 774, 775, 780, 781, 782, 786, 789, 791, 792, 794, 795, 798, 799, 801
cur_area: 515, 520, 528, 532, 533, 1260, 1263, 1354, 1377
cur_boundary: 270, 271, 272, 274, 282
cur_box: 1077, 1078, 1079, 1080, 1081, 1082, 1083, 1084, 1085, 1087, 1089, 1090
cur_break: 824, 848, 882, 883, 884
cur_c: 725, 726, 727, 741, 752, 755, 756, 758
cur_cface: 230, 1480, 1537
cur_cface_loc: 230, 234, 1480, 1541
cur_cfont: 230, 1037, 1453, 1463, 1464, 1467, 1501, 1541, 1558
cur_cfont_loc: 230, 234, 1501, 1527, 1537, 1539
cur_chr: 88, 296, 297, 299, 332, 337, 341, 343, 348, 349, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 364, 365, 381, 383, 384, 389, 390, 392, 406, 410, 416, 427, 431, 445, 473, 475, 477, 479, 482, 486, 497, 498, 501, 503, 509, 510, 511, 512, 513, 529, 580, 785, 788, 792, 938, 940, 965, 1033, 1037, 1039, 1052, 1061, 1063, 1064, 1069, 1076, 1082, 1086, 1093, 1096, 1108, 1109, 1113, 1120, 1127, 1131, 1143, 1145, 1154, 1155, 1157, 1158, 1161, 1162, 1163, 1174, 1184, 1194, 1214, 1215, 1216, 1220, 1221, 1224, 1227, 1229, 1230, 1231, 1235, 1236, 1237, 1240, 1246, 1248, 1249, 1250, 1255, 1256, 1268, 1278, 1282, 1291, 1296, 1338, 1351, 1353, 1378, 1416, 1418, 1445, 1451, 1452, 1453, 1457, 1458, 1459, 1460, 1463, 1464, 1467, 1477, 1487, 1493, 1526, 1528, 1535, 1537, 1538, 1539, 1541, 1544, 1546, 1550, 1558, 1563
cur_cmd: 88, 211, 296, 297, 299, 332, 337, 341, 342, 343, 344, 348, 349, 351, 353, 354, 357, 358, 360, 364, 365, 369, 370, 371, 375, 383, 384, 389, 390, 406, 407, 409, 410, 416, 418, 431, 443, 445, 446, 447, 451, 455, 458, 464, 466, 477, 479, 480, 481, 482, 486, 497, 509, 510, 529, 580, 780, 785, 786, 787, 788, 791, 792, 938, 964, 1032, 1033, 1052, 1069, 1081, 1082, 1087, 1098, 1102, 1127, 1131, 1141, 1154, 1155, 1163, 1168, 1179, 1180, 1200, 1209, 1214, 1215, 1216, 1224, 1229, 1230, 1231, 1239, 1240, 1255, 1273, 1378, 1416, 1418, 1452, 1453, 1459, 1460, 1462, 1463, 1464, 1477, 1487, 1493, 1526, 1528, 1535, 1550, 1558
cur_cs: 297, 332, 333, 336, 337, 338, 341, 351, 353, 354, 356, 357, 358, 365, 375, 377, 382, 383, 384, 392, 394, 410, 475, 476, 510, 777, 1155, 1218, 1221, 1224, 1227, 1228, 1229, 1260, 1297, 1355, 1374, 1482
cur_ext: 515, 520, 528, 532, 533, 1354, 1377
cur_f: 725, 727, 741, 744, 752, 755, 756, 758
cur_fam: 236, 1154, 1158, 1168
cur_fam_code: 236, 237, 238, 1142, 1148
cur_file: 304, 329, 362, 540, 541
cur_font: 230, 232, 561, 562, 580, 1035, 1037, 1045, 1047, 1120, 1126, 1127, 1149, 1526, 1527, 1535, 1558
cur_font_loc: 230, 232, 233, 234, 1220
cur_g: 622, 628, 632, 637
cur_glue: 622, 628, 632, 637
cur_group: 270, 271, 272, 274, 281, 282, 803, 1065, 1066, 1067, 1068, 1070, 1071, 1072, 1133, 1134, 1143, 1145, 1194, 1195, 1196, 1197, 1203
cur_h: 619, 620, 621, 622, 623, 625, 626, 629, 630, 631, 632, 635, 640, 1402

cur_head: 773, 774, 775, 789, 802
cur_height: 973, 975, 976, 977, 978, 979
cur_i: 725, 726, 727, 741, 744, 752, 755, 756, 758
cur_if: 336, 492, 493, 498, 499, 1338
cur_indent: 880, 892
cur_input: 35, 36, 87, 301, 302, 311, 321, 322, 537, 1134
cur_l: 910, 911, 912, 913, 914, 1035, 1037, 1038, 1039, 1040, 1042, 1043
cur_lang: 894, 895, 926, 927, 933, 937, 942, 947, 966, 1094, 1203, 1365
cur_length: 41, 180, 182, 260, 519, 528, 620, 695, 1371, 1373
cur_level: 270, 271, 272, 274, 277, 278, 280, 281, 1307, 1338
cur_line: 880, 892, 893
cur_list: 213, 216, 217, 218, 425, 1247
cur_loop: 773, 774, 775, 780, 786, 795, 796, 797
cur_mark: 296, 385, 389, 1338
cur_mlist: 722, 723, 729, 757, 1197, 1199, 1202
cur_mu: 706, 722, 733, 735, 769
cur_name: 515, 520, 528, 532, 533, 540, 1260, 1261, 1263, 1354, 1377
cur_order: 369, 442, 450, 451, 457, 465
cur_p: 826, 831, 832, 833, 836, 840, 842, 843, 848, 854, 856, 858, 859, 860, 861, 862, 863, 865, 866, 868, 869, 870, 871, 872, 875, 880, 881, 882, 883, 884, 897, 906, 1365
cur_q: 910, 911, 913, 914, 1037, 1038, 1039, 1040, 1043
cur_r: 910, 911, 912, 913, 914, 1035, 1037, 1040, 1042, 1043, 1460
cur_rh: 909, 911, 912, 913
cur_s: 596, 601, 602, 619, 622, 632, 643, 645
cur_size: 703, 704, 706, 722, 725, 726, 735, 739, 740, 747, 749, 750, 751, 752, 760, 761, 762, 765
cur_span: 773, 774, 775, 790, 799, 801
cur_style: 706, 722, 723, 729, 733, 734, 737, 738, 740, 741, 745, 747, 748, 749, 751, 752, 753, 757, 759, 760, 761, 762, 763, 766, 769, 1197, 1199, 1202
cur_tail: 773, 774, 775, 789, 799, 802
cur_tok: 88, 281, 297, 325, 326, 327, 336, 364, 365, 369, 371, 372, 375, 378, 382, 383, 384, 395, 396, 397, 398, 400, 402, 406, 408, 410, 443, 444, 445, 447, 448, 451, 455, 477, 479, 480, 482, 486, 497, 506, 509, 786, 787, 1050, 1098, 1130, 1131, 1135, 1218, 1224, 1271, 1272, 1274, 1374, 1375, 1453, 1460
cur_v: 619, 621, 622, 626, 627, 631, 632, 634, 635, 636, 638, 639, 640, 643, 1402
cur_val: 264, 265, 334, 369, 413, 416, 417, 418, 422, 423, 424, 426, 427, 428, 429, 430, 432, 433, 434, 436, 437, 438, 439, 440, 441, 442, 443, 445, 447, 448, 450, 451, 453, 454, 456, 458, 460, 461, 463, 464, 465, 466, 468, 469, 475, 485, 494, 504, 506, 507, 508, 512, 556, 580, 581, 582, 583, 648, 783, 785, 938, 1033, 1063, 1064, 1076, 1082, 1085, 1102, 1106, 1113, 1126, 1127, 1154, 1157, 1163, 1164, 1168, 1185, 1191, 1227, 1228, 1229, 1230, 1231, 1232, 1235, 1237, 1239, 1240, 1241, 1242, 1243, 1244, 1246, 1247, 1248, 1249, 1250, 1251, 1256, 1261, 1262, 1278, 1299, 1347, 1353, 1380, 1388, 1416, 1419, 1428, 1430, 1432, 1433, 1435, 1437, 1438, 1445, 1452, 1453, 1460, 1463, 1464, 1488, 1489, 1490, 1491, 1492, 1544, 1545, 1546, 1547, 1552, 1555, 1558
cur_val_level: 369, 413, 416, 422, 423, 424, 426, 427, 430, 432, 433, 442, 452, 454, 458, 464, 468, 469
cur_width: 880, 892
current page: 983
current_character_being_worked_on: 573
cv_backup: 369
cvl_backup: 369
d: 107, 176, 177, 259, 341, 443, 563, 652, 671, 682, 709, 833, 947, 973, 1071, 1089, 1141, 1201, 1373, 1441, 1482
d_fixed: 611, 612
danger: 1197, 1198, 1202
data: 210, 232, 1220, 1227, 1235, 1237, 1416, 1480, 1501, 1519, 1527, 1529, 1532, 1535, 1537, 1539, 1541
data structure assumptions: 161, 164, 204, 819, 971, 984, 1292
date_and_time: 241
day: 236, 241, 539, 620, 1331
\day primitive: 238
day_code: 236, 237, 238
db_char: 377, 1409
dd: 461
deactivate: 832, 854, 857
dead_cycles: 422, 595, 596, 641, 1015, 1027, 1028, 1057, 1245, 1249
\deadcycles primitive: 419
debug: 7, 9, 78, 84, 93, 114, 165, 166, 167, 172, 1034, 1341
debug #: 1341
debug_format_file: 1309, 1322, 1386
debug_help: 78, 84, 93, 1341
debugging: 7, 84, 96, 114, 165, 182, 1034, 1341
decent_fit: 820, 837, 855, 856, 867
decr: 42, 44, 64, 71, 86, 88, 89, 90, 92, 102, 120, 121, 123, 175, 177, 200, 201, 205, 217, 245, 260,

281, 282, 311, 322, 324, 325, 329, 331, 347, 356, 357, 360, 362, 369, 397, 402, 425, 432, 445, 480, 486, 497, 512, 520, 537, 541, 571, 579, 604, 622, 632, 641, 645, 646, 719, 720, 806, 811, 843, 861, 872, 886, 918, 919, 933, 934, 943, 944, 947, 951, 968, 1063, 1103, 1123, 1130, 1134, 1177, 1189, 1197, 1247, 1296, 1314, 1338, 1340, 1387, 1391, 1451, 1455, 1524, 1525, 1573
def: 209, 1211, 1212, 1213, 1216, 1221
\def primitive: 1211
def_cfont: 580
def_code: 209, 416, 1213, 1233, 1234, 1235
def_family: 209, 416, 580, 1213, 1233, 1234, 1237
def_font: 209, 265, 266, 416, 580, 1213, 1259
def_ref: 305, 306, 476, 485, 963, 1104, 1221, 1229, 1282, 1291, 1355, 1357, 1373, 1406
default_cesp: 236
default_cesp_width: 1560
default_cface_fw_height: 1474
default_cface_fw_width: 1474
default_cface_style: 1474
default_cface_weight: 1474
default_cfont: 232, 1501, 1504, 1580, 1581
default_code: 686, 700, 746, 1185
default_csp: 236
default_csp_width: 1560
default_depth: 236, 240
default_hyphen_char: 236, 579
\defaulthyphenchar primitive: 238
default_hyphen_char_code: 236, 237, 238
default_rule: 466
default_rule_thickness: 686, 704, 737, 738, 740, 746, 748, 762
default_skew_char: 236, 579
\defaultskewchar primitive: 238
default_skew_char_code: 236, 237, 238
defecation: 600
define: 1217, 1220, 1221, 1224, 1227, 1228, 1229, 1230, 1231, 1235, 1237, 1239, 1251, 1260, 1416, 1482, 1509, 1524, 1527, 1529, 1535, 1537, 1539, 1541
defined_cfont: 1512, 1513
defining: 305, 306, 339, 476, 485
del_code: 236, 240, 1163
\delcode primitive: 1233
del_code_base: 236, 240, 242, 1233, 1235, 1236
delete_glue_ref: 201, 202, 275, 454, 468, 581, 735, 805, 819, 829, 884, 979, 999, 1007, 1020, 1025, 1103, 1232, 1239, 1242, 1338
delete_last: 1107, 1108
delete_q: 729, 763, 766
delete_token_ref: 200, 202, 275, 324, 980, 982, 1015, 1019, 1338, 1361
deletions_allowed: 76, 77, 84, 85, 98, 336, 346
delim_num: 207, 265, 266, 1049, 1154, 1157, 1163
delimited_code: 1181, 1182, 1185, 1186
delimiter: 690, 699, 765, 1194
\delimiter primitive: 265
delimiter_factor: 236, 765
\delimiterfactor primitive: 238
delimiter_factor_code: 236, 237, 238
delimiter_shortfall: 247, 765
\delimitershortfall primitive: 248
delimiter_shortfall_code: 247, 248
delim1: 703, 751
delim2: 703, 751
delta: 103, 729, 731, 736, 738, 739, 740, 741, 745, 746, 748, 749, 750, 751, 752, 753, 757, 758, 759, 762, 765, 997, 1011, 1013, 1126, 1128, 1399, 1402, 1440
delta_node: 825, 833, 835, 846, 847, 863, 864, 868, 877, 878
delta_node_size: 825, 846, 847, 863, 864, 868
delta1: 746, 749, 765
delta2: 746, 749, 765
den: 588, 590, 593
denom: 453, 461
denom_style: 705, 747
denominator: 686, 693, 700, 701, 747, 1184, 1188
denom1: 703, 747
denom2: 703, 747
deplorable: 977, 1008
depth: 466
depth: 135, 136, 138, 139, 140, 184, 187, 188, 466, 557, 625, 627, 629, 634, 635, 638, 644, 652, 656, 659, 671, 673, 682, 691, 707, 709, 712, 716, 730, 733, 734, 738, 739, 740, 748, 749, 750, 752, 753, 754, 759, 761, 762, 771, 772, 804, 809, 813, 976, 1005, 1012, 1013, 1024, 1090, 1103
depth_base: 553, 557, 569, 574, 1325, 1326, 1340
depth_index: 546, 557
depth_offset: 135, 419, 772, 1250
depth_threshold: 181, 182, 198, 233, 236, 695, 1342
dig: 54, 64, 65, 67, 102, 455, 1524
digit_base: 473, 1422, 1423, 1424, 1433, 1434, 1436
digit_sensed: 963, 964, 965
dimen: 247, 430, 1011, 1013
\dimen primitive: 414
dimen_base: 220, 236, 247, 248, 249, 250, 251, 252, 1073, 1148
\dimendef primitive: 1225
dimen_def_code: 1225, 1226, 1227
dimen_par: 247

dimen_pars: 247
dimen_val: 413, 414, 415, 416, 418, 419, 420, 421, 423, 424, 427, 428, 430, 431, 432, 452, 458, 468, 1240
Dimension too large: 463
dirty Pascal: 3, 114, 172, 182, 186, 285, 815, 1334
disc_break: 880, 883, 884, 885, 893
disc_group: 269, 1120, 1121, 1122
disc_node: 145, 148, 175, 183, 202, 206, 733, 764, 820, 822, 832, 859, 861, 869, 884, 917, 1084, 1108
disc_width: 842, 843, 872, 873
discretionary: 208, 1093, 1117, 1118, 1119
Discretionary list is too long: 1123
\discretionary primitive: 1117
Display math...with $$: 1200
display_indent: 247, 803, 1141, 1148, 1202
\displayindent primitive: 248
display_indent_code: 247, 248, 1148
\displaylimits primitive: 1159
display_mlist: 692, 698, 701, 734, 1177
display_style: 691, 697, 734, 1172, 1202
\displaystyle primitive: 1172
display_widow_penalty: 236, 1148
\displaywidowpenalty primitive: 238
display_widow_penalty_code: 236, 237, 238
display_width: 247, 1141, 1148, 1202
\displaywidth primitive: 248
display_width_code: 247, 248, 1148
div: 100, 630, 639
divide: 209, 265, 266, 1213, 1238, 1239
\divide primitive: 265
do_all_six: 826, 832, 835, 840, 846, 847, 863, 864, 867, 973, 990
do_assignments: 803, 1126, 1209, 1273
do_endv: 1133, 1134
do_extension: 1350, 1351, 1378
do_final_end: 81, 1335
do_nothing: 16, 34, 57, 58, 84, 175, 202, 275, 344, 357, 541, 572, 612, 614, 615, 625, 634, 654, 672, 695, 731, 736, 764, 840, 869, 902, 1048, 1239, 1362, 1363, 1376
do_register_command: 1238, 1239
doc_charset: 620, 645, 1409
doing_leaders: 595, 596, 631, 640, 1377
done: 15, 47, 53, 202, 281, 282, 311, 383, 392, 400, 443, 448, 451, 456, 461, 476, 477, 479, 485, 486, 497, 529, 533, 534, 540, 563, 570, 579, 618, 641, 643, 644, 701, 729, 741, 743, 763, 764, 777, 780, 818, 832, 840, 866, 876, 880, 884, 898, 909, 912, 914, 934, 963, 964, 973, 977, 980, 982, 997, 1000, 1001, 1008, 1082, 1084, 1122, 1124, 1141, 1149, 1214, 1230, 1255, 1361, 1481, 1482, 1512, 1523, 1535
done_with_noad: 729, 730, 731, 736, 757
done_with_node: 729, 730, 733, 734, 757
done1: 15, 167, 168, 392, 402, 451, 455, 476, 477, 741, 744, 777, 786, 818, 832, 855, 880, 882, 897, 899, 902, 963, 968, 997, 1000, 1003, 1305, 1318, 1423, 1482, 1485, 1526
done2: 15, 167, 169, 451, 461, 462, 476, 481, 777, 787, 818, 899, 1305, 1319, 1526
done3: 15, 818, 900, 901
done4: 15, 818, 902
done5: 15, 818, 869, 872
done6: 15
dont_expand: 210, 258, 357, 372
Double subscript: 1180
Double superscript: 1180
double_hyphen_demerits: 236, 862
\doublehyphendemerits primitive: 238
double_hyphen_demerits_code: 236, 237, 238
Doubly free location...: 169
down_ptr: 608, 609, 610, 618
downdate_width: 863
down1: 588, 589, 610, 612, 613, 616, 617, 619
down2: 588, 597, 613
down3: 588, 613
down4: 588, 613
\dp primitive: 419
dry rot: 95
ds: 588, 1508, 1511
dsize: 473, 475, 1508, 1509, 1511, 1514
dummy_xchr: 1306, 1390
dummy_xord: 1306, 1390
dummy_xprn: 1306, 1390
\dump...only by INITEX: 1338
\dump primitive: 1055
dump_core: 1341
dump_four_ASCII: 1312
dump_hh: 1321
dump_int: 1310, 1312, 1314, 1316, 1318, 1319, 1321, 1323, 1327, 1329, 1403, 1576, 1578, 1580
dump_line: 32, 1340
dump_option: 32
dump_qqqq: 1312
dump_things: 1310, 1312, 1314, 1318, 1319, 1321, 1323, 1325, 1327, 1389
Duplicate pattern: 966
dvi length exceeds...: 601, 602, 643
dvi_buf: 597, 598, 600, 601, 610, 616, 617, 1335
dvi_buf_size: 14, 32, 597, 598, 599, 601, 602, 610, 616, 617, 643, 645, 1335
dvi_cf: 620, 623, 1572, 1574

dvi_cfont_def: 605, 1573, 1574
dvi_f: 619, 620, 623, 624
dvi_file: 535, 595, 598, 600, 601, 645
DVI files: 586
dvi_font_def: 605, 624, 646
dvi_four: 603, 605, 613, 620, 627, 636, 643, 645, 1371
dvi_gone: 597, 598, 599, 601, 615, 643
dvi_h: 619, 620, 622, 623, 626, 627, 631, 632, 635, 640, 1402
dvi_index: 597
dvi_limit: 597, 598, 599, 601, 602, 643
dvi_offset: 597, 598, 599, 601, 602, 604, 608, 610, 616, 617, 622, 632, 643, 645
dvi_out: 601, 603, 604, 605, 606, 612, 613, 620, 622, 623, 624, 627, 632, 636, 643, 645, 1371, 1402, 1574
dvi_pop: 604, 622, 632
dvi_ptr: 597, 598, 599, 601, 602, 604, 610, 622, 632, 643, 645
dvi_swap: 601
dvi_v: 619, 620, 622, 626, 631, 632, 635, 640
dyn_used: 117, 120, 121, 122, 123, 164, 642, 1314, 1315
e: 277, 279, 521, 522, 533, 1201, 1214
easy_line: 822, 838, 850, 851, 853
ec: 543, 544, 546, 548, 563, 568, 569, 573, 579, 585, 1415, 1416
ectbl_cface_num: 230, 1519, 1530, 1571
ectbl_cface_num_base: 230, 1519, 1529
ectbl_eface_name: 1518, 1523, 1526, 1529, 1571, 1578, 1579
ectbl_found: 1523, 1526, 1530
ectbl_ptr: 1518, 1519, 1523, 1526, 1529, 1571, 1578, 1579
\edef primitive: 1211
edge: 622, 626, 629, 632, 638
edit_file: 84
edit_line: 84, 1336, 1382
edit_name_length: 84, 1336, 1382
edit_name_start: 84, 1336, 1382, 1383
eface_name: 1523, 1526, 1529, 1530
effective_char: 557, 585, 1039, 1396, 1397
effective_char_info: 1039, 1397
efname: 1526
efont_name: 1525, 1526, 1530
efont_num: 1535
eight_bit_p: 24, 32, 1390
eight_bits: 25, 64, 112, 297, 552, 563, 584, 585, 598, 610, 652, 709, 712, 715, 980, 995, 996, 1082, 1250, 1326, 1335, 1340, 1409, 1441, 1474
eightbits: 1396, 1397
eject_penalty: 157, 832, 834, 854, 862, 876, 973, 975, 977, 1008, 1013, 1014
else: 10
\else primitive: 494
else_code: 492, 494, 501
em: 458
Emergency stop: 93
emergency_stretch: 247, 831, 866
\emergencystretch primitive: 248
emergency_stretch_code: 247, 248
empty: 16, 215, 424, 684, 688, 690, 695, 725, 726, 741, 752, 754, 755, 757, 758, 759, 983, 989, 990, 994, 1004, 1011, 1179, 1180, 1189
empty line at end of file: 489, 541
empty_field: 687, 688, 689, 745, 1166, 1168, 1184
empty_flag: 124, 126, 130, 150, 164, 1315
end: 7, 8, 10
End of file on the terminal: 37, 71
(\end occurred...): 1338
\end primitive: 1055
end_cs_name: 208, 265, 266, 375, 1137
\endcsname primitive: 265
end_diagnostic: 245, 284, 299, 323, 403, 404, 505, 512, 584, 641, 644, 666, 678, 866, 990, 995, 1009, 1014, 1124, 1227, 1301, 1396, 1400, 1401
end_file_reading: 329, 330, 360, 362, 486, 540, 1338
end_graf: 1029, 1088, 1097, 1099, 1103, 1134, 1136, 1171
end_group: 208, 265, 266, 1066
\endgroup primitive: 265
\endinput primitive: 379
end_line_char: 87, 236, 240, 303, 318, 332, 360, 362, 486, 537, 541, 1340
\endlinechar primitive: 238
end_line_char_code: 236, 237, 238
end_line_char_inactive: 360, 362, 486, 541, 1340
end_match: 207, 289, 291, 294, 394, 395, 397
end_match_token: 289, 392, 394, 395, 396, 397, 477, 479, 485
end_name: 515, 520, 528, 529, 534
end_of_TEX: 81
end_span: 162, 771, 782, 796, 800, 804, 806
end_template: 210, 366, 369, 378, 383, 783, 1298
end_template_token: 783, 787, 793
end_token_list: 324, 325, 357, 393, 1029, 1338, 1374
end_write: 222, 1372, 1374
\endwrite: 1372
end_write_token: 1374, 1375
endcases: 10
endif: 7, 8, 643, 645
endifn: 645

endv: 207, 298, 378, 383, 771, 783, 785, 794, 1049, 1133, 1134
engine_name: 11, 1310, 1311
ensure_dvi_open: 535, 620
ensure_vbox: 996, 1012, 1021
eof: 26, 31, 52, 567
eoln: 31, 52
eop: 586, 588, 589, 591, 643, 645
eq_define: 277, 278, 279, 375, 785, 1073, 1080, 1217
eq_destroy: 275, 277, 279, 283
eq_level: 221, 222, 228, 232, 236, 253, 264, 277, 279, 283, 783, 980, 1311, 1318, 1372, 1480, 1501, 1519, 1532
eq_level_field: 221
eq_no: 208, 1143, 1144, 1146, 1147
\eqno primitive: 1144
eq_save: 276, 277, 278
eq_type: 210, 221, 222, 223, 228, 232, 253, 258, 264, 265, 267, 277, 279, 351, 353, 354, 357, 358, 375, 392, 394, 783, 1155, 1311, 1318, 1372, 1480, 1501, 1519, 1532
eq_type_field: 221, 275
eq_word_define: 278, 279, 1073, 1142, 1148, 1217
eqtb: 115, 163, 220, 221, 222, 223, 224, 228, 230, 232, 236, 240, 242, 247, 250, 251, 252, 253, 255, 256, 262, 264, 265, 266, 267, 268, 270, 272, 274, 275, 276, 277, 278, 279, 281, 282, 283, 284, 285, 286, 289, 291, 297, 298, 305, 307, 332, 333, 354, 392, 416, 417, 476, 494, 551, 556, 783, 817, 1191, 1211, 1225, 1241, 1243, 1256, 1260, 1311, 1318, 1319, 1320, 1335, 1340, 1342, 1347, 1348, 1428, 1482, 1519, 1524, 1532, 1544, 1546
eqtb_size: 220, 247, 250, 252, 253, 254, 256, 260, 262, 283, 290, 1218, 1310, 1311, 1319, 1320, 1321, 1322, 1335
eqtb_top: 222, 252, 256, 262, 1218, 1311, 1335
equiv: 221, 222, 223, 224, 228, 229, 230, 232, 233, 234, 235, 253, 255, 264, 265, 267, 275, 277, 279, 351, 353, 354, 357, 358, 416, 417, 418, 511, 580, 783, 1155, 1230, 1242, 1243, 1260, 1292, 1311, 1318, 1372, 1482, 1519, 1532
equiv_field: 221, 275, 285
err: 1526, 1528, 1529
err_help: 79, 230, 1286, 1287
\errhelp primitive: 230
err_help_loc: 230
\errmessage primitive: 1280
err_p: 1396
error: 72, 75, 76, 78, 79, 82, 88, 91, 93, 98, 327, 338, 346, 373, 401, 411, 421, 431, 448, 457, 459, 462, 463, 478, 479, 489, 503, 513, 526, 538, 564, 570, 582, 644, 726, 779, 787, 795, 829, 939, 940, 963, 964, 965, 966, 979, 981, 995, 1007, 1012, 1027, 1030, 1053, 1067, 1069, 1071, 1072, 1083, 1085, 1098, 1102, 1109, 1113, 1123, 1124, 1131, 1132, 1138, 1154, 1157, 1162, 1169, 1180, 1186, 1195, 1198, 1216, 1228, 1235, 1239, 1240, 1244, 1255, 1262, 1286, 1287, 1296, 1375, 1416, 1427, 1437, 1444, 1445, 1461, 1477, 1484, 1485, 1488, 1489, 1490, 1491, 1497, 1510, 1511, 1514, 1526, 1528, 1529, 1544, 1546, 1547, 1550
error_context_lines: 236, 311
\errorcontextlines primitive: 238
error_context_lines_code: 236, 237, 238
error_count: 76, 77, 82, 86, 1099, 1296
error_line: 14, 32, 58, 306, 311, 315, 316, 317, 1335
error_message_issued: 76, 82, 95
error_stop_mode: 72, 73, 74, 82, 93, 98, 1265, 1286, 1296, 1297, 1300, 1330, 1338
\errorstopmode primitive: 1265
escape: 207, 232, 344, 1340
escape_char: 236, 240, 243
\escapechar primitive: 238
escape_char_code: 236, 237, 238
etc: 182
ETC: 292
every_cr: 230, 777, 802
\everycr primitive: 230
every_cr_loc: 230, 231
every_cr_text: 307, 314, 777, 802
every_display: 230, 1148
\everydisplay primitive: 230
every_display_loc: 230, 231
every_display_text: 307, 314, 1148
every_hbox: 230, 1086
\everyhbox primitive: 230
every_hbox_loc: 230, 231
every_hbox_text: 307, 314, 1086
every_job: 230, 1033
\everyjob primitive: 230
every_job_loc: 230, 231
every_job_text: 307, 314, 1033
every_math: 230, 1142
\everymath primitive: 230
every_math_loc: 230, 231
every_math_text: 307, 314, 1142
every_par: 230, 1094
\everypar primitive: 230
every_par_loc: 230, 231, 307, 1229
every_par_text: 307, 314, 1094
every_vbox: 230, 1086, 1170
\everyvbox primitive: 230
every_vbox_loc: 230, 231
every_vbox_text: 307, 314, 1086, 1170

ex: 458
ex_hyphen_penalty: 145, 236, 872
\exhyphenpenalty primitive: 238
ex_hyphen_penalty_code: 236, 237, 238
ex_space: 208, 265, 266, 1093, 1461
exactly: 647, 648, 718, 892, 980, 1020, 1065, 1204
exit: 15, 16, 37, 47, 58, 59, 69, 82, 125, 182, 292, 341, 392, 410, 464, 500, 501, 527, 585, 610, 618, 652, 671, 755, 794, 832, 898, 937, 947, 951, 980, 997, 1015, 1033, 1057, 1082, 1108, 1113, 1116, 1122, 1154, 1162, 1177, 1214, 1239, 1273, 1306, 1338, 1341, 1392, 1397, 1416, 1526
expand: 32, 358, 367, 369, 371, 374, 383, 384, 442, 470, 481, 501, 513, 785
expand_after: 210, 265, 266, 366, 370
\expandafter primitive: 265
expand_char: 354, 355, 1409, 1410
expand_depth: 32, 367, 369, 1335
expand_depth_count: 367, 368, 369
explicit: 155, 720, 840, 869, 871, 882, 1061, 1116
ext_bot: 549, 716, 717
ext_delimiter: 516, 518, 519, 520, 528
ext_mid: 549, 716, 717
ext_rep: 549, 716, 717
ext_tag: 547, 572, 711, 713
ext_top: 549, 716, 717
exten: 547
exten_base: 553, 569, 576, 577, 579, 716, 1325, 1326, 1340
extensible_recipe: 544, 549
extension: 208, 1347, 1349, 1350, 1378
extensions to TEX: 2, 146, 1343
Extra \else: 513
Extra \endcsname: 1138
Extra \fi: 513
Extra \or: 503, 513
Extra \right.: 1195
Extra }, or forgotten x: 1072
Extra alignment tab...: 795
Extra x: 1069
extra_info: 772, 791, 792, 794, 795
extra_mem_bot: 32, 1311, 1335
extra_mem_top: 32, 1311, 1335
extra_right_brace: 1071, 1072
extra_space: 550, 561, 1047
extra_space_code: 550, 561
eyes and mouth: 332
f: 144, 451, 528, 563, 580, 581, 584, 585, 595, 605, 652, 709, 712, 714, 715, 718, 719, 720, 741, 833, 865, 1071, 1116, 1126, 1141, 1214, 1260, 1396, 1397, 1481, 1482, 1512
fabs: 186
face_id: 1508, 1510
face_name: 1482, 1485, 1497
face_num: 1478, 1479
false: 31, 37, 45, 46, 47, 51, 76, 80, 88, 89, 98, 106, 107, 166, 167, 168, 169, 238, 264, 284, 299, 311, 323, 327, 331, 336, 346, 354, 361, 362, 365, 377, 403, 404, 410, 428, 443, 444, 448, 450, 451, 452, 458, 463, 464, 465, 468, 475, 488, 504, 505, 508, 510, 512, 515, 518, 519, 520, 521, 527, 528, 529, 531, 541, 566, 584, 585, 596, 709, 723, 725, 757, 777, 794, 829, 831, 840, 854, 857, 866, 884, 906, 909, 913, 914, 954, 957, 963, 964, 965, 966, 969, 990, 993, 1009, 1014, 1023, 1029, 1034, 1036, 1037, 1038, 1039, 1043, 1054, 1057, 1064, 1104, 1170, 1185, 1186, 1194, 1195, 1197, 1202, 1227, 1229, 1239, 1244, 1261, 1273, 1282, 1285, 1286, 1291, 1306, 1328, 1339, 1340, 1345, 1346, 1355, 1357, 1373, 1374, 1377, 1394, 1395, 1396, 1400, 1401, 1410, 1422, 1487, 1503, 1524, 1526
false_bchar: 1035, 1037, 1460
fam: 684, 685, 686, 690, 694, 725, 726, 755, 756, 1154, 1158, 1168
\fam primitive: 238
fam_fnt: 230, 703, 704, 710, 725, 1198
fam_in_range: 1154, 1158, 1168
fast_delete_glue_ref: 201, 202, 1558
fast_get_avail: 122, 374, 1037, 1453, 1460
fast_store_new_token: 374, 402, 467, 469
Fatal format file error: 1306
fatal_error: 71, 93, 324, 360, 487, 533, 538, 601, 602, 643, 785, 792, 794, 1134
fatal_error_stop: 76, 77, 82, 93, 1335
fbyte: 567, 571, 574, 578
\PUXfcnumber primitive: 471
fcnumber_code: 471, 472, 474, 475
fd: 588
feof: 578
Ferguson, Michael John: 2
fetch: 725, 727, 741, 744, 752, 755, 758
fetch_efont_face: 1525, 1526, 1530
fetch_next_tok: 1033, 1453
fewest_demerits: 875, 877, 878
fflush: 34
fget: 567, 568, 571, 574, 578
fh: 588
\fi primitive: 494
fi_code: 492, 494, 495, 497, 501, 503, 512, 513
fi_or_else: 210, 366, 370, 492, 494, 495, 497, 513
fil: 457
fil: 135, 150, 164, 177, 457, 653, 662, 668, 1204
fil_code: 1061, 1062, 1063
fil_glue: 162, 164, 1063

fil_neg_code: 1061, 1063
fil_neg_glue: 162, 164, 1063
`File ended while scanning...`: 338
`File ended within \read`: 489
file_line_error_style_p: 32, 61, 73, 539
file_name_size: 11, 26, 522, 525, 526, 528
file_offset: 54, 55, 57, 58, 62, 540, 641, 1283
file_opened: 563, 564, 566
fill: 135, 150, 164, 653, 662, 668, 1204
fill_code: 1061, 1062, 1063
fill_glue: 162, 164, 1057, 1063
filll: 135, 150, 177, 457, 653, 662, 668, 1204
fin_align: 776, 788, 803, 1134
fin_col: 776, 794, 1134
fin_mlist: 1177, 1187, 1189, 1194, 1197
fin_row: 776, 802, 1134
fin_rule: 622, 625, 629, 632, 634, 638
final_cleanup: 1335, 1338
final_end: 6, 35, 331, 1335, 1340
final_hyphen_demerits: 236, 862
`\finalhyphendemerits` primitive: 238
final_hyphen_demerits_code: 236, 237, 238
final_pass: 831, 857, 866, 876
final_widow_penalty: 817, 818, 879, 880, 893
find_cface_num: 1481, 1484, 1510
find_ec_num: 1523, 1526, 1530
find_font_dimen: 428, 581, 1045, 1256
fingers: 514
finite_shrink: 828, 829
fire_up: 1008, 1015
firm_up_the_line: 340, 362, 363, 541
first: 30, 31, 35, 36, 37, 71, 83, 87, 88, 328, 329, 331, 355, 360, 362, 363, 377, 486, 534, 541
first_child: 963, 966, 967
first_control_char: 341, 354, 356
first_count: 54, 315, 316, 317
first_fit: 956, 960, 969
first_indent: 850, 852, 892
first_mark: 385, 386, 1015, 1019
`\firstmark` primitive: 387
first_mark_code: 385, 387, 388
first_text_char: 19, 24
first_width: 850, 852, 853, 892
fit_class: 833, 839, 848, 849, 855, 856, 858, 862
fitness: 822, 848, 862, 867
fix_d: 1482, 1496, 1497
fix_date_and_time: 241, 1335, 1340
fix_h: 1482, 1496, 1497
fix_language: 1037, 1379
fix_w: 1482, 1496, 1497
fix_word: 544, 545, 550, 551, 574
fixword: 1439, 1440, 1441, 1474, 1482
float: 109, 114, 186, 628, 637, 812
float_constant: 109, 186, 622, 628, 632, 1126, 1128, 1402
float_cost: 140, 188, 1011, 1103
floating_penalty: 140, 236, 1071, 1103
`\floatingpenalty` primitive: 238
floating_penalty_code: 236, 237, 238
flush_char: 42, 180, 195, 695, 698
flush_dvi: 643
flush_list: 123, 200, 324, 375, 399, 410, 804, 906, 944, 963, 1282, 1300, 1373
flush_math: 721, 779, 1198
flush_node_list: 199, 202, 275, 642, 701, 721, 734, 735, 745, 803, 819, 882, 886, 906, 921, 971, 995, 1002, 1081, 1108, 1123, 1124, 1378
flush_string: 44, 264, 520, 540, 944, 1282, 1331, 1392, 1484, 1485, 1510, 1526, 1530
flushable_string: 1260
fmem_ptr: 428, 552, 569, 572, 573, 579, 581, 582, 583, 1323, 1324, 1326, 1337, 1340
fmemory_word: 552, 1324, 1335
fmt_file: 527, 1308, 1331, 1332, 1340
fnt_def1: 588, 589, 605
fnt_def2: 588
fnt_def3: 588
fnt_def4: 588
fnt_num_0: 588, 589, 624
fnt1: 588, 589, 624
fnt2: 588
fnt3: 588
fnt4: 588
font: 134, 143, 144, 174, 176, 193, 206, 267, 551, 585, 623, 657, 684, 712, 718, 727, 844, 845, 869, 870, 873, 874, 899, 900, 901, 906, 911, 914, 1037, 1116, 1150, 1453, 1460, 1508
font metric files: 542
font parameters: 703, 704
`Font x has only...`: 582
`Font x=xx not loadable...`: 564
`Font x=xx not loaded...`: 570
`\font` primitive: 265
font_area: 552, 579, 605, 606, 1263, 1325, 1326, 1340
font_base: 11, 32, 111, 134, 222, 230, 232, 605, 624, 646, 1263, 1323, 1324, 1337, 1340, 1535
font_bc: 552, 557, 579, 585, 623, 711, 725, 1039, 1325, 1326, 1340, 1396, 1397, 1400
font_bchar: 552, 579, 900, 901, 918, 1035, 1037, 1325, 1326, 1340
font_check: 552, 571, 605, 1325, 1326, 1340
`\fontdimen` primitive: 265

font_dsize: 475, 552, 571, 605, 1263, 1264, 1325, 1326, 1340, 1527, 1538, 1568
font_ec: 552, 579, 585, 623, 711, 725, 1039, 1325, 1326, 1340, 1396, 1397, 1400
font_false_bchar: 552, 579, 1035, 1037, 1325, 1326, 1340
font_glue: 552, 579, 581, 1045, 1325, 1326, 1340
font_id_base: 222, 234, 256, 418, 551, 1260, 1524
font_id_text: 234, 256, 267, 582, 1260, 1325, 1524
font_in_short_display: 173, 174, 193, 666, 867, 1342
font_index: 551, 552, 563, 909, 1035, 1214, 1326, 1340
font_info: 32, 428, 551, 552, 553, 557, 560, 561, 563, 569, 572, 574, 576, 577, 578, 581, 583, 703, 704, 716, 744, 755, 912, 1035, 1042, 1045, 1214, 1256, 1311, 1323, 1324, 1335, 1340, 1342, 1396, 1397
font_k: 32, 1340
font_matching_table: 230, 1537, 1568
font_matching_table_base: 230, 1519, 1532, 1535
font_max: 12, 32, 111, 174, 429, 475, 569, 1326, 1335, 1337, 1340
font_max_limit: 222, 230, 1499
font_mem_size: 32, 569, 583, 1324, 1335, 1337
font_name: 475, 552, 579, 584, 605, 606, 1263, 1264, 1325, 1326, 1340, 1396, 1400, 1401, 1526, 1537, 1568
\fontname primitive: 471
font_name_code: 471, 472, 474, 475
font_params: 552, 579, 581, 582, 583, 1198, 1325, 1326, 1340
font_ptr: 552, 569, 579, 581, 646, 1263, 1323, 1324, 1325, 1326, 1337, 1340, 1568
font_size: 475, 552, 571, 605, 1263, 1264, 1325, 1326, 1340, 1527, 1538, 1568
font_used: 552, 624, 646, 1340
FONTx: 1260
for accent: 191
Forbidden control sequence...: 338
force_eof: 331, 361, 362, 381
formal: 1422, 1423, 1424
format_area_length: 523
format_debug: 1309, 1311
format_debug_end: 1309
format_default_length: 523, 525, 526, 527
format_engine: 1305, 1306, 1310, 1311
format_ext_length: 523, 526, 527
format_extension: 523, 532, 1331
format_ident: 61, 539, 1302, 1303, 1304, 1329, 1330, 1331, 1340
forward: 78, 218, 281, 340, 369, 412, 621, 695, 696, 723, 777, 803
found: 15, 125, 128, 129, 259, 341, 354, 356, 392, 395, 397, 451, 458, 476, 478, 480, 527, 610, 612, 615, 616, 617, 622, 648, 709, 711, 723, 898, 926, 934, 937, 943, 944, 956, 958, 1141, 1149, 1150, 1151, 1239, 1240, 1391, 1396, 1398, 1400
found1: 15, 898, 905, 1305, 1318
found2: 15, 898, 906, 1305, 1319
four_choices: 113
four_quarters: 551, 552, 557, 558, 563, 652, 686, 687, 709, 712, 715, 727, 741, 752, 909, 1035, 1126, 1326, 1340, 1397, 1399
fputs: 61, 527, 539
fraction_noad: 686, 690, 693, 701, 736, 764, 1181, 1184
fraction_noad_size: 686, 701, 764, 1184
fraction_rule: 707, 708, 738, 750
free: 165, 167, 168, 169, 170, 171
free_arr: 165
free_avail: 121, 202, 204, 217, 403, 455, 775, 918, 1039, 1229, 1291
free_node: 130, 201, 202, 275, 499, 618, 658, 701, 718, 724, 730, 754, 756, 759, 763, 775, 806, 863, 864, 868, 906, 913, 980, 1022, 1024, 1025, 1040, 1103, 1113, 1189, 1190, 1204, 1338, 1361
freeze_page_specs: 990, 1004, 1011
frozen_control_sequence: 222, 258, 1218, 1321, 1322
frozen_cr: 222, 339, 783, 1135
frozen_dont_expand: 222, 258, 372
frozen_end_group: 222, 265, 1068
frozen_end_template: 222, 378, 783
frozen_endv: 222, 378, 383, 783
frozen_fi: 222, 336, 494
frozen_null_font: 222, 556
frozen_protection: 222, 1218, 1219
frozen_relax: 222, 265, 382
frozen_right: 222, 1068, 1191
frozen_special: 222, 1347, 1406
Fuchs, David Raymond: 2, 586, 594
full_source_filename_stack: 304, 328, 331, 540, 1335, 1387
\futurelet primitive: 1222
fw: 588
fw_one_fifth: 1439
fw_times_sd: 1441, 1514, 1547, 1555
fw_two: 1439
fw_unity: 1439, 1440, 1474
fwrite: 600
g: 47, 182, 563, 595, 652, 671, 709, 719
g_cespace_shrink: 1479, 1559, 1560, 1563
g_cespace_stretch: 1479, 1559, 1560, 1563
g_cespace_width: 1479, 1559, 1560, 1563

g_cspace_shrink: 1478, <u>1559</u>, 1560, 1563
g_cspace_stretch: 1478, <u>1559</u>, 1560, 1563
g_cspace_width: 1478, <u>1559</u>, 1560, 1563
g_order: <u>622</u>, 628, <u>632</u>, 637
g_sign: <u>622</u>, 628, <u>632</u>, 637
garbage: <u>162</u>, 470, 473, 963, 1186, 1195, 1282
`\gdef` primitive: <u>1211</u>
geq_define: <u>279</u>, 785, 1080, 1217
geq_word_define: <u>279</u>, 288, 1016, 1217
get: 26, 29, 31, 488, 541, 567
get_avail: <u>120</u>, 122, 204, 205, 216, 325, 337, 339, 372, 374, 375, 455, 476, 485, 585, 712, 775, 786, 787, 797, 911, 914, 941, 1067, 1068, 1229, 1374, 1406
get_cat_code: 262, 343, 354, 356, <u>1413</u>
get_date_and_time: 241
get_job_name: 537, 540
get_next: 76, 297, 332, 336, 340, <u>341</u>, 357, 360, 364, 365, 366, 369, 372, 383, 384, 390, 392, 481, 497, 510, 647, 1129, 1453, 1460
get_node: <u>125</u>, 131, 136, 139, 144, 145, 147, 151, 152, 153, 156, 158, 206, 498, 610, 652, 671, 689, 691, 692, 719, 775, 801, 846, 847, 848, 867, 917, 1012, 1103, 1104, 1166, 1168, 1184, 1251, 1252, 1352, 1360, 1450
get_nullstr: <u>1407</u>
get_preamble_token: <u>785</u>, 786, 787
get_r_token: <u>1218</u>, 1221, 1224, 1227, 1228, 1260, 1482
get_strings_started: <u>47</u>, 51, 1335
get_token: 76, 78, 88, 364, <u>365</u>, 371, 372, 395, 402, 445, 455, 474, 476, 477, 479, 480, 482, 486, 785, 1030, 1141, 1218, 1224, 1255, 1271, 1274, 1297, 1374, 1375
get_wchar: <u>343</u>, 354, 356
get_x_token: 364, 369, 375, <u>383</u>, 384, 405, 407, 409, 410, 446, 447, 448, 455, 468, 482, 509, 529, 783, 938, 964, 1032, 1033, 1141, 1196, 1200, 1240, 1378, 1418, 1461, 1462, 1463, 1464, 1467
get_x_token_or_active_char: <u>509</u>
getc: 567
give_err_help: 78, 89, 90, <u>1287</u>
global: <u>1217</u>, 1221, 1244
global definitions: 221, 279, 283
`\global` primitive: <u>1211</u>
global_defs: <u>236</u>, 785, 1217, 1221
`\globaldefs` primitive: <u>238</u>
global_defs_code: <u>236</u>, 237, 238
glue_base: 220, <u>222</u>, 224, 226, 227, 228, 229, 252, 785
glue_node: <u>149</u>, 152, 153, 175, 183, 202, 206, 427, 625, 634, 654, 672, 733, 735, 764, 819, 820, 840, 859, 865, 869, 882, 884, 902, 906, 971, 975, 976, 991, 999, 1000, 1003, 1109, 1110, 1111, 1150, 1205, 1450
glue_offset: <u>135</u>, 159, 186
glue_ord: <u>150</u>, 450, 622, 632, 649, 652, 671, 794
glue_order: <u>135</u>, 136, 159, 185, 186, 622, 632, 660, 661, 667, 675, 676, 679, 772, 799, 804, 810, 812, 813, 814, 1151
glue_par: <u>224</u>, 769
glue_pars: <u>224</u>
glue_ptr: <u>149</u>, 152, 153, 175, 189, 190, 202, 206, 427, 628, 637, 659, 674, 682, 735, 789, 796, 798, 805, 806, 812, 819, 841, 871, 884, 972, 979, 999, 1004, 1007, 1151, 1450, 1451, 1455
glue_ratio: 109, 110, 135, 186
glue_ref: <u>210</u>, 228, 275, 785, 1231, 1239
glue_ref_count: <u>150</u>, 151, 152, 153, 154, 164, 201, 203, 228, 769, 1046, 1063, 1450, 1451, 1455
glue_set: <u>135</u>, 136, 159, 186, 628, 637, 660, 661, 667, 675, 676, 679, 810, 812, 813, 814, 1151
glue_shrink: <u>159</u>, 185, 799, 802, 804, 813, 814
glue_sign: <u>135</u>, 136, 159, 185, 186, 622, 632, 660, 661, 667, 675, 676, 679, 772, 799, 804, 810, 812, 813, 814, 1151
glue_spec_size: <u>150</u>, 151, 162, 164, 201, 719
glue_stretch: <u>159</u>, 185, 799, 802, 804, 813, 814
glue_temp: <u>622</u>, 628, <u>632</u>, 637
glue_val: <u>413</u>, 414, 415, 416, 419, 420, 427, 430, 432, 433, 454, 464, 468, 785, 1063, 1231, 1239, 1240, 1241, 1243, 1558
`goal height`: 989, 990
goto: <u>35</u>, <u>81</u>
gr: 110, 114, 135
group_code: <u>269</u>, 271, 274, 648, 1139
gubed: <u>7</u>
Guibas, Leonidas Ioannis: 2
g1: <u>1201</u>, 1206
g2: <u>1201</u>, 1206, 1208
h: <u>204</u>, <u>259</u>, <u>652</u>, <u>671</u>, <u>741</u>, <u>932</u>, <u>937</u>, <u>947</u>, <u>951</u>, <u>956</u>, <u>973</u>, <u>980</u>, <u>997</u>, <u>1089</u>, <u>1094</u>, <u>1126</u>, <u>1482</u>
h_offset: <u>247</u>, 620, 644
`\hoffset` primitive: <u>248</u>
h_offset_code: <u>247</u>, 248
ha: <u>895</u>, 899, 903, 906, 915
half: <u>100</u>, 709, 739, 740, 741, 748, 749, 752, 753, 1205
half_buf: 597, <u>598</u>, 599, 601, 602, 643
half_error_line: 14, <u>32</u>, 311, 315, 316, 317, 1335
halfword: 108, 110, <u>113</u>, 115, 130, 264, 277, 279, 280, 281, 297, 298, 300, 333, 341, 369, 392, 416, 467, 476, 552, 563, 580, 684, 794, 803, 824, 832, 833, 836, 850, 875, 880, 895, 904,

909, 910, 1035, 1082, 1214, 1246, 1269, 1291, 1326, 1335, 1340, 1413, 1415
halign: 208, 265, 266, 1097, 1133
\halign primitive: 265
halt_on_error_p: 32, 82
handle_right_brace: 1070, 1071
hang_after: 236, 240, 850, 852, 1073, 1152
\hangafter primitive: 238
hang_after_code: 236, 237, 238, 1073
hang_indent: 247, 850, 851, 852, 1073, 1152
\hangindent primitive: 248
hang_indent_code: 247, 248, 1073
hanging indentation: 850
hash: 234, 256, 259, 260, 1311, 1321, 1322, 1335
hash_base: 11, 220, 222, 256, 259, 262, 263, 290, 1260, 1311, 1317, 1321, 1322, 1335, 1482
hash_brace: 476, 479
hash_extra: 256, 260, 290, 1311, 1319, 1320, 1335, 1337
hash_high: 256, 258, 260, 1310, 1311, 1319, 1320, 1321, 1322
hash_is_full: 256, 260
hash_offset: 11, 290, 1311, 1335
hash_prime: 12, 14, 259, 261, 1310, 1311
hash_size: 11, 12, 14, 222, 260, 261, 1337
hash_top: 256, 1311, 1317, 1335
hash_used: 256, 258, 260, 1321, 1322, 1335
hb: 895, 900, 901, 903, 906
hbadness: 236, 663, 669, 670
\hbadness primitive: 238
hbadness_code: 236, 237, 238
\hbox primitive: 1074
hbox_group: 269, 274, 1086, 1088
hbox_tail: 1448, 1462, 1463
hc: 895, 896, 900, 901, 903, 904, 922, 923, 926, 933, 934, 937, 940, 942, 963, 965, 966, 968
hchar: 908, 909, 911, 912
hd: 652, 657, 709, 711, 712, 715
head: 212, 213, 215, 216, 217, 427, 721, 779, 799, 802, 808, 815, 817, 819, 1029, 1037, 1057, 1083, 1084, 1089, 1094, 1099, 1103, 1108, 1116, 1122, 1124, 1148, 1162, 1171, 1179, 1184, 1187, 1188, 1190, 1194, 1311, 1451, 1454, 1462, 1463, 1464
head_field: 212, 213, 218
head_for_vmode: 1097, 1098
head_forbidden: 207, 1412, 1447
header: 545
Hedrick, Charles Locke: 3
height: 135, 136, 138, 139, 140, 184, 187, 188, 466, 557, 625, 627, 629, 632, 634, 635, 638, 640, 643, 644, 652, 656, 659, 673, 675, 682, 707, 709, 712, 714, 716, 730, 733, 738, 739, 740, 741, 742, 745, 748, 749, 750, 752, 753, 754, 759, 760, 762, 771, 772, 799, 804, 807, 809, 810, 812, 813, 814, 972, 976, 984, 989, 1004, 1005, 1011, 1012, 1013, 1024, 1090, 1103
height: 466
height_base: 553, 557, 569, 574, 1325, 1326, 1340
height_depth: 557, 657, 711, 712, 715, 1128, 1402
height_index: 546, 557
height_offset: 135, 419, 420, 772, 1250
height_plus_depth: 715, 717
held over for next output: 989
help_line: 79, 89, 90, 336, 1109
help_ptr: 79, 80, 89, 90
help0: 79, 1255, 1296
help1: 79, 93, 95, 288, 411, 431, 457, 479, 489, 503, 506, 513, 963, 964, 965, 966, 1069, 1083, 1102, 1124, 1135, 1138, 1154, 1157, 1162, 1180, 1195, 1215, 1216, 1235, 1240, 1246, 1247, 1261, 1286, 1307, 1416, 1445, 1461, 1492, 1546
help2: 72, 79, 88, 89, 94, 95, 288, 346, 376, 436, 437, 438, 439, 440, 445, 448, 463, 478, 479, 580, 582, 644, 939, 940, 981, 1018, 1030, 1050, 1071, 1083, 1085, 1098, 1109, 1123, 1132, 1169, 1200, 1210, 1228, 1239, 1244, 1262, 1375, 1388, 1419, 1444, 1488, 1489, 1490, 1493, 1535, 1544, 1558
help3: 72, 79, 98, 336, 399, 418, 449, 482, 779, 786, 787, 795, 996, 1012, 1027, 1031, 1081, 1087, 1113, 1130, 1186, 1198, 1296, 1491
help4: 79, 89, 338, 401, 406, 421, 459, 570, 726, 979, 1007, 1053, 1286
help5: 79, 373, 564, 829, 1067, 1072, 1131, 1218, 1296
help6: 79, 398, 462, 1131, 1164
Here is how much...: 1337
hex_to_cur_chr: 352, 355
hex_token: 441, 447
hf: 895, 899, 900, 901, 906, 911, 912, 913, 914, 918, 919
\hfil primitive: 1061
\hfilneg primitive: 1061
\hfill primitive: 1061
hfinish: 1385
hfuzz: 247, 669
\hfuzz primitive: 248
hfuzz_code: 247, 248
hh: 110, 114, 118, 133, 182, 213, 219, 221, 268, 689, 745, 1166, 1168, 1184, 1189
hi: 112, 232, 557, 1227, 1235
hi_byte: 1409
hi_mem_min: 116, 118, 120, 125, 126, 134, 164, 165, 167, 168, 171, 172, 176, 293, 642, 1314, 1315, 1337

hi_mem_stat_min: 162, 164, 1315
hi_mem_stat_usage: 162, 164
history: 76, 77, 81, 82, 93, 95, 245, 1335, 1338
hlist_node: 135, 136, 137, 138, 148, 159, 175, 183, 184, 202, 206, 508, 621, 622, 625, 634, 647, 652, 654, 672, 684, 810, 813, 817, 844, 845, 869, 873, 874, 971, 976, 996, 1003, 1077, 1083, 1090, 1113, 1150, 1206
hlist_out: 595, 618, 619, 621, 622, 623, 626, 631, 632, 635, 640, 641, 643, 696, 1376, 1398, 1399
hlp1: 79
hlp2: 79
hlp3: 79
hlp4: 79
hlp5: 79
hlp6: 79
hmode: 211, 218, 419, 504, 789, 790, 799, 802, 1033, 1048, 1049, 1051, 1059, 1060, 1074, 1076, 1079, 1082, 1086, 1089, 1094, 1095, 1096, 1097, 1099, 1100, 1112, 1113, 1115, 1119, 1120, 1122, 1125, 1133, 1140, 1203, 1246, 1380, 1451, 1454, 1461
hmove: 208, 1051, 1074, 1075, 1076
hn: 895, 900, 901, 902, 905, 915, 916, 918, 919, 920, 922, 926, 933, 934
ho: 112, 235, 417, 557, 1154, 1157
hold_head: 162, 306, 782, 786, 787, 797, 811, 908, 909, 916, 917, 918, 919, 920, 1017, 1020
holding_inserts: 236, 1017
\holdinginserts primitive: 238
holding_inserts_code: 236, 237, 238
hpack: 162, 236, 647, 648, 649, 650, 652, 664, 712, 718, 723, 730, 740, 751, 757, 759, 799, 802, 807, 809, 892, 1065, 1089, 1128, 1197, 1202, 1204, 1207
hrule: 208, 265, 266, 466, 1049, 1059, 1087, 1097, 1098
\hrule primitive: 265
hsize: 247, 850, 851, 852, 1057, 1152
\hsize primitive: 248
hsize_code: 247, 248
hskip: 208, 1060, 1061, 1062, 1081, 1093
\hskip primitive: 1061
\hss primitive: 1061
hstart: 1385
\ht primitive: 419
hu: 895, 896, 900, 901, 904, 906, 908, 910, 911, 913, 914, 915, 918, 919
Huge page...: 644
hundred_million_wchar_offset: 1421, 1422
hundred_wchar_offset: 1421, 1424
hyf: 903, 905, 908, 911, 912, 916, 917, 922, 923, 926, 927, 935, 963, 964, 965, 966, 968
hyf_bchar: 895, 900, 901, 906
hyf_char: 895, 899, 916, 918
hyf_distance: 923, 924, 925, 927, 946, 947, 948, 1327, 1328
hyf_next: 923, 924, 927, 946, 947, 948, 1327, 1328
hyf_node: 915, 918
hyf_num: 923, 924, 927, 946, 947, 948, 1327, 1328
hyph_count: 929, 931, 943, 944, 1327, 1328, 1337
hyph_data: 209, 1213, 1253, 1254, 1255
hyph_link: 928, 929, 931, 933, 943, 1327, 1328, 1335
hyph_list: 929, 931, 932, 935, 936, 937, 943, 944, 1327, 1328, 1335
hyph_next: 929, 931, 943, 1327, 1328
hyph_pointer: 928, 929, 930, 932, 937, 1335
hyph_prime: 11, 12, 931, 933, 942, 943, 1310, 1311, 1327, 1328
hyph_size: 32, 931, 936, 943, 1327, 1328, 1335, 1337
hyph_word: 929, 931, 932, 934, 937, 943, 944, 1327, 1328, 1335
hyphen_char: 429, 552, 579, 894, 899, 1038, 1120, 1256, 1325, 1326, 1340
\hyphenchar primitive: 1257
hyphen_passed: 908, 909, 912, 916, 917
hyphen_penalty: 145, 236, 872
\hyphenpenalty primitive: 238
hyphen_penalty_code: 236, 237, 238
hyphen_size: 1327
hyphenate: 897, 898
hyphenated: 822, 823, 832, 849, 862, 872, 876
Hyphenation trie...: 1327
\hyphenation primitive: 1253
i: 19, 315, 590, 652, 741, 752, 904, 1126, 1524
I can't find file x: 533
I can't find the format...: 527
I can't go on...: 95
I can't read TEX.POOL: 51
I can't write on file x: 533
i_flag: 1486, 1487, 1493
ia_c: 1399, 1400, 1402
ib_c: 1399, 1400, 1402
id: 1481, 1482, 1484, 1497
id_byte: 590, 620, 645
id_lookup: 259, 264, 356, 377, 1524
ident_val: 413, 418, 468, 469
\ifcase primitive: 490
if_case_code: 490, 491, 504
if_cat_code: 490, 491, 504
\ifcat primitive: 490

\if primitive: 490
if_char_code: 490, 504, 509
if_code: 492, 498, 513
\ifdim primitive: 490
if_dim_code: 490, 491, 504
\ifeof primitive: 490
if_eof_code: 490, 491, 504
\iffalse primitive: 490
if_false_code: 490, 491, 504
\ifhbox primitive: 490
if_hbox_code: 490, 491, 504, 508
\ifhmode primitive: 490
if_hmode_code: 490, 491, 504
\ifinner primitive: 490
if_inner_code: 490, 491, 504
\ifnum primitive: 490
if_int_code: 490, 491, 504, 506
if_limit: 492, 493, 498, 499, 500, 501, 513
if_line: 492, 493, 498, 499, 1338
if_line_field: 492, 498, 499, 1338
\ifmmode primitive: 490
if_mmode_code: 490, 491, 504
if_node_size: 492, 498, 499, 1338
\ifodd primitive: 490
if_odd_code: 490, 491, 504
if_test: 210, 336, 366, 370, 490, 491, 497, 501, 506, 1338
\iftrue primitive: 490
if_true_code: 490, 491, 504
\ifvbox primitive: 490
if_vbox_code: 490, 491, 504
\ifvmode primitive: 490
if_vmode_code: 490, 491, 504
\ifvoid primitive: 490
if_void_code: 490, 491, 504, 508
ifdef: 7, 8, 643, 645
ifndef: 645
\ifx primitive: 490
ifx_code: 490, 491, 504
ignore: 207, 232, 332, 345
ignore_depth: 212, 215, 219, 682, 790, 1028, 1059, 1086, 1102, 1170
ignore_spaces: 208, 265, 266, 1048
\ignorespaces primitive: 265
iinf_hyphen_size: 11, 12
Illegal magnification...: 288, 1261
Illegal math \disc...: 1123
Illegal parameter number...: 482
Illegal unit of measure: 457, 459, 462
\immediate primitive: 1347
immediate_code: 1347, 1349, 1351
IMPOSSIBLE: 262
Improper \halign...: 779
Improper \hyphenation...: 939, 1444
Improper \prevdepth: 421
Improper \setbox: 1244
Improper \spacefactor: 421
Improper 'at' size...: 1262
Improper alphabetic constant: 445
Improper discretionary list: 1124
in: 461
in_open: 304, 328, 329, 331, 540, 1387, 1406
in_set_box: 1244, 1448, 1462
in_state_record: 300, 301, 1335
in_stream: 208, 1275, 1276, 1277
Incompatible glue units: 411
Incompatible list...: 1113
Incompatible magnification: 288
incompleat_noad: 212, 213, 721, 779, 1139, 1181, 1184, 1185, 1187, 1188
Incomplete \if...: 336
incr: 37, 42, 43, 45, 46, 53, 58, 59, 60, 65, 67, 70, 71, 82, 90, 98, 120, 122, 152, 153, 170, 182, 203, 216, 260, 274, 276, 280, 294, 311, 312, 321, 325, 328, 343, 347, 352, 354, 355, 357, 360, 362, 369, 377, 395, 398, 400, 402, 403, 406, 410, 445, 455, 457, 467, 478, 479, 480, 497, 520, 521, 522, 527, 528, 534, 540, 583, 601, 622, 632, 643, 645, 648, 717, 801, 848, 880, 900, 901, 913, 914, 917, 918, 926, 933, 934, 940, 942, 943, 944, 947, 957, 959, 965, 966, 967, 989, 1025, 1028, 1038, 1042, 1072, 1102, 1120, 1122, 1124, 1130, 1145, 1156, 1175, 1177, 1318, 1319, 1321, 1328, 1340, 1431, 1450, 1451, 1455, 1481, 1485, 1497, 1510, 1511, 1512, 1514, 1523, 1524, 1525, 1529, 1545, 1547, 1553, 1554, 1555, 1563, 1568, 1569, 1570, 1571
\indent primitive: 1091
indent_in_hmode: 1095, 1096
indented: 1094
index: 300, 302, 303, 304, 307, 328, 329, 331
index_field: 300, 302, 1134
inf: 450, 451, 456, 1335
inf_bad: 108, 157, 854, 855, 856, 859, 866, 977, 1008, 1020
inf_buf_size: 11
inf_dvi_buf_size: 11
inf_expand_depth: 11
inf_font_max: 11
inf_font_mem_size: 11
inf_hash_extra: 11
inf_hyph_size: 11
inf_main_memory: 11
inf_max_in_open: 11
inf_max_strings: 11

inf_mem_bot: 11
inf_nest_size: 11
inf_param_size: 11
inf_penalty: 157, 764, 770, 819, 832, 834, 977, 1008, 1016, 1206, 1208, 1451, 1452, 1453, 1457, 1458, 1459, 1463
inf_pool_free: 11
inf_pool_size: 11
inf_save_size: 11
inf_stack_size: 11
inf_string_vacancies: 11
inf_strings_free: 11
inf_trie_size: 11
`Infinite glue shrinkage...`: 829, 979, 1007, 1012
infinity: 448
info: 118, 124, 126, 140, 164, 172, 200, 233, 275, 291, 293, 325, 337, 339, 357, 358, 372, 374, 377, 392, 394, 395, 396, 397, 400, 403, 426, 455, 469, 511, 608, 611, 612, 613, 614, 615, 616, 617, 618, 684, 692, 695, 696, 701, 723, 737, 738, 739, 740, 741, 745, 752, 757, 771, 772, 775, 782, 786, 787, 793, 796, 797, 800, 801, 804, 806, 824, 850, 851, 928, 935, 941, 984, 1068, 1079, 1096, 1152, 1154, 1171, 1184, 1188, 1189, 1194, 1229, 1251, 1252, 1292, 1315, 1342, 1344, 1374, 1406
ini_version: 8, 32, 1304
Init: 8, 1255, 1335, 1338
init: 8, 32, 47, 50, 131, 264, 894, 945, 946, 950, 953, 1305, 1328, 1339, 1340
init_align: 776, 777, 1133
init_col: 776, 788, 791, 794
init_cur_lang: 819, 894, 895
init_l_hyf: 819, 894, 895
init_lft: 903, 906, 908, 911
init_lig: 903, 906, 908, 911
init_list: 903, 906, 908, 911
init_math: 1140, 1141
init_pool_ptr: 39, 42, 1313, 1335, 1337
init_prim: 1335, 1339
init_r_hyf: 819, 894, 895
init_row: 776, 788, 789
init_span: 776, 789, 790, 794
init_str_ptr: 39, 43, 520, 1313, 1335, 1337
init_terminal: 37, 331
init_trie: 894, 969, 1327
`INITEX`: 8, 11, 12, 47, 50, 116, 1302, 1334
initialize: 4, 1335, 1340
inner loop: 31, 112, 120, 121, 122, 123, 125, 127, 128, 130, 202, 324, 325, 341, 342, 343, 357, 365, 383, 402, 410, 557, 600, 614, 623, 654, 657, 658, 835, 838, 854, 855, 870, 1033, 1042, 1044, 1396, 1397
inner_noad: 685, 686, 693, 699, 701, 736, 764, 767, 1159, 1160, 1194
input: 210, 366, 370, 379, 380
Input Encoding Dependencies: 61
`\input` primitive: 379
input_file: 304, 1335
`\inputlineno` primitive: 419
input_line_no_code: 419, 420, 427
input_ln: 30, 31, 37, 58, 71, 362, 488, 489, 541
input_ptr: 301, 311, 312, 321, 322, 330, 331, 360, 537, 1134, 1338
input_stack: 84, 301, 311, 321, 322, 537, 1134, 1335
ins_disc: 1035, 1036, 1038
ins_error: 327, 336, 398, 1050, 1130, 1135, 1218
ins_list: 323, 339, 470, 473, 1067, 1374, 1406
ins_node: 140, 148, 175, 183, 202, 206, 650, 654, 733, 764, 869, 902, 971, 976, 984, 989, 1003, 1017, 1103
ins_node_size: 140, 202, 206, 1025, 1103
ins_ptr: 140, 188, 202, 206, 1013, 1023, 1024, 1103
ins_the_toks: 369, 370, 470
insert: 208, 265, 266, 1100
`insert>`: 87
`\insert` primitive: 265
insert_dollar_sign: 1048, 1050
insert_group: 269, 1071, 1102, 1103
insert_penalties: 422, 985, 993, 1008, 1011, 1013, 1017, 1025, 1029, 1245, 1249
`\insertpenalties` primitive: 419
insert_relax: 381, 382, 513
insert_src_special: 1094, 1142, 1170, 1406
insert_src_special_auto: 32, 1037
insert_src_special_every_cr: 32
insert_src_special_every_display: 32
insert_src_special_every_hbox: 32
insert_src_special_every_math: 32, 1142
insert_src_special_every_par: 32, 1094
insert_src_special_every_parend: 32
insert_src_special_every_vbox: 32, 1170
insert_token: 268, 280, 282
inserted: 307, 314, 323, 324, 327, 382, 1098
inserting: 984, 1012
`Insertions can only...`: 996
inserts_only: 983, 990, 1011
int: 110, 113, 114, 140, 141, 157, 186, 213, 219, 236, 240, 242, 274, 278, 279, 416, 417, 492, 608, 728, 772, 775, 822, 1241, 1243, 1319, 1428, 1544, 1546
int_base: 220, 230, 232, 236, 238, 239, 240, 242, 252, 253, 254, 268, 283, 288, 1016, 1073, 1142,

1148, 1227, 1318, 1425, 1426, 1428, 1475, 1542, 1543, 1544, 1546
int_error: 91, 288, 436, 437, 438, 439, 440, 1246, 1247, 1261, 1388, 1419, 1492
int_par: 236
int_pars: 236
int_val: 413, 414, 415, 416, 417, 419, 420, 421, 422, 425, 426, 427, 429, 430, 431, 432, 442, 443, 452, 464, 468, 1239, 1240, 1241, 1243, 1428
intcast: 127, 862, 878, 947, 951
integer: 3, 11, 13, 19, 32, 38, 45, 47, 54, 59, 60, 63, 65, 66, 67, 69, 82, 91, 94, 96, 100, 101, 102, 105, 106, 107, 108, 109, 110, 113, 117, 125, 158, 163, 172, 173, 174, 176, 177, 178, 181, 182, 211, 212, 218, 225, 237, 247, 256, 259, 262, 278, 279, 286, 292, 304, 308, 309, 311, 315, 341, 367, 369, 413, 443, 451, 453, 473, 485, 492, 496, 497, 501, 521, 522, 523, 526, 551, 552, 553, 563, 580, 581, 595, 598, 603, 604, 605, 610, 618, 619, 622, 632, 641, 648, 649, 664, 694, 697, 702, 709, 719, 720, 729, 741, 755, 767, 818, 831, 832, 833, 836, 875, 880, 895, 915, 925, 929, 969, 973, 983, 985, 997, 1015, 1033, 1035, 1071, 1078, 1082, 1087, 1094, 1120, 1122, 1141, 1154, 1158, 1197, 1214, 1305, 1306, 1326, 1334, 1335, 1336, 1340, 1341, 1351, 1373, 1382, 1385, 1391, 1396, 1397, 1399, 1422, 1423, 1424, 1431, 1434, 1439, 1441, 1474, 1478, 1479, 1482, 1505, 1506, 1508, 1523, 1524, 1525, 1526, 1530, 1551, 1559
inter_line_penalty: 236, 893
`\interlinepenalty` primitive: 238
inter_line_penalty_code: 236, 237, 238
interaction: 71, 72, 73, 74, 75, 82, 84, 86, 90, 92, 93, 98, 360, 363, 487, 533, 1268, 1286, 1296, 1297, 1300, 1329, 1330, 1331, 1336, 1338
interaction_option: 73, 74, 1330
internal_cface_number: 1473, 1474, 1481, 1482, 1502, 1508, 1512, 1514, 1526, 1530, 1535, 1536
internal_cfont_number: 605, 1448, 1500, 1502, 1512, 1514, 1524, 1526, 1535, 1536, 1572
internal_ectbl_number: 1517, 1518, 1523
internal_font_number: 144, 551, 552, 563, 581, 584, 585, 595, 605, 619, 652, 709, 712, 714, 715, 718, 727, 741, 833, 865, 895, 1035, 1116, 1126, 1141, 1214, 1260, 1396, 1397, 1535
interrupt: 96, 97, 98, 1034
`Interruption`: 98
`interwoven alignment preambles...`: 324, 785, 792, 794, 1134
`Invalid code`: 1235, 1416
`Invalid range`: 1416
invalid_char: 207, 232, 344
invalid_code: 22, 24, 232
inverse: 1474, 1493
ipc_on: 643, 1382
ipc_page: 643
is_char_node: 134, 174, 183, 202, 205, 427, 623, 633, 654, 672, 718, 723, 724, 759, 808, 819, 840, 844, 845, 869, 870, 871, 873, 874, 882, 899, 900, 902, 906, 1039, 1043, 1083, 1084, 1108, 1116, 1124, 1150, 1205, 1451, 1453, 1462, 1463, 1464
is_digit: 1511, 1525
IS_DIR_SEP: 519
is_empty: 124, 127, 169, 170
is_head_forbidden: 1447, 1451, 1452, 1453, 1457, 1459, 1463
is_head_forbidden_wchar: 1447, 1457
is_hex: 352, 355
is_letter: 1510
is_new_source: 1406
is_punc_wchar: 1447, 1452, 1453, 1457, 1458, 1459, 1463
is_running: 138, 176, 627, 636, 809
is_tail_forbidden: 1447, 1458
is_wchar: 134, 294, 298, 354, 377, 475, 623, 657, 844, 845, 869, 873, 874, 940, 1033, 1154, 1157, 1163, 1418, 1452, 1459, 1460, 1463, 1464
is_wchar_node: 134, 174, 176, 718, 844, 845, 870, 873, 874, 1116, 1150, 1451, 1455, 1457, 1463, 1464
issue_message: 1279, 1282
ital_corr: 208, 265, 266, 1114, 1115
italic: 1474, 1493
italic correction: 546
italic_base: 553, 557, 569, 574, 1325, 1326, 1340
italic_index: 546
its_all_over: 1048, 1057, 1338
j: 45, 46, 59, 60, 69, 70, 259, 264, 315, 369, 520, 521, 522, 526, 527, 641, 896, 904, 909, 937, 969, 1214, 1260, 1305, 1306, 1373, 1376
Japanese characters: 134, 588
Jensen, Kathleen: 10
jj: 1508, 1510
`job aborted`: 360
`job aborted, file error...`: 533
job_name: 92, 474, 475, 530, 531, 532, 535, 537, 540, 1260, 1331, 1338, 1482
`\jobname` primitive: 471
job_name_code: 471, 473, 474, 475
jump_out: 81, 82, 84, 93
just_box: 817, 891, 892, 1149, 1151
just_open: 483, 486, 1278
k: 45, 46, 47, 64, 65, 67, 69, 71, 102, 163, 259, 264, 341, 363, 410, 453, 467, 522, 526, 528,

533, 537, 563, 590, 605, 610, 641, 708, 909, 932, 937, 963, 969, 1082, 1214, 1305, 1306, 1336, 1341, 1351, 1371, 1384, 1431, 1434, 1482, 1523, 1525, 1526, 1530
kern: 208, 548, 1060, 1061, 1062
\kern primitive: 1061
kern_base: 553, 560, 569, 576, 579, 1325, 1326, 1340
kern_base_offset: 560, 569, 576
kern_break: 869
kern_flag: 548, 744, 756, 912, 1043
kern_node: 155, 156, 183, 202, 206, 427, 625, 634, 654, 672, 724, 733, 735, 764, 840, 844, 845, 859, 869, 871, 873, 874, 882, 884, 899, 900, 902, 971, 975, 976, 979, 999, 1000, 1003, 1007, 1109, 1110, 1111, 1124, 1150
kk: 453, 455
Knuth, Donald Ervin: 2, 86, 696, 816, 894, 928, 1000, 1157, 1374
kpse_find_file: 566
kpse_in_name_ok: 540, 1278
kpse_make_tex_discard_errors: 1268
kpse_out_name_ok: 1377
kpse_tex_format: 540, 1278
kpse_texpool_format: 51
l: 47, 259, 264, 276, 281, 292, 315, 497, 500, 537, 604, 618, 671, 833, 904, 947, 956, 963, 1141, 1197, 1239, 1305, 1341, 1379
l_hyf: 894, 895, 897, 902, 905, 926, 1365
language: 236, 937, 1037, 1379
\language primitive: 238
language_code: 236, 237, 238
language_node: 1344, 1359, 1360, 1361, 1365, 1376, 1379, 1380
large_attempt: 709
large_char: 686, 694, 700, 709, 1163
large_fam: 686, 694, 700, 709, 1163
last: 30, 31, 35, 36, 37, 71, 83, 87, 88, 331, 360, 363, 486, 527, 534
last_active: 822, 823, 835, 838, 847, 857, 863, 864, 866, 867, 868, 876, 877, 878
last_badness: 427, 649, 651, 652, 663, 667, 670, 671, 677, 679, 681
last_bop: 595, 596, 643, 645
\lastbox primitive: 1074
last_box_code: 1074, 1075, 1082
last_glue: 215, 427, 985, 994, 999, 1020, 1109, 1338
last_ins_ptr: 984, 1008, 1011, 1021, 1023
last_item: 208, 416, 419, 420, 1051
last_kern: 215, 427, 985, 994, 999
\lastkern primitive: 419
last_penalty: 215, 427, 985, 994, 999
\lastpenalty primitive: 419
\lastskip primitive: 419
last_special_line: 850, 851, 852, 853, 892
last_text_char: 19, 24
lc_code: 230, 232, 894, 899, 900, 901, 940, 965
\lccode primitive: 1233
lc_code_base: 230, 235, 1233, 1234, 1289, 1290, 1291
leader_box: 622, 629, 631, 632, 638, 640
leader_flag: 1074, 1076, 1081, 1087
leader_ht: 632, 638, 639, 640
leader_ptr: 149, 152, 153, 190, 202, 206, 629, 638, 659, 674, 819, 1081, 1450
leader_ship: 208, 1074, 1075, 1076
leader_wd: 622, 629, 630, 631
leaders: 1377
Leaders not followed by...: 1081
\leaders primitive: 1074
least_cost: 973, 977, 983
least_page_cost: 983, 990, 1008, 1009
\left primitive: 1191
left_brace: 207, 289, 294, 298, 347, 357, 406, 476, 479, 780, 1066, 1153, 1229
left_brace_limit: 289, 325, 395, 397, 402
left_brace_token: 289, 406, 1130, 1229, 1374, 1406
left_delimiter: 686, 699, 700, 740, 751, 1166, 1184, 1185
left_edge: 622, 630, 632, 635, 640
left_hyphen_min: 236, 1094, 1203, 1379, 1380
\lefthyphenmin primitive: 238
left_hyphen_min_code: 236, 237, 238
left_noad: 690, 693, 699, 701, 728, 731, 736, 763, 764, 765, 1188, 1191, 1192, 1194
left_right: 208, 1049, 1191, 1192, 1193
left_skip: 224, 830, 883, 890
\leftskip primitive: 226
left_skip_code: 224, 225, 226, 890
len: 1391
length: 40, 46, 259, 522, 533, 540, 566, 605, 934, 944, 1260, 1283, 1391
length of lines: 850
\leqno primitive: 1144
let: 209, 1213, 1222, 1223, 1224
\let primitive: 1222
letter: 207, 232, 262, 289, 291, 294, 298, 347, 354, 356, 938, 964, 1032, 1033, 1093, 1127, 1154, 1157, 1163, 1411, 1418, 1452, 1459, 1460, 1463, 1464, 1487, 1493
letter_token: 289, 448
level: 413, 416, 418, 421, 431, 464, 1387
level_boundary: 268, 270, 274, 282

level_one: 221, 228, 232, 254, 264, 272, 277, 278, 279, 280, 281, 283, 783, 1307, 1338, 1372, 1480, 1501, 1519, 1532
level_zero: 221, 222, 272, 276, 280, 1311
lf: 543, 563, 568, 569, 578, 579
lft_hit: 909, 910, 911, 913, 914, 1036, 1038, 1043
lh: 110, 114, 118, 213, 219, 256, 543, 544, 563, 568, 569, 571, 688
Liang, Franklin Mark: 2, 922
libc_free: 522, 526, 1310, 1311
lig_char: 143, 144, 193, 206, 655, 844, 845, 869, 873, 874, 901, 906, 1116
lig_kern: 547, 548, 552
lig_kern_base: 553, 560, 569, 574, 576, 579, 1325, 1326, 1340
lig_kern_command: 544, 548
lig_kern_restart: 560, 744, 755, 912, 1042
lig_kern_restart_end: 560
lig_kern_start: 560, 744, 755, 912, 1042
lig_ptr: 143, 144, 175, 193, 202, 206, 899, 901, 906, 910, 913, 914, 1040, 1043
lig_stack: 910, 911, 913, 914, 1035, 1037, 1038, 1039, 1040, 1043, 1453, 1460
lig_tag: 547, 572, 744, 755, 912, 1042
lig_trick: 162, 655
ligature_node: 143, 144, 148, 175, 183, 202, 206, 625, 654, 755, 844, 845, 869, 873, 874, 899, 900, 902, 906, 1116, 1124, 1150
ligature_present: 909, 910, 911, 913, 914, 1036, 1038, 1040, 1043
limit: 300, 302, 303, 307, 318, 328, 330, 331, 343, 348, 350, 351, 352, 354, 355, 356, 360, 362, 363, 486, 529, 540, 541, 1340
`Limit controls must follow...`: 1162
limit_field: 35, 87, 300, 302, 537
limit_switch: 208, 1049, 1159, 1160, 1161
limits: 685, 699, 736, 752, 1159, 1160
`\limits` primitive: 1159
line: 84, 216, 304, 313, 328, 329, 331, 362, 427, 497, 498, 541, 666, 678, 1028, 1387, 1406
line_break: 162, 817, 818, 831, 842, 851, 865, 866, 869, 879, 897, 937, 970, 973, 985, 1099, 1148
line_diff: 875, 878
line_number: 822, 823, 836, 838, 848, 849, 853, 867, 875, 877, 878
line_penalty: 236, 862
`\linepenalty` primitive: 238
line_penalty_code: 236, 237, 238
line_skip: 224, 247
`\lineskip` primitive: 226
line_skip_code: 149, 152, 224, 225, 226, 682
line_skip_limit: 247, 682
`\lineskiplimit` primitive: 248
line_skip_limit_code: 247, 248
line_stack: 304, 328, 329, 1335, 1387
line_width: 833, 853, 854
link: 118, 120, 121, 122, 123, 124, 125, 126, 130, 133, 134, 135, 140, 143, 150, 164, 168, 172, 174, 175, 176, 182, 202, 204, 212, 214, 215, 218, 223, 233, 292, 295, 306, 319, 323, 339, 357, 358, 369, 372, 374, 377, 392, 393, 394, 397, 399, 400, 403, 410, 455, 467, 469, 470, 473, 481, 492, 498, 499, 500, 511, 608, 610, 612, 614, 618, 623, 625, 633, 652, 654, 655, 657, 658, 669, 672, 682, 684, 692, 708, 714, 718, 721, 722, 723, 724, 730, 734, 735, 738, 740, 741, 742, 750, 751, 754, 755, 756, 757, 758, 759, 762, 763, 764, 769, 770, 773, 775, 781, 782, 786, 787, 789, 793, 794, 796, 797, 798, 799, 800, 801, 802, 804, 805, 806, 807, 808, 809, 810, 811, 812, 815, 817, 819, 822, 824, 825, 832, 833, 840, 843, 846, 847, 848, 857, 860, 861, 863, 864, 865, 866, 867, 868, 869, 870, 872, 876, 877, 878, 880, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 893, 897, 899, 900, 901, 902, 906, 908, 909, 910, 911, 913, 914, 916, 917, 918, 919, 920, 921, 935, 941, 963, 971, 972, 973, 976, 982, 983, 984, 989, 991, 994, 997, 1001, 1002, 1003, 1004, 1008, 1011, 1012, 1017, 1020, 1021, 1022, 1023, 1024, 1025, 1026, 1029, 1038, 1039, 1040, 1043, 1044, 1046, 1067, 1068, 1079, 1084, 1089, 1094, 1103, 1104, 1108, 1113, 1122, 1123, 1124, 1126, 1128, 1149, 1158, 1171, 1184, 1187, 1188, 1189, 1190, 1194, 1197, 1199, 1202, 1207, 1208, 1209, 1229, 1282, 1291, 1300, 1314, 1315, 1338, 1342, 1344, 1352, 1371, 1374, 1378, 1406, 1451, 1454, 1456
list_offset: 135, 652, 772, 1021
list_ptr: 135, 136, 184, 202, 206, 622, 626, 632, 635, 661, 666, 667, 671, 676, 679, 712, 714, 718, 724, 742, 750, 754, 810, 980, 982, 1024, 1090, 1103, 1113, 1149, 1202
list_state_record: 212, 213, 1335
list_tag: 547, 572, 573, 711, 743, 752
ll: 956, 959
llink: 124, 126, 127, 129, 130, 131, 145, 149, 164, 169, 775, 822, 824, 1315
lo_byte: 1409
lo_mem_max: 116, 120, 125, 126, 164, 165, 167, 169, 170, 171, 172, 178, 642, 1314, 1315, 1326, 1337
lo_mem_stat_max: 162, 164, 1315
load_fmt_file: 1306, 1340
loc: 36, 37, 87, 300, 302, 303, 307, 312, 314, 318, 319, 323, 325, 328, 330, 331, 343, 348, 350, 351,

352, 354, 356, 357, 358, 360, 362, 372, 393, 486, 527, 529, 540, 541, 1029, 1030, 1340
loc_field: 35, 36, 300, 302, 1134
local_base: 220, 224, 228, 230, 252
local_names: 230, 1417, 1422, 1423, 1424, 1434, 1438
location: 608, 610, 615, 616, 617, 618
log_file: 54, 56, 75, 537, 539, 1336
log_name: 535, 537, 1336
log_only: 54, 57, 58, 62, 75, 98, 360, 537, 1331, 1373, 1377, 1567
log_opened: 92, 93, 530, 531, 537, 538, 1268, 1336, 1337, 1373, 1377
`\long` primitive: 1211
long_call: 210, 275, 366, 390, 392, 395, 402, 1298
long_help_seen: 1284, 1285, 1286
long_outer_call: 210, 275, 366, 390, 392, 1298
long_state: 339, 390, 394, 395, 398, 399, 402
lookup_cface: 1530, 1537
loop: 15, 16
`Loose \hbox...`: 663
`Loose \vbox...`: 677
loose_fit: 820, 837, 855
looseness: 236, 851, 876, 878, 1073
`\looseness` primitive: 238
looseness_code: 236, 237, 238, 1073
`\lower` primitive: 1074
lower_cdigit_base: 471, 475
`\lowercase` primitive: 1289
lq: 595, 630, 639
lr: 595, 630, 639
lx: 622, 629, 630, 631, 632, 638, 639, 640
m: 47, 65, 158, 211, 218, 292, 315, 392, 416, 443, 485, 501, 580, 652, 671, 709, 719, 720, 1082, 1108, 1197, 1341, 1422, 1524
mac_param: 207, 291, 294, 298, 347, 477, 480, 482, 786, 787, 1048
macro: 307, 314, 319, 323, 324, 393
macro_call: 291, 369, 383, 385, 390, 391, 392, 394
macro_def: 476, 480
mag: 236, 240, 288, 460, 588, 590, 591, 593, 620, 645
`\mag` primitive: 238
mag_code: 236, 237, 238, 288
mag_set: 286, 287, 288
magic_offset: 767, 768, 769
main_body: 1335
main_cf: 1037, 1448, 1451, 1452, 1453, 1457, 1459, 1463, 1464
main_control: 1032, 1033, 1035, 1043, 1044, 1055, 1057, 1058, 1059, 1060, 1129, 1137, 1211, 1293, 1335, 1340, 1347, 1350
main_f: 1035, 1037, 1038, 1039, 1040, 1042, 1043, 1460
main_i: 1035, 1039, 1040, 1042, 1043
main_j: 1035, 1042, 1043
main_k: 1035, 1037, 1042, 1043, 1045
main_lig_loop: 1033, 1037, 1040, 1042, 1043, 1460
main_loop: 1033, 1037, 1453, 1463, 1464
main_loop_lookahead: 1033, 1037, 1039, 1040, 1460
main_loop_move: 1033, 1037, 1039, 1043
main_loop_move_lig: 1033, 1037, 1039, 1040
main_loop_wchar: 1033, 1453, 1463, 1464
main_loop_wrapup: 1033, 1037, 1042, 1043
main_memory: 32, 1335
main_p: 1035, 1038, 1040, 1043, 1044, 1045, 1046, 1047
main_s: 1035, 1037
major_tail: 915, 917, 920, 921
make_accent: 1125, 1126, 1398, 1402
make_box: 208, 1074, 1075, 1076, 1082, 1087
make_cfont: 1509, 1514, 1527, 1538
make_cfont_id: 1524, 1527, 1538
make_fraction: 736, 737, 746
make_full_name_string: 540
make_left_right: 764, 765
make_mark: 1100, 1104
make_math_accent: 736, 741
make_name_string: 528
make_op: 736, 752
make_ord: 736, 755
make_over: 736, 737
make_radical: 736, 737, 740
make_scripts: 757, 759
make_src_special: 1406
make_string: 43, 48, 52, 260, 520, 528, 942, 1260, 1282, 1331, 1336, 1392, 1418, 1482, 1510, 1525
make_under: 736, 738
make_vcenter: 736, 739
mark: 208, 265, 266, 1100
`\mark` primitive: 265
mark_node: 141, 148, 175, 183, 202, 206, 650, 654, 733, 764, 869, 902, 971, 976, 982, 1003, 1017, 1104
mark_ptr: 141, 142, 196, 202, 206, 982, 1019, 1104
mark_text: 307, 314, 323, 389
mastication: 341
match: 207, 289, 291, 292, 294, 394, 395
match_chr: 292, 294, 392, 394, 403
match_ec_face: 1522, 1526
match_ec_font: 1534, 1535
match_token: 289, 394, 395, 396, 397, 479
matching: 305, 306, 339, 394
`Math formula deleted...`: 1198

math_ac: 1167, 1168
math_accent: 208, 265, 266, 1049, 1167
\mathaccent primitive: 265
\mathbin primitive: 1159
math_char: 684, 695, 723, 725, 727, 741, 744, 752, 755, 756, 757, 1154, 1158, 1168
\mathchar primitive: 265
\mathchardef primitive: 1225
math_char_def_code: 1225, 1226, 1227
math_char_num: 208, 265, 266, 1049, 1154, 1157
math_choice: 208, 265, 266, 1049, 1174
\mathchoice primitive: 265
math_choice_group: 269, 1175, 1176, 1177
\mathclose primitive: 1159
math_code: 230, 232, 236, 417, 1154, 1157
\mathcode primitive: 1233
math_code_base: 230, 235, 417, 1233, 1234, 1235, 1236
math_comp: 208, 1049, 1159, 1160, 1161
math_font_base: 230, 232, 234, 1233, 1234, 1532
math_fraction: 1183, 1184
math_given: 208, 416, 1049, 1154, 1157, 1225, 1226, 1227
math_glue: 719, 735, 769
math_group: 269, 1139, 1153, 1156, 1189
\mathinner primitive: 1159
math_kern: 720, 733
math_left_group: 269, 1068, 1071, 1072, 1153, 1194
math_left_right: 1193, 1194
math_limit_switch: 1161, 1162
math_mode_save: 1196, 1199, 1448
math_node: 147, 148, 175, 183, 202, 206, 625, 654, 820, 840, 869, 882, 884, 1150
\mathop primitive: 1159
\mathopen primitive: 1159
\mathord primitive: 1159
\mathpunct primitive: 1159
math_quad: 703, 706, 1202
math_radical: 1165, 1166
\mathrel primitive: 1159
math_shift: 207, 289, 294, 298, 347, 1093, 1140, 1141, 1196, 1200, 1209, 1453
math_shift_group: 269, 1068, 1071, 1072, 1133, 1142, 1143, 1145, 1148, 1195, 1196, 1197, 1203
math_shift_token: 289, 1050, 1068
math_spacing: 767, 768
math_style: 208, 1049, 1172, 1173, 1174
math_surround: 247, 1199
\mathsurround primitive: 248
math_surround_code: 247, 248
math_text_char: 684, 755, 756, 757, 758
math_type: 684, 686, 690, 695, 701, 723, 725, 726, 737, 738, 740, 741, 744, 745, 752, 754, 755, 756, 757, 758, 759, 1079, 1096, 1154, 1158, 1168, 1171, 1179, 1184, 1188, 1189, 1194
math_x_height: 703, 740, 760, 761, 762
mathex: 704
mathsy: 703
mathsy_end: 703
max_answer: 105
max_buf_stack: 30, 31, 331, 377, 1337
max_cface: 230, 1473, 1497, 1577, 1581
max_char_code: 207, 303, 341, 344, 1236, 1416
max_command: 209, 210, 211, 219, 358, 366, 369, 371, 383, 384, 481, 785
max_d: 729, 730, 733, 763, 764, 765
max_dead_cycles: 236, 240, 1015
\maxdeadcycles primitive: 238
max_dead_cycles_code: 236, 237, 238
max_depth: 247, 983, 990
\maxdepth primitive: 248
max_depth_code: 247, 248
max_dimen: 424, 463, 644, 671, 1013, 1020, 1148, 1149, 1151
max_ectbl: 1516, 1517, 1529, 1579
max_font_max: 11, 32, 111, 1324
max_group_code: 269
max_h: 595, 596, 644, 645, 729, 730, 733, 763, 764, 765
max_halfword: 14, 32, 110, 111, 112, 113, 124, 125, 126, 131, 132, 215, 289, 290, 427, 823, 851, 853, 923, 985, 994, 999, 1020, 1109, 1252, 1310, 1311, 1326, 1328, 1338
max_in_open: 14, 32, 304, 328, 1335, 1387
max_in_stack: 301, 321, 331, 1337
max_internal: 209, 416, 443, 451, 458, 464
max_nest_stack: 213, 215, 216, 1337
max_non_prefixed_command: 208, 1214, 1273
max_op_used: 946, 947, 949
max_param_stack: 308, 331, 393, 1337
max_print_line: 14, 32, 54, 58, 61, 72, 176, 540, 641, 1283, 1335
max_push: 595, 596, 622, 632, 645
max_quarterword: 32, 110, 111, 113, 274, 800, 801, 923, 1123
max_save_stack: 271, 272, 273, 1337
max_selector: 54, 246, 311, 468, 473, 537, 641, 1260, 1282, 1371, 1373, 1376, 1564
max_strings: 32, 43, 111, 520, 528, 1313, 1335, 1337
max_trie_op: 11, 923, 947, 1328
max_type_code: 207, 1236, 1416
max_v: 595, 596, 644, 645

max_val: 473, 1437
maxint: 11
\meaning primitive: 471
meaning_code: 471, 472, 474, 475
med_mu_skip: 224
\medmuskip primitive: 226
med_mu_skip_code: 224, 225, 226, 769
mem: 32, 115, 116, 118, 124, 126, 131, 133, 134, 135, 140, 141, 150, 151, 157, 159, 162, 163, 164, 165, 167, 172, 182, 186, 203, 205, 206, 221, 224, 275, 291, 390, 423, 492, 608, 655, 683, 684, 686, 689, 690, 723, 728, 745, 756, 772, 773, 775, 800, 819, 821, 822, 825, 826, 835, 846, 847, 850, 851, 853, 863, 864, 892, 928, 1152, 1154, 1163, 1166, 1168, 1184, 1189, 1250, 1251, 1311, 1314, 1315, 1335, 1342
mem_bot: 14, 32, 111, 116, 125, 126, 162, 164, 1310, 1311, 1314, 1315, 1335
mem_end: 116, 118, 120, 164, 165, 167, 168, 171, 172, 174, 176, 182, 293, 1314, 1315, 1337
mem_max: 12, 14, 32, 110, 111, 116, 120, 124, 125, 166, 1311, 1335
mem_min: 12, 32, 111, 116, 120, 125, 166, 167, 169, 170, 171, 172, 174, 178, 182, 1252, 1311, 1315, 1335, 1337
mem_top: 14, 32, 111, 116, 162, 164, 1252, 1310, 1311, 1315, 1335
Memory usage...: 642
memory_word: 110, 113, 114, 116, 182, 212, 218, 221, 253, 268, 271, 275, 551, 803, 1308, 1311, 1335
message: 208, 1279, 1280, 1281
\message primitive: 1280
METAFONT: 592
mid: 549
mid_line: 87, 303, 328, 344, 347, 352, 353, 354
min_ectbl: 1516, 1517, 1519, 1523, 1571, 1578, 1579
min_halfword: 32, 110, 111, 112, 113, 115, 230, 1030, 1326, 1328
min_internal: 208, 416, 443, 451, 458, 464
min_quarterword: 11, 110, 111, 112, 113, 134, 136, 140, 185, 221, 274, 551, 553, 557, 559, 560, 569, 579, 652, 671, 688, 700, 710, 716, 717, 799, 804, 806, 811, 961, 997, 1015, 1326, 1327, 1328
min_trie_op: 11, 923, 926, 927, 946, 947, 948, 949, 961, 966, 967, 968
min_val: 473, 1437
minimal_demerits: 836, 837, 839, 848, 858
minimum_demerits: 836, 837, 838, 839, 857, 858
minor_tail: 915, 918, 919
minus: 465
Misplaced &: 1131
Misplaced \cr: 1131
Misplaced \noalign: 1132
Misplaced \omit: 1132
Misplaced \span: 1131
Missing = inserted: 506
Missing # inserted...: 786
Missing $ inserted: 1050, 1068
Missing \cr inserted: 1135
Missing \endcsname...: 376
Missing \endgroup inserted: 1068
Missing \right. inserted: 1068
Missing { inserted: 406, 478, 1130
Missing } inserted: 1068, 1130
Missing 'to' inserted: 1085
Missing 'to'...: 1228
Missing $$ inserted: 1210
Missing character: 584, 1396, 1400
Missing control...: 1218
Missing delimiter...: 1164
Missing font identifier: 580
Missing number...: 418, 449
mkern: 208, 1049, 1060, 1061, 1062
\mkern primitive: 1061
ml_field: 212, 213, 218
mlist: 729, 763
mlist_penalties: 722, 723, 729, 757, 1197, 1199, 1202
mlist_to_hlist: 696, 722, 723, 728, 729, 737, 757, 763, 1197, 1199, 1202
mltex_enabled_p: 238, 537, 623, 1340, 1394, 1395, 1396, 1397, 1404
mltex_p: 238, 1225, 1393, 1394, 1403, 1404
mm: 461
mmode: 211, 212, 213, 218, 504, 721, 778, 779, 803, 815, 1048, 1049, 1051, 1059, 1060, 1076, 1083, 1095, 1100, 1112, 1113, 1115, 1119, 1123, 1133, 1139, 1143, 1148, 1153, 1157, 1161, 1165, 1167, 1170, 1174, 1178, 1183, 1193, 1196, 1197, 1445, 1448, 1461
mode: 211, 212, 213, 215, 216, 299, 421, 425, 427, 504, 721, 778, 779, 788, 789, 790, 799, 802, 807, 810, 811, 812, 815, 1028, 1032, 1033, 1037, 1038, 1052, 1054, 1059, 1079, 1081, 1083, 1086, 1089, 1094, 1096, 1097, 1098, 1099, 1102, 1106, 1108, 1113, 1120, 1122, 1123, 1139, 1141, 1148, 1170, 1197, 1199, 1203, 1246, 1373, 1374, 1380, 1451, 1454
mode_field: 212, 213, 218, 425, 803, 1247
mode_line: 212, 213, 215, 216, 304, 807, 818, 1028
Modified for handling DBCS characters: 343, 354, 355, 356, 377

month: 236, 241, 539, 620, 1331
\month primitive: 238
month_code: 236, 237, 238
months: 537, 539
more_name: 515, 519, 528, 529, 534, 1382
more_param: 1486, 1487
\moveleft primitive: 1074
move_past: 622, 625, 628, 632, 634, 637
\moveright primitive: 1074
movement: 610, 612, 619
movement_node_size: 608, 610, 618
mskip: 208, 1049, 1060, 1061, 1062
\mskip primitive: 1061
mskip_code: 1061, 1063
mstate: 610, 614, 615
mtype: 4
mu: 450, 451, 452, 456, 458, 464, 465
mu: 459
mu_error: 411, 432, 452, 458, 464
mu_glue: 149, 155, 191, 427, 720, 735, 1061, 1063, 1064
mu_mult: 719, 720
mu_skip: 224, 430
\muskip primitive: 414
mu_skip_base: 224, 227, 229, 1227, 1240
\muskipdef primitive: 1225
mu_skip_def_code: 1225, 1226, 1227
mu_val: 413, 414, 416, 427, 430, 432, 433, 452, 454, 458, 464, 468, 1063, 1231, 1239, 1240
mult_and_add: 105
mult_integers: 105, 1243, 1511
multiply: 209, 265, 266, 1213, 1238, 1239, 1243
\multiply primitive: 265
Must increase the x: 1306
must_quote: 520, 521
n: 47, 65, 66, 67, 69, 91, 94, 105, 106, 107, 152, 154, 174, 182, 225, 237, 247, 252, 292, 315, 392, 485, 501, 521, 522, 526, 581, 709, 719, 720, 794, 803, 909, 937, 947, 980, 995, 996, 997, 1015, 1082, 1122, 1141, 1214, 1278, 1341, 1422, 1434, 1524
name: 300, 302, 303, 304, 307, 311, 313, 314, 323, 328, 329, 331, 337, 360, 393, 486, 540
name_field: 84, 300, 302
name_in_progress: 381, 528, 529, 530, 531, 1261
name_length: 26, 51, 522, 526, 528
name_of_file: 26, 51, 522, 526, 527, 528, 533, 537, 540, 1278, 1311, 1377
name_too_long: 563, 564, 566
nameseq_code: 471, 472, 474, 475
natural: 647, 708, 718, 723, 730, 738, 740, 741, 751, 757, 759, 762, 799, 802, 809, 980, 1024, 1103, 1128, 1197, 1202, 1207
nd: 543, 544, 563, 568, 569, 572
ne: 543, 544, 563, 568, 569, 572
neg_trie_op_size: 11, 946, 947
negate: 16, 65, 103, 105, 106, 107, 433, 434, 443, 451, 464, 778, 1422, 1431, 1440
negative: 106, 416, 433, 443, 444, 451, 464
negative_wchar_offset: 1421, 1422, 1433
negative_wsym_offset: 1421, 1435
nest: 212, 213, 216, 217, 218, 219, 416, 425, 778, 803, 998, 1247, 1335, 1451, 1454
nest_ptr: 213, 215, 216, 217, 218, 425, 778, 803, 998, 1020, 1026, 1094, 1103, 1148, 1203, 1247, 1451, 1454
nest_size: 32, 213, 216, 218, 416, 1247, 1335, 1337
new_cface: 1471, 1482
new_character: 585, 758, 918, 1120, 1126, 1127
new_choice: 692, 1175
new_delta_from_break_width: 847
new_delta_to_break_width: 846
new_disc: 145, 1038, 1120
new_font: 1259, 1260
new_glue: 153, 154, 718, 769, 789, 796, 798, 812, 1044, 1046, 1057, 1063, 1174, 1457, 1467
new_graf: 1093, 1094
new_hlist: 728, 730, 746, 751, 752, 753, 757, 759, 765, 770
new_hyph_exceptions: 937, 1255
new_interaction: 1267, 1268
new_kern: 156, 708, 718, 738, 741, 742, 750, 754, 756, 758, 762, 913, 1043, 1064, 1115, 1116, 1128, 1207
new_lig_item: 144, 914, 1043
new_ligature: 144, 913, 1038
new_line: 303, 331, 343, 344, 345, 347, 486, 540
new_line_char: 59, 236, 244
\newlinechar primitive: 238
new_line_char_code: 236, 237, 238
new_math: 147, 1199
new_noad: 689, 723, 745, 756, 1079, 1096, 1153, 1158, 1161, 1171, 1180, 1194
new_null_box: 136, 709, 712, 716, 723, 750, 753, 782, 796, 812, 1021, 1057, 1094, 1096
new_param_glue: 152, 154, 682, 781, 819, 889, 890, 1044, 1046, 1094, 1206, 1208, 1209
new_patterns: 963, 1255
new_penalty: 158, 770, 819, 893, 1057, 1106, 1206, 1208, 1209, 1451, 1452, 1453, 1457, 1458, 1459, 1463
new_rule: 139, 466, 669, 707
new_save_level: 274, 648, 777, 788, 794, 1028, 1066, 1102, 1120, 1122, 1139

new_skip_param: 154, 682, 972, 1004
new_spec: 151, 154, 433, 465, 829, 979, 1007, 1045, 1046, 1242, 1243, 1507, 1514
new_string: 54, 57, 58, 468, 473, 620, 1260, 1282, 1331, 1371, 1373, 1482
new_style: 691, 1174
new_trie_op: 946, 947, 948, 968
new_whatsit: 1352, 1353, 1357, 1379, 1380, 1406
new_write_whatsit: 1353, 1354, 1355, 1356
next: 256, 259, 260, 1311, 1335
next_break: 880, 881
next_char: 548, 744, 756, 912, 1042
next_is_a_char: 1033, 1453, 1459
next_p: 622, 625, 629, 632, 633, 634, 636, 638
nh: 543, 544, 563, 568, 569, 572
ni: 543, 544, 563, 568, 569, 572
nil: 16
nine_bits: 551, 552, 1326, 1340
nk: 543, 544, 563, 568, 569, 576
nl: 59, 543, 544, 548, 563, 568, 569, 572, 576, 579
nn: 311, 312
`No pages of output`: 645
no_align: 208, 265, 266, 788, 1129
`\noalign` primitive: 265
no_align_error: 1129, 1132
no_align_group: 269, 771, 788, 1136
no_boundary: 208, 265, 266, 1033, 1048, 1093, 1460
`\noboundary` primitive: 265
no_break_yet: 832, 839, 840
no_expand: 210, 265, 266, 366, 370
`\noexpand` primitive: 265
no_expand_flag: 358, 509
`\noindent` primitive: 1091
no_limits: 685, 1159, 1160
`\nolimits` primitive: 1159
no_new_control_sequence: 256, 257, 259, 264, 365, 377, 1339, 1524
no_print: 54, 57, 58, 75, 98
no_shrink_error_yet: 828, 829, 830
no_tag: 547, 572
noad_size: 684, 689, 701, 756, 764, 1189, 1190
node_list_display: 180, 184, 188, 190, 195, 197
node_r_stays_active: 833, 854, 857
node_size: 124, 126, 127, 128, 130, 164, 169, 1314, 1315
nom: 563, 564, 566, 579
non_address: 552, 579, 912, 919, 1037, 1340
non_char: 551, 552, 579, 900, 901, 904, 911, 912, 913, 914, 918, 919, 920, 1035, 1037, 1038, 1042, 1043, 1326, 1340, 1460
non_discardable: 148, 882
non_math: 1049, 1066, 1147
non_script: 208, 265, 266, 1049, 1174
`\nonscript` primitive: 265, 735
none_seen: 614, 615
`NONEXISTENT`: 262
`Nonletter`: 965
nonnegative_integer: 69, 101, 107
nonstop_mode: 73, 86, 360, 363, 487, 1265, 1266
`\nonstopmode` primitive: 1265
nop: 586, 588, 589, 591, 593
noreturn: 81, 93, 94, 95
norm_min: 1094, 1203, 1379, 1380
normal: 135, 136, 149, 150, 153, 155, 156, 164, 177, 186, 189, 191, 305, 331, 336, 372, 442, 451, 474, 476, 483, 485, 488, 492, 493, 510, 622, 628, 632, 637, 653, 660, 661, 662, 663, 667, 668, 669, 670, 675, 676, 677, 679, 680, 681, 685, 689, 699, 719, 735, 752, 780, 804, 813, 814, 828, 829, 899, 900, 902, 979, 991, 1007, 1012, 1159, 1166, 1168, 1184, 1204, 1222, 1223, 1224, 1242, 1450
normal_paragraph: 777, 788, 790, 1028, 1073, 1086, 1097, 1099, 1102, 1170
normalize_selector: 78, 92, 93, 94, 95, 866
`Not a letter`: 940
not_found: 15, 45, 46, 451, 458, 563, 573, 610, 614, 615, 898, 933, 934, 937, 944, 956, 958, 973, 975, 976, 1141, 1149, 1368
`notexpanded:`: 258
np: 543, 544, 563, 568, 569, 578, 579
nucleus: 684, 685, 686, 689, 690, 693, 699, 701, 723, 728, 737, 738, 739, 740, 741, 744, 745, 752, 753, 755, 756, 757, 758, 1079, 1096, 1153, 1154, 1158, 1161, 1166, 1168, 1171, 1189, 1194
null: 115, 116, 118, 120, 122, 123, 125, 126, 135, 136, 144, 145, 149, 150, 151, 152, 153, 154, 164, 168, 169, 175, 176, 182, 200, 201, 202, 204, 210, 212, 215, 218, 219, 222, 223, 232, 233, 275, 292, 295, 306, 307, 312, 314, 325, 331, 357, 358, 374, 377, 385, 386, 389, 393, 394, 395, 400, 403, 410, 413, 423, 426, 455, 467, 469, 476, 481, 485, 492, 493, 500, 508, 511, 552, 579, 581, 585, 609, 614, 618, 622, 626, 632, 635, 651, 652, 654, 658, 661, 667, 669, 671, 676, 679, 684, 688, 692, 695, 718, 721, 722, 723, 724, 729, 734, 735, 755, 757, 758, 759, 763, 764, 769, 770, 774, 777, 779, 780, 786, 787, 792, 793, 794, 795, 797, 799, 800, 802, 804, 807, 808, 809, 810, 815, 824, 832, 840, 843, 849, 850, 851, 853, 859, 860, 861, 862, 866, 867, 868, 870, 872, 875, 880, 881, 882, 884, 885, 886, 887, 888, 890, 891, 892, 897, 899, 901, 906, 909, 910, 911, 913, 914, 916, 917, 918, 919, 920, 921, 931, 935, 938, 971, 972, 973, 975, 976, 980, 981, 982, 984, 994, 995, 996, 997, 1001, 1002, 1003,

1012, 1013, 1014, 1015, 1017, 1018, 1019, 1020, 1021, 1023, 1024, 1025, 1026, 1029, 1030, 1031, 1033, 1035, 1038, 1039, 1040, 1043, 1045, 1046, 1073, 1077, 1078, 1079, 1082, 1083, 1084, 1086, 1090, 1094, 1108, 1113, 1124, 1126, 1127, 1134, 1139, 1142, 1148, 1149, 1152, 1170, 1177, 1179, 1184, 1187, 1188, 1189, 1197, 1199, 1202, 1205, 1208, 1209, 1229, 1230, 1250, 1251, 1286, 1291, 1299, 1311, 1314, 1315, 1338, 1340, 1342, 1356, 1357, 1371, 1372, 1378, 1406, 1449, 1450, 1451, 1454, 1455, 1456, 1460, 1462, 1463
null delimiter: 240, 1068
null_cface: 1472, 1480, 1482, 1483, 1484, 1485, 1495, 1496, 1497, 1503, 1519, 1537, 1550
null_cfont: 232, 620, 666, 867, 1342, 1503, 1507, 1509, 1532, 1535, 1537, 1558
null_character: 558, 559, 725, 726, 1397
null_code: 22, 232, 1373
null_cs: 222, 262, 263, 354, 377, 1260, 1482
null_delimiter: 687, 688, 1184
null_delimiter_space: 247, 709
`\nulldelimiterspace` primitive: 248
null_delimiter_space_code: 247, 248
null_flag: 138, 139, 466, 656, 782, 796, 804
null_font: 232, 556, 563, 580, 620, 666, 709, 710, 725, 867, 1260, 1325, 1326, 1340, 1342, 1535
`\nullfont` primitive: 556
null_list: 14, 162, 383, 783
num: 453, 461, 588, 590, 593
num_style: 705, 747
`Number too big`: 448
`\number` primitive: 471
number_code: 471, 472, 473, 474, 475
numerator: 686, 693, 700, 701, 747, 1184, 1188
num1: 703, 747
num2: 703, 747
num3: 703, 747
nw: 543, 544, 563, 568, 569, 572
nx_plus_y: 105, 458, 719, 1243
o: 264, 610, 652, 671, 794, 803
octal_token: 441, 447
odd: 62, 100, 193, 507, 761, 901, 905, 911, 912, 916, 917, 1214, 1221
off_save: 1066, 1067, 1097, 1098, 1133, 1134, 1143, 1195, 1196
offset: 473, 1437
`OK`: 1301
OK_so_far: 443, 448
OK_to_interrupt: 88, 96, 97, 98, 327, 1034
old_l: 832, 838, 853
old_mode: 1373, 1374
old_rover: 131
old_setting: 245, 246, 311, 312, 468, 473, 537, 620, 641, 1260, 1282, 1371, 1373, 1376, 1377, 1482, 1564, 1567
omit: 208, 265, 266, 791, 792, 1129
`\omit` primitive: 265
omit_error: 1129, 1132
omit_template: 162, 792, 793
`Only one # is allowed...`: 787
op_byte: 548, 560, 744, 756, 912, 914, 1043
op_noad: 685, 693, 699, 701, 729, 731, 736, 752, 764, 1159, 1160, 1162
op_start: 923, 924, 927, 948, 1328
open_area: 1344, 1354, 1359, 1377
open_ext: 1344, 1354, 1359, 1377
open_fmt_file: 527, 1340
`\openin` primitive: 1275
open_input: 540, 1278
open_log_file: 78, 92, 360, 474, 535, 537, 538, 540, 1260, 1338, 1373, 1482
open_name: 1344, 1354, 1359, 1377
open_noad: 685, 693, 699, 701, 731, 736, 764, 765, 1159, 1160
open_node: 1344, 1347, 1349, 1351, 1359, 1360, 1361, 1376
open_node_size: 1344, 1354, 1360, 1361
open_or_close_in: 1277, 1278
`\openout` primitive: 1347
open_parens: 304, 331, 362, 540, 1338
`\or` primitive: 494
or_code: 492, 494, 495, 503, 512
ord: 20
ord_noad: 684, 685, 689, 690, 693, 699, 701, 731, 732, 736, 755, 756, 764, 767, 768, 1078, 1158, 1159, 1160, 1189
order: 177
oriental characters: 134, 588
orig_char_info: 557, 573, 576, 579, 585, 623, 711, 725, 743, 752, 1396, 1397
orig_char_info_end: 557
other_A_token: 448
other_char: 207, 232, 289, 291, 294, 298, 347, 448, 467, 529, 938, 964, 1033, 1093, 1127, 1154, 1157, 1163, 1413, 1452, 1459, 1460, 1463, 1464
other_token: 289, 408, 441, 444, 448, 467, 506, 1068, 1224
othercases: 10
others: 10
`Ouch...clobbered`: 1335
out_param: 207, 289, 291, 294, 357
out_param_token: 289, 482
out_what: 1369, 1370, 1376, 1378
`\outer` primitive: 1211

outer_call: 210, 275, 339, 351, 353, 354, 357, 366, 390, 394, 399, 783, 1155, 1298, 1372
outer_doing_leaders: 622, 631, 632, 640
outer_tail: 1448, 1451, 1454
`Output loop...`: 1027
`Output routine didn't use...`: 1031
`Output written on x`: 645
`\output` primitive: 230
output_active: 424, 666, 678, 989, 992, 993, 997, 1008, 1028, 1029
output_comment: 620, 1384
output_file_name: 535, 536, 645
output_group: 269, 1028, 1103
output_penalty: 236
`\outputpenalty` primitive: 238
output_penalty_code: 236, 237, 238, 1016
output_routine: 230, 1015, 1028
output_routine_loc: 230, 231, 232, 307, 323, 1229
output_text: 307, 314, 323, 1028, 1029
`\over` primitive: 1181
over_code: 1181, 1182, 1185
over_noad: 690, 693, 699, 701, 736, 764, 1159
`\overwithdelims` primitive: 1181
overbar: 708, 737, 740
overflow: 35, 42, 43, 94, 120, 125, 216, 260, 273, 274, 321, 328, 369, 377, 393, 520, 583, 943, 947, 957, 967, 1336
overflow in arithmetic: 9, 104
`Overfull \hbox...`: 669
`Overfull \vbox...`: 680
overfull boxes: 857
overfull_rule: 247, 669, 803, 807
`\overfullrule` primitive: 248
overfull_rule_code: 247, 248
`\overline` primitive: 1159
p: 120, 123, 125, 130, 131, 136, 139, 144, 145, 147, 151, 152, 153, 154, 156, 158, 167, 172, 174, 176, 178, 182, 198, 200, 201, 202, 204, 218, 259, 262, 263, 276, 277, 278, 279, 281, 284, 292, 295, 306, 315, 323, 325, 336, 369, 392, 410, 416, 453, 467, 468, 476, 485, 500, 501, 585, 610, 618, 622, 632, 641, 652, 671, 682, 689, 691, 692, 694, 695, 707, 708, 712, 714, 718, 719, 720, 723, 729, 738, 741, 746, 752, 755, 759, 775, 777, 790, 794, 802, 803, 829, 909, 937, 951, 952, 956, 960, 962, 963, 969, 971, 973, 996, 997, 1015, 1067, 1071, 1078, 1082, 1089, 1096, 1104, 1108, 1113, 1116, 1122, 1126, 1141, 1154, 1158, 1163, 1177, 1179, 1187, 1194, 1197, 1214, 1239, 1247, 1291, 1296, 1305, 1306, 1351, 1352, 1358, 1371, 1373, 1376, 1525
pack_begin_line: 664, 665, 666, 678, 807, 818
pack_buffered_name: 526, 527
pack_cur_name: 532, 533, 540, 1278, 1377
pack_file_name: 522, 532, 566
pack_job_name: 532, 535, 537, 1331
pack_lig: 1038
package: 1088, 1089, 1462
packed_ASCII_code: 38, 39, 950, 1313, 1335, 1340
page: 304
page_contents: 215, 424, 983, 989, 990, 994, 1003, 1004, 1011
page_depth: 215, 985, 990, 994, 1005, 1006, 1007, 1011, 1013
`\pagedepth` primitive: 986
`\pagefilstretch` primitive: 986
`\pagefillstretch` primitive: 986
`\pagefilllstretch` primitive: 986
page_goal: 983, 985, 989, 990, 1008, 1009, 1010, 1011, 1012, 1013
`\pagegoal` primitive: 986
page_head: 162, 215, 983, 989, 991, 994, 1017, 1020, 1026, 1029, 1057, 1311
page_ins_head: 162, 984, 989, 1008, 1011, 1021, 1022, 1023
page_ins_node_size: 984, 1012, 1022
page_loc: 641, 643
page_max_depth: 215, 983, 985, 990, 994, 1006, 1020
page_shrink: 985, 988, 1007, 1010, 1011, 1012
`\pageshrink` primitive: 986
page_so_far: 424, 985, 988, 990, 1007, 1010, 1012, 1248
page_stack: 304
`\pagestretch` primitive: 986
page_tail: 215, 983, 989, 994, 1001, 1003, 1020, 1026, 1029, 1057, 1311
page_total: 985, 988, 1005, 1006, 1007, 1010, 1011, 1013
`\pagetotal` primitive: 986
panicking: 165, 166, 1034, 1342
`\par` primitive: 334
par_end: 207, 334, 335, 1049, 1097
par_fill_skip: 224, 819
`\parfillskip` primitive: 226
par_fill_skip_code: 224, 225, 226, 819
par_indent: 247, 1094, 1096
`\parindent` primitive: 248
par_indent_code: 247, 248
par_loc: 333, 334, 351, 1316, 1317
`\parshape` primitive: 265
par_shape_loc: 230, 232, 233, 1073, 1251
par_shape_ptr: 230, 232, 233, 426, 817, 850, 851, 853, 892, 1073, 1152, 1252
par_skip: 224, 1094

\parskip primitive: 226
par_skip_code: 224, 225, 226, 1094
par_token: 333, 334, 339, 395, 398, 402, 1098, 1317
Paragraph ended before...: 399
param: 545, 550, 561
param_base: 553, 561, 569, 577, 578, 579, 581, 583, 703, 704, 1045, 1325, 1326, 1340
param_end: 561
param_ptr: 308, 323, 324, 331, 393
param_size: 32, 308, 393, 1335, 1337
param_stack: 307, 308, 324, 359, 391, 392, 393, 1335
param_start: 307, 323, 324, 359
parameter: 307, 314, 359
parameters for symbols: 703, 704
Parameters...consecutively: 479
parse_first_line_p: 32, 61, 539
Pascal-H: 3, 9, 10
Pascal: 1, 10, 696, 767
pass_number: 824, 848, 867
pass_text: 369, 497, 503, 512, 513
passive: 824, 848, 849, 867, 868
passive_node_size: 824, 848, 868
Patterns can be...: 1255
\patterns primitive: 1253
pause_for_instructions: 96, 98
pausing: 236, 363
\pausing primitive: 238
pausing_code: 236, 237, 238
pc: 186
pc: 461
pen: 729, 764, 770, 880, 893
penalties: 1105
penalties: 729, 770
penalty: 157, 158, 194, 427, 819, 869, 976, 999, 1003, 1013, 1014, 1016
\penalty primitive: 265
penalty_node: 157, 158, 183, 202, 206, 427, 733, 764, 770, 819, 820, 840, 859, 869, 882, 902, 971, 976, 999, 1003, 1013, 1014, 1016, 1110
pg_field: 212, 213, 218, 219, 425, 1247
pi: 832, 834, 854, 859, 862, 973, 975, 976, 977, 997, 1003, 1008, 1009
plain: 524, 527, 1334
Plass, Michael Frederick: 2, 816
Please type...: 360, 533
Please use \mathaccent...: 1169
PLtoTF: 564
plus: 465
point_token: 441, 443, 451, 455
pointer: 115, 116, 118, 120, 123, 124, 125, 130, 131, 136, 139, 144, 145, 147, 151, 152, 153, 154, 156, 158, 165, 167, 172, 198, 200, 201, 202, 204, 212, 218, 252, 256, 259, 263, 275, 276, 277, 278, 279, 281, 284, 295, 297, 305, 306, 308, 323, 325, 333, 336, 369, 385, 391, 392, 410, 453, 464, 466, 467, 468, 476, 485, 492, 500, 501, 552, 563, 585, 595, 608, 610, 618, 622, 632, 641, 650, 652, 671, 682, 689, 691, 692, 694, 695, 707, 708, 709, 712, 714, 718, 719, 720, 722, 723, 725, 729, 737, 738, 739, 740, 741, 746, 752, 755, 759, 765, 773, 775, 777, 790, 794, 802, 803, 817, 824, 829, 831, 832, 833, 836, 865, 875, 880, 895, 903, 904, 909, 910, 915, 929, 937, 971, 973, 980, 983, 985, 996, 997, 1015, 1035, 1046, 1067, 1071, 1077, 1078, 1082, 1089, 1096, 1104, 1108, 1113, 1116, 1122, 1126, 1141, 1154, 1158, 1163, 1177, 1179, 1187, 1194, 1197, 1201, 1214, 1239, 1260, 1291, 1296, 1305, 1306, 1348, 1351, 1352, 1358, 1371, 1373, 1376, 1406, 1448, 1482, 1502, 1524
Poirot, Hercule: 1286
pool_file: 47, 50, 51, 52, 53
pool_free: 32, 1313, 1335
pool_name: 11, 51, 52, 53, 1311
pool_pointer: 38, 39, 45, 46, 59, 60, 69, 70, 264, 410, 467, 468, 473, 516, 520, 521, 522, 528, 605, 641, 932, 937, 1260, 1313, 1335, 1371, 1382, 1384, 1525
pool_ptr: 38, 39, 41, 42, 43, 44, 47, 52, 58, 70, 198, 260, 467, 468, 473, 519, 520, 528, 620, 1312, 1313, 1335, 1337, 1342, 1371, 1373, 1418, 1510
pool_size: 32, 42, 52, 58, 198, 528, 1313, 1335, 1337, 1342, 1371, 1577, 1579
pop: 587, 588, 589, 593, 604, 611, 645, 1402
pop_alignment: 775, 803
pop_input: 322, 324, 329
pop_lig_stack: 913, 914
pop_nest: 217, 799, 802, 815, 819, 1029, 1089, 1099, 1103, 1122, 1148, 1171, 1187, 1209
positive: 107
post: 586, 588, 589, 593, 594, 645
post_break: 145, 175, 195, 202, 206, 843, 861, 885, 887, 919, 1122
post_disc_break: 880, 884, 887
post_display_penalty: 236, 1208, 1209
\postdisplaypenalty primitive: 238
post_display_penalty_code: 236, 237, 238
post_line_break: 879, 880
post_post: 588, 589, 593, 594, 645
pre: 586, 588, 589, 620
pre_break: 145, 175, 195, 202, 206, 861, 872, 885, 888, 918, 1120, 1122
pre_display_penalty: 236, 1206, 1209
\predisplaypenalty primitive: 238

pre_display_penalty_code: 236, 237, 238
pre_display_size: 247, 1141, 1148, 1151, 1206
\predisplaysize primitive: 248
pre_display_size_code: 247, 248, 1148
pre_glue_char_ptr: 1448, 1449, 1451, 1455, 1463
pre_undet_glue_ptr: 1448, 1449, 1451, 1454, 1456, 1463, 1464
preamble: 771, 777
preamble: 773, 774, 775, 780, 789, 804, 807
preamble of DVI file: 620
precedes_break: 148, 871, 976, 1003
prefix: 209, 1211, 1212, 1213, 1214
prefixed_command: 1213, 1214, 1273
prepare_mag: 288, 460, 620, 645, 1336
pretolerance: 236, 831, 866
\pretolerance primitive: 238
pretolerance_code: 236, 237, 238
prev_break: 824, 848, 849, 880, 881
prev_depth: 212, 213, 215, 421, 682, 778, 789, 790, 1028, 1059, 1086, 1102, 1170, 1209, 1245, 1246
\prevdepth primitive: 419
prev_dp: 973, 975, 976, 977, 979
prev_graf: 212, 213, 215, 216, 425, 817, 819, 867, 880, 893, 1094, 1152, 1203, 1245
\prevgraf primitive: 265
prev_main_cf: 1448, 1451, 1455, 1463, 1464
prev_p: 865, 866, 869, 870, 871, 872, 971, 972, 973, 976, 1015, 1017, 1020, 1025
prev_prev_r: 833, 835, 846, 847, 863
prev_r: 832, 833, 835, 846, 847, 848, 854, 857, 863
prev_s: 865, 897, 899
primitive: 226, 230, 238, 248, 264, 265, 266, 298, 334, 379, 387, 414, 419, 471, 490, 494, 556, 783, 986, 1055, 1061, 1074, 1091, 1110, 1117, 1144, 1159, 1172, 1181, 1191, 1211, 1222, 1225, 1233, 1253, 1257, 1265, 1275, 1280, 1289, 1294, 1334, 1335, 1347, 1414, 1425, 1429, 1442, 1465, 1469, 1475, 1520, 1533, 1542, 1548, 1556, 1561, 1565
print: 54, 59, 60, 62, 63, 68, 70, 71, 73, 85, 86, 89, 91, 94, 95, 175, 177, 178, 182, 183, 184, 185, 186, 187, 188, 190, 191, 192, 193, 195, 211, 218, 219, 225, 233, 234, 237, 247, 251, 262, 263, 284, 288, 294, 298, 299, 317, 318, 323, 336, 338, 339, 363, 376, 398, 399, 401, 403, 431, 457, 459, 462, 468, 475, 505, 512, 521, 533, 537, 539, 564, 570, 582, 584, 620, 641, 642, 645, 663, 666, 669, 677, 678, 680, 695, 697, 700, 726, 779, 849, 859, 939, 981, 988, 989, 990, 1009, 1014, 1018, 1027, 1067, 1098, 1135, 1169, 1216, 1227, 1235, 1240, 1260, 1262, 1264, 1298, 1299, 1301, 1312, 1314, 1321, 1323, 1325, 1327, 1331, 1337, 1338, 1341, 1342, 1349, 1359, 1373, 1377, 1387, 1396, 1400, 1401, 1416, 1428, 1437, 1444, 1477, 1482, 1484, 1488, 1489, 1490, 1491, 1493, 1515, 1546, 1547, 1568, 1569, 1570, 1571, 1576, 1580
print_ASCII: 68, 174, 176, 298, 584, 694, 726, 1227, 1396, 1400, 1401
print_c_string: 533
print_char: 58, 59, 60, 64, 65, 66, 67, 69, 70, 82, 91, 94, 95, 103, 114, 171, 172, 174, 175, 176, 177, 178, 184, 186, 187, 188, 189, 190, 191, 193, 218, 223, 229, 233, 234, 235, 242, 251, 252, 255, 262, 284, 285, 294, 296, 299, 306, 313, 317, 362, 468, 475, 512, 521, 539, 540, 564, 584, 620, 641, 642, 694, 726, 849, 859, 936, 1009, 1014, 1068, 1072, 1215, 1216, 1227, 1283, 1297, 1299, 1314, 1325, 1331, 1336, 1338, 1342, 1343, 1358, 1359, 1373, 1396, 1400, 1401, 1422, 1440, 1576
print_chinese_int: 475, 1422
print_cjk_int: 1434, 1435, 1436
print_cmd_chr: 223, 233, 266, 296, 298, 299, 323, 336, 421, 431, 506, 513, 1052, 1069, 1131, 1215, 1216, 1240, 1338, 1342
print_cs: 262, 293, 314, 404
print_csnames: 1322, 1385
print_current_string: 70, 182, 695
print_dbchar: 1422
print_delimiter: 694, 699, 700
print_err: 72, 73, 93, 94, 95, 98, 288, 336, 338, 346, 373, 376, 398, 399, 401, 406, 411, 418, 421, 431, 436, 437, 438, 439, 440, 445, 448, 449, 457, 459, 462, 463, 478, 479, 482, 489, 503, 506, 513, 533, 564, 580, 582, 644, 726, 779, 786, 787, 795, 829, 939, 940, 963, 964, 965, 966, 979, 981, 996, 1007, 1012, 1018, 1027, 1030, 1031, 1050, 1052, 1067, 1069, 1071, 1072, 1081, 1085, 1087, 1098, 1102, 1113, 1123, 1124, 1130, 1131, 1132, 1135, 1138, 1154, 1157, 1162, 1164, 1169, 1180, 1186, 1195, 1198, 1200, 1210, 1215, 1216, 1218, 1228, 1235, 1239, 1240, 1244, 1246, 1247, 1255, 1261, 1262, 1286, 1301, 1307, 1375, 1388, 1416, 1419, 1427, 1428, 1437, 1444, 1445, 1461, 1477, 1484, 1485, 1488, 1489, 1490, 1491, 1492, 1493, 1497, 1510, 1511, 1514, 1526, 1528, 1529, 1535, 1544, 1546, 1547, 1550, 1558
print_esc: 63, 86, 176, 184, 187, 188, 189, 190, 191, 192, 194, 195, 196, 197, 225, 227, 229, 231, 233, 234, 235, 237, 239, 242, 247, 249, 251, 262, 263, 266, 267, 292, 293, 294, 323, 335, 376, 380, 388, 415, 420, 431, 472, 489, 491, 495, 503, 582, 694, 697, 698, 699, 700, 702, 779, 784, 795, 859, 939, 963, 964, 981, 987, 989, 1012, 1018, 1031, 1056, 1062, 1068, 1072, 1075, 1092, 1098, 1102, 1111, 1118, 1123, 1132, 1135, 1138, 1146,

1160, 1169, 1182, 1192, 1195, 1212, 1216, 1223, 1226, 1234, 1244, 1247, 1254, 1258, 1266, 1276, 1281, 1290, 1295, 1298, 1325, 1338, 1349, 1358, 1359, 1426, 1443, 1444, 1446, 1466, 1470, 1476, 1521, 1543, 1549, 1557, 1562, 1566
print_fam_and_char: 694, 695, 699
print_file_line: 73, 1387
print_file_name: 521, 533, 564, 645, 1325, 1336, 1359, 1377
print_fixword: 1440, 1569, 1576
print_font_and_char: 176, 183, 193
print_glue: 177, 178, 185, 186
print_hex: 67, 694, 1226, 1446
print_in_mode: 211, 1052
print_int: 65, 91, 94, 103, 114, 168, 169, 170, 171, 172, 185, 188, 194, 195, 218, 219, 227, 229, 231, 233, 234, 235, 239, 242, 249, 251, 255, 285, 288, 313, 336, 403, 468, 475, 512, 539, 564, 582, 620, 641, 642, 645, 663, 666, 670, 677, 678, 681, 694, 726, 849, 859, 936, 989, 1009, 1012, 1014, 1027, 1031, 1102, 1235, 1299, 1312, 1314, 1321, 1323, 1327, 1331, 1338, 1342, 1358, 1359, 1377, 1387, 1416, 1428, 1440, 1488, 1489, 1490, 1491, 1546, 1547, 1568, 1569, 1570, 1571, 1576, 1580
print_length_param: 247, 249, 251
print_ln: 57, 58, 59, 61, 62, 71, 86, 89, 90, 114, 182, 198, 218, 236, 245, 296, 306, 314, 317, 330, 360, 363, 404, 487, 533, 537, 540, 641, 642, 663, 666, 669, 670, 677, 678, 680, 681, 695, 989, 1268, 1283, 1312, 1314, 1321, 1323, 1327, 1336, 1343, 1373, 1377, 1568, 1569, 1570, 1571, 1576, 1580
print_locs: 167
print_mark: 176, 196, 1359
print_meaning: 296, 475, 1297
print_medium_chinese_int: 1422, 1424
print_mode: 211, 218, 299
print_nl: 62, 73, 82, 85, 90, 168, 169, 170, 171, 172, 218, 219, 245, 255, 285, 288, 299, 306, 311, 313, 314, 323, 360, 403, 533, 537, 584, 641, 642, 644, 645, 663, 669, 670, 677, 680, 681, 849, 859, 860, 866, 936, 989, 990, 995, 1009, 1014, 1124, 1227, 1297, 1299, 1300, 1325, 1327, 1331, 1336, 1338, 1341, 1373, 1377, 1387, 1396, 1400, 1401
print_param: 237, 239, 242, 1546
print_plus: 988
print_plus_end: 988
print_quoted: 521
print_roman_int: 69, 475
print_rule_dimen: 176, 187
print_scaled: 103, 114, 176, 177, 178, 184, 188, 191, 192, 219, 251, 468, 475, 564, 669, 680, 700, 988, 989, 990, 1009, 1014, 1262, 1264, 1325, 1342, 1515, 1568, 1570, 1580
print_size: 702, 726, 1234
print_skip_param: 189, 225, 227, 229
print_small_chinese_int: 1422, 1423, 1424
print_spec: 178, 188, 189, 190, 229, 468
print_style: 693, 697, 1173
print_subsidiary_data: 695, 699, 700
print_the_digs: 64, 65, 67
print_totals: 218, 988, 989, 1009
print_two: 66, 539, 620
print_wchar: 58, 174, 176, 235, 294, 298, 468, 475, 1422, 1423, 1424, 1434, 1438
print_word: 114, 1342
print_write_whatsit: 1358, 1359
printed_node: 824, 859, 860, 861, 867
privileged: 1054, 1057, 1133, 1143
prompt_file_name: 533, 535, 538, 540, 1331, 1377
prompt_file_name_help_msg: 533
prompt_input: 71, 83, 87, 360, 363, 487, 533
prune_movements: 618, 622, 632
prune_page_top: 971, 980, 1024
pseudo: 54, 57, 58, 59, 316
pstack: 391, 393, 399, 403
pt: 456
punct_noad: 685, 693, 699, 701, 731, 755, 764, 1159, 1160
push: 587, 588, 589, 593, 595, 604, 611, 619, 622, 632, 1402
push_alignment: 775, 777
push_input: 321, 323, 325, 328
push_math: 1139, 1142, 1148, 1156, 1175, 1177, 1194
push_nest: 216, 777, 789, 790, 1028, 1086, 1094, 1102, 1120, 1122, 1139, 1170, 1203
put: 26, 29
put_byte: 1385
put_rule: 588, 589, 636
PUTeX_banner: 2
putex_banner: 61
PUTeX_banner_k: 2
PUTeX_version_string: 2
put1: 588
put2: 588
put3: 588
put4: 588
\PUXcatcode primitive: 1233
pux_cat_code_base: 230, 235, 417, 1233, 1234, 1235, 1236
pux_cespace_code: 1465
pux_cface_def: 209, 1213, 1469, 1470, 1471
\toksdef primitive: 1225
pux_char_def_code: 1225, 1227

pux_char_given: 209, 416, 938, 1033, 1093, 1154, 1227, 1445, 1446, 1452, 1459, 1460, 1463, 1464
pux_char_num: 209, 938, 1033, 1093, 1154, 1442, 1443, 1445, 1452, 1453, 1460, 1463, 1464
pux_charset: 236, 620, 1483
pux_charset_code: 236, 237, 1542, 1543, 1546
pux_CJKinput: 236, 240, 343
pux_CJKinput_code: 236, 237, 1542, 1543, 1546
pux_cspace_code: 1465, 1466, 1467
pux_default_cface: 236, 1509, 1528, 1530
pux_default_cface_code: 236, 1475
pux_digit_base: 236, 1425, 1426, 1428
pux_digit_num: 236, 1431, 1434
pux_digit_num_code: 236, 1425, 1426
pux_dump_font_info: 209, 1213, 1565, 1566, 1567
pux_exspace_code: 1465, 1466, 1467
pux_face_match: 209, 1213, 1520, 1521, 1522
pux_font_match: 209, 266, 1213, 1533, 1534
pux_get_int: 209, 416, 1213, 1425, 1426, 1427
pux_local_names_base: 230, 235, 417, 1233, 1234, 1235, 1236
\PUXlocalnames primitive: 1233
pux_nth_digit: 236, 1431, 1434
pux_num_sign: 236, 1431, 1434
pux_range_catcode: 209, 266, 1213, 1414, 1416
pux_range_type_code: 209, 266, 1213, 1414, 1416
pux_set_cface: 209, 266, 1213, 1477, 1482, 1526, 1528, 1535, 1541, 1550
pux_set_cface_attrib: 209, 1213, 1548, 1549, 1550
pux_set_cface_cesp: 1548, 1549, 1550
pux_set_cface_csp: 1548, 1549, 1550
pux_set_cface_depth: 1548, 1549, 1550
pux_set_cfont_attrib: 209, 1213, 1556, 1557, 1558
pux_set_cfont_cesp: 1556, 1557, 1558
pux_set_cfont_csp: 1556, 1557, 1558
pux_set_default_cface: 209, 1213, 1475, 1476, 1477
pux_sign_code: 236, 1425, 1426
pux_space: 209, 1461, 1465, 1466
pux_space_code: 1465, 1466, 1467
pux_split_number: 209, 266, 1213, 1429, 1430
pux_type_code_base: 230, 235, 417, 1233, 1234, 1235, 1236, 1416
\PUXtypecode primitive: 1233
pux_wcharother: 236, 1413
pux_wcharother_code: 236, 237, 1542, 1543, 1546
pux_xspace: 236, 1044, 1461, 1463, 1464, 1467
pux_xspace_code: 236, 237, 1542, 1543, 1546
puxg_assign_flag: 209, 416, 1213, 1542, 1543, 1544
puxg_assign_int: 209, 416, 1213, 1542, 1543, 1546
puxg_assign_space: 209, 1213, 1561, 1562, 1563
puxg_cface_depth: 236, 240, 1480, 1483, 1491, 1547
puxg_cface_depth_code: 236, 237, 1542, 1543, 1546
puxg_rotate_ctext: 236, 1483, 1494, 1545
puxg_rotate_ctext_code: 236, 237, 1542, 1543, 1544
puxg_set_cespace: 1552, 1561, 1562, 1563
puxg_set_cspace: 1552, 1561, 1562, 1563
q: 123, 125, 130, 131, 144, 151, 152, 153, 167, 172, 202, 204, 218, 275, 292, 315, 336, 369, 392, 410, 453, 464, 466, 467, 468, 476, 485, 500, 501, 610, 652, 708, 709, 712, 715, 723, 729, 737, 738, 739, 740, 741, 746, 752, 755, 759, 765, 794, 803, 829, 833, 865, 880, 904, 909, 937, 951, 956, 960, 962, 963, 971, 973, 997, 1015, 1046, 1071, 1082, 1096, 1108, 1122, 1126, 1141, 1187, 1201, 1214, 1239, 1305, 1306, 1373, 1406
qi: 112, 548, 552, 567, 573, 576, 579, 585, 623, 756, 910, 911, 914, 916, 926, 961, 962, 984, 1011, 1012, 1037, 1038, 1039, 1042, 1043, 1103, 1154, 1158, 1163, 1168, 1312, 1328, 1396, 1397, 1400, 1460
qo: 112, 159, 174, 176, 185, 188, 557, 573, 579, 585, 605, 623, 694, 711, 725, 726, 744, 755, 758, 899, 900, 901, 906, 912, 926, 948, 984, 989, 1011, 1021, 1024, 1039, 1042, 1313, 1327, 1328, 1396, 1397, 1400, 1401
qqqq: 110, 114, 553, 557, 572, 576, 577, 686, 716, 744, 755, 912, 1042, 1184, 1342
quad: 550, 561, 1149
quad_code: 550, 561
quarterword: 110, 113, 144, 253, 264, 271, 276, 277, 279, 281, 298, 300, 323, 585, 595, 684, 709, 712, 714, 715, 727, 741, 752, 880, 924, 1064, 1082, 1108, 1291, 1328, 1340, 1396, 1397, 1409
quoted_filename: 32, 518, 519
qw: 563, 567, 573, 576, 579
r: 108, 123, 125, 131, 204, 218, 369, 392, 468, 485, 501, 652, 671, 709, 723, 729, 755, 794, 803, 832, 865, 880, 904, 956, 969, 973, 997, 1015, 1126, 1163, 1201, 1239, 1373
r_count: 915, 917, 921
r_flag: 1486, 1487, 1493, 1494
r_hyf: 894, 895, 897, 902, 905, 926, 1365
r_type: 729, 730, 731, 732, 763, 769, 770
radical: 208, 265, 266, 1049, 1165
\radical primitive: 265
radical_noad: 686, 693, 699, 701, 736, 764, 1166
radical_noad_size: 686, 701, 764, 1166
radix: 369, 441, 442, 443, 447, 448, 451
radix_backup: 369
\raise primitive: 1074
Ramshaw, Lyle Harold: 542
rbrace_ptr: 392, 402, 403
read: 52, 53, 1341, 1342
\read primitive: 265

read_file: 483, 488, 489, 1278
read_font_info: 563, 567, 1043, 1260
read_ln: 52
read_open: 483, 484, 486, 488, 489, 504, 1278
read_sixteen: 567, 568, 571
read_tcx_file: 24
read_to_cs: 209, 265, 266, 1213, 1228
read_toks: 303, 485, 1228
ready_already: 81, 1334, 1335
real: 3, 109, 110, 182, 186, 622, 632, 1126, 1128, 1399
real addition: 1128, 1402
real division: 661, 667, 676, 679, 813, 814, 1126, 1128, 1402
real multiplication: 114, 186, 628, 637, 812, 1128, 1402
rebox: 718, 747, 753
reconstitute: 908, 909, 916, 918, 919, 920, 1035
recorder_change_filename: 537
recursion: 76, 78, 173, 180, 198, 202, 203, 369, 405, 410, 501, 530, 595, 621, 695, 722, 723, 728, 757, 952, 960, 962, 1336, 1378
ref_count: 392, 393, 404
reference counts: 150, 200, 201, 203, 275, 291, 307
register: 209, 414, 415, 416, 1213, 1238, 1239, 1240
regular: 1474
rel_noad: 685, 693, 699, 701, 731, 764, 770, 1159, 1160
rel_penalty: 236, 685, 764
`\relpenalty` primitive: 238
rel_penalty_code: 236, 237, 238
relax: 207, 265, 266, 358, 375, 407, 509, 1048, 1227
`\relax` primitive: 265
rem_byte: 548, 557, 560, 573, 711, 716, 743, 752, 756, 914, 1043
remainder: 104, 106, 107, 460, 461, 546, 547, 548, 719, 720
remember_source_info: 1406
remove_item: 208, 1107, 1110, 1111
rep: 549
replace_c: 1399
replace_count: 145, 175, 195, 843, 861, 872, 885, 886, 921, 1084, 1108, 1123
report_illegal_case: 1048, 1053, 1054, 1246, 1380
reset: 26
reset_cface_cespace: 1479, 1480, 1497
reset_cface_cspace: 1478, 1480, 1497
restart: 15, 125, 126, 341, 346, 357, 359, 360, 362, 383, 755, 756, 785, 788, 792, 1154, 1218
restore_old_value: 268, 276, 282
restore_trace: 283, 284
restore_zero: 268, 276, 278
restrictedshell: 61, 539, 1384
result: 45, 46, 1391, 1396
resume_after_display: 803, 1202, 1203, 1209
reswitch: 15, 341, 343, 352, 466, 622, 623, 652, 654, 655, 729, 731, 937, 938, 1032, 1033, 1039, 1044, 1048, 1141, 1150, 1154, 1196, 1453, 1463, 1464, 1467
return: 15, 16
rewrite: 26
rh: 110, 114, 118, 213, 219, 221, 234, 256, 268, 688
`\right` primitive: 1191
right_brace: 207, 289, 294, 298, 347, 357, 392, 445, 477, 480, 788, 938, 964, 1070, 1255
right_brace_limit: 289, 325, 395, 402, 403, 477, 480
right_brace_token: 289, 339, 1068, 1130, 1229, 1374, 1406
right_delimiter: 686, 700, 751, 1184, 1185
right_hyphen_min: 236, 1094, 1203, 1379, 1380
`\righthyphenmin` primitive: 238
right_hyphen_min_code: 236, 237, 238
right_noad: 690, 693, 699, 701, 728, 731, 763, 764, 765, 1187, 1191, 1194
right_ptr: 608, 609, 610, 618
right_skip: 224, 830, 883, 884
`\rightskip` primitive: 226
right_skip_code: 224, 225, 226, 884, 889
right1: 588, 589, 610, 613, 619
right2: 588, 613
right3: 588, 613
right4: 588, 613
rlink: 124, 125, 126, 127, 129, 130, 131, 132, 145, 149, 164, 169, 775, 822, 824, 1314, 1315
`\romannumeral` primitive: 471
roman_numeral_code: 471, 472, 474, 475
rotated: 1474, 1483, 1494, 1545
round: 3, 114, 186, 628, 637, 812, 1128, 1402
round_decimals: 102, 103, 455
rover: 124, 125, 126, 127, 128, 129, 130, 131, 132, 164, 169, 1314, 1315
rt_hit: 909, 910, 913, 914, 1036, 1038, 1043
rule_dp: 595, 625, 627, 629, 634, 636, 638
rule_ht: 595, 625, 627, 629, 634, 636, 637, 638, 639
rule_node: 138, 139, 148, 175, 183, 202, 206, 625, 629, 634, 638, 654, 656, 672, 673, 733, 764, 808, 844, 845, 869, 873, 874, 971, 976, 1003, 1077, 1090, 1124, 1150
rule_node_size: 138, 139, 202, 206
rule_save: 803, 807
rule_wd: 595, 625, 627, 628, 629, 630, 634, 636, 638
rules aligning with characters: 592
runaway: 120, 306, 338, 399, 489
`Runaway...`: 306

runsystem: 1373
runsystem_ret: 1373
s: 45, 46, 58, 59, 60, 62, 63, 93, 94, 95, 103, 108, 125, 130, 147, 177, 178, 264, 284, 392, 410, 476, 485, 520, 532, 533, 563, 641, 648, 652, 671, 691, 702, 709, 723, 729, 741, 794, 803, 833, 865, 880, 904, 937, 969, 990, 1015, 1063, 1064, 1126, 1141, 1201, 1239, 1260, 1282, 1352, 1358, 1391, 1392, 1440, 1525
s_flag: 1486, 1487, 1493
save_area_delimiter: 528
save_cond_ptr: 501, 503, 512
save_cs_ptr: 777, 780
save_cur_val: 453, 458
save_cur_wchar: 1033, 1453, 1454, 1456, 1459, 1463, 1464
save_ext_delimiter: 528
save_for_after: 280, 1274
save_h: 622, 626, 630, 631, 632, 635, 640
save_index: 268, 274, 276, 280, 282
save_level: 268, 269, 274, 276, 280, 282
save_link: 833, 860
save_loc: 622, 632
save_name_in_progress: 528
save_pool_ptr: 1384
save_ptr: 268, 271, 272, 273, 274, 276, 280, 282, 283, 285, 648, 807, 1089, 1102, 1103, 1120, 1123, 1145, 1156, 1171, 1175, 1177, 1189, 1197, 1307
save_scanner_status: 369, 372, 392, 473, 474, 497, 501, 510
save_size: 32, 111, 271, 273, 1335, 1337
save_split_top_skip: 1015, 1017
save_stack: 203, 268, 270, 271, 273, 274, 275, 276, 277, 281, 282, 283, 285, 300, 375, 492, 648, 771, 1065, 1074, 1134, 1143, 1153, 1156, 1335, 1342
save_stop_at_space: 528
save_str_ptr: 1384
save_style: 723, 729, 757
save_type: 268, 274, 276, 280, 282
save_v: 622, 626, 631, 632, 635, 639, 640
save_vbadness: 1015, 1020
save_vfuzz: 1015, 1020
save_warning_index: 392
saved: 274, 648, 807, 1086, 1089, 1102, 1103, 1120, 1122, 1145, 1156, 1171, 1175, 1177, 1189, 1197
saved_cur_area: 533
saved_cur_ext: 533
saved_cur_name: 533
saved_val: 473, 1433, 1436, 1437
sc: 110, 113, 114, 135, 150, 159, 164, 213, 219, 247, 250, 251, 416, 423, 428, 553, 557, 560, 561, 574, 576, 578, 583, 703, 704, 778, 825, 826, 835, 846, 847, 851, 853, 863, 864, 892, 1045, 1152, 1209, 1250, 1251, 1256, 1340, 1342
scaled: 101, 102, 103, 104, 105, 106, 107, 108, 110, 113, 147, 150, 156, 176, 177, 450, 451, 453, 456, 551, 552, 563, 587, 595, 610, 619, 622, 632, 649, 652, 671, 682, 707, 708, 709, 715, 718, 719, 720, 722, 729, 738, 739, 740, 741, 746, 752, 759, 765, 794, 803, 826, 833, 842, 850, 880, 909, 973, 974, 980, 983, 985, 997, 1015, 1071, 1089, 1126, 1141, 1201, 1260, 1326, 1340, 1399, 1441, 1502, 1508, 1512, 1514
`scaled`: 1261
scaled_base: 247, 249, 251, 1227, 1240
scan_box: 1076, 1087, 1244
scan_char_num: 417, 437, 938, 1033, 1126, 1127, 1154, 1157, 1227, 1235, 1453, 1460, 1463, 1464
scan_delimiter: 1163, 1166, 1185, 1186, 1194, 1195
scan_dimen: 413, 443, 450, 451, 464, 465, 1064
scan_eight_bit_int: 417, 418, 423, 430, 436, 508, 1082, 1085, 1102, 1113, 1227, 1229, 1230, 1235, 1240, 1244, 1250, 1299, 1433, 1437
scan_fifteen_bit_int: 439, 1154, 1157, 1168, 1227
scan_file_name: 265, 334, 529, 530, 540, 1260, 1278, 1354
scan_font_ident: 418, 429, 474, 580, 581, 1237, 1256
scan_four_bit_int: 438, 580, 1237, 1278, 1353, 1388
scan_four_bit_int_or_18: 504, 1388
scan_glue: 413, 464, 785, 1063, 1231, 1241, 1558
scan_int: 412, 413, 435, 436, 437, 438, 439, 440, 441, 443, 450, 451, 464, 474, 506, 507, 512, 581, 1106, 1228, 1231, 1235, 1241, 1243, 1246, 1247, 1249, 1251, 1256, 1261, 1353, 1380, 1388, 1416, 1419, 1428, 1430, 1432, 1433, 1488, 1489, 1490, 1491, 1492, 1544, 1546, 1550, 1552
scan_keyword: 162, 410, 456, 457, 458, 459, 461, 465, 466, 648, 1085, 1228, 1239, 1261, 1416, 1433, 1437, 1552
scan_left_brace: 406, 476, 648, 788, 937, 963, 1028, 1102, 1120, 1122, 1156, 1175, 1177
scan_math: 1153, 1154, 1161, 1166, 1168, 1179
scan_name: 1418, 1484, 1485, 1526
scan_normal_dimen: 451, 466, 506, 648, 1076, 1085, 1185, 1186, 1231, 1241, 1246, 1248, 1250, 1251, 1256, 1262
scan_optional_equals: 408, 785, 1227, 1229, 1231, 1235, 1237, 1239, 1244, 1246, 1247, 1248, 1249, 1250, 1251, 1256, 1260, 1278, 1354, 1416, 1437, 1482, 1488, 1489, 1490, 1491, 1492, 1493, 1544, 1546, 1550, 1552, 1558
scan_rule_spec: 466, 1059, 1087
scan_something_internal: 412, 413, 416, 435, 443,

452, 454, 458, 464, 468
scan_spec: 648, 771, 777, 1074, 1086, 1170
scan_toks: 291, 467, 476, 963, 1104, 1221, 1229, 1282, 1291, 1355, 1357, 1374
scan_twenty_seven_bit_int: 440, 1154, 1157, 1163
scan_wchar_num: 417, 938, 1033, 1154, 1227, 1235, 1416, 1419, 1445, 1452, 1453, 1460, 1463, 1464
scanned_result: 416, 417, 418, 421, 425, 428, 429, 431, 1428
scanned_result_end: 416
scanner_status: 305, 306, 331, 336, 339, 369, 372, 392, 394, 473, 474, 476, 485, 497, 501, 510, 780, 792
\PUXscnumber primitive: 471
scnumber_code: 471, 472, 474, 475
\scriptfont primitive: 1233
script_mlist: 692, 698, 701, 734, 1177
\scriptscriptfont primitive: 1233
script_script_mlist: 692, 698, 701, 734, 1177
script_script_size: 702, 759, 1198, 1233
script_script_style: 691, 697, 734, 1172
\scriptscriptstyle primitive: 1172
script_size: 702, 759, 1198, 1233
script_space: 247, 760, 761, 762
\scriptspace primitive: 248
script_space_code: 247, 248
script_style: 691, 697, 705, 706, 734, 759, 765, 769, 1172
\scriptstyle primitive: 1172
scripts_allowed: 690, 1179
scroll_mode: 71, 73, 84, 86, 93, 533, 1265, 1266, 1284
\scrollmode primitive: 1265
search: 1391
search_mem: 165, 172, 255, 1342
search_string: 520, 540, 1391, 1392
second_indent: 850, 851, 852, 892
second_pass: 831, 866, 869
second_width: 850, 851, 852, 853, 892
Sedgewick, Robert: 2
see the transcript file...: 1338
selector: 54, 55, 57, 58, 59, 62, 71, 75, 86, 90, 92, 98, 245, 311, 312, 316, 360, 468, 473, 537, 538, 620, 641, 1260, 1268, 1282, 1301, 1331, 1336, 1338, 1371, 1373, 1377, 1482, 1564, 1567
semi_simple_group: 269, 1066, 1068, 1071, 1072
serial: 824, 848, 849, 859
set_aux: 209, 416, 419, 420, 421, 1213, 1245
set_box: 209, 265, 266, 1213, 1244
\setbox primitive: 265
set_box_allowed: 76, 77, 1244, 1273
set_box_dimen: 209, 416, 419, 420, 1213, 1245
set_break_width_to_background: 840
set_ceglue_spec: 1506, 1514, 1554, 1563
set_cfont: 209, 416, 580, 1213, 1509, 1515, 1524, 1535, 1539, 1558
set_cglue_spec: 1505, 1514, 1553, 1563
set_char_0: 588, 589, 623
set_conversion: 461
set_conversion_end: 461
set_cur_lang: 937, 963, 1094, 1203
set_cur_r: 911, 913, 914
set_font: 209, 416, 556, 580, 1213, 1220, 1260, 1264, 1535, 1558
set_glue_ratio_one: 109, 667, 679, 813, 814
set_glue_ratio_zero: 109, 136, 660, 661, 667, 675, 676, 679, 813, 814
set_head_forbidden: 1412
set_height_zero: 973
set_interaction: 209, 1213, 1265, 1266, 1267
\setlanguage primitive: 1347
set_language_code: 1347, 1349, 1351
set_math_char: 1157, 1158
set_page_dimen: 209, 416, 985, 986, 987, 1213, 1245
set_page_int: 209, 416, 419, 420, 1213, 1245
set_page_so_far_zero: 990
set_prev_graf: 209, 265, 266, 416, 1213, 1245
set_rule: 586, 588, 589, 627
set_shape: 209, 265, 266, 416, 1213, 1251
set_tail_forbidden: 1412
set_trick_count: 316, 317, 318, 320
set_type_code: 207, 1412
set_type_code_end: 207
setup_bound_var: 1335
setup_bound_var_end: 1335
setup_bound_var_end_end: 1335
setup_bound_variable: 1335
set1: 588, 589, 623, 1402
set2: 588, 589, 623
set3: 588
set4: 588, 589
sf_code: 230, 232, 1037
\sfcode primitive: 1233
sf_code_base: 230, 235, 1233, 1234, 1236
shape_ref: 210, 232, 275, 1073, 1251
shellenabledp: 61, 504, 539, 1373, 1384
shift_amount: 135, 136, 159, 184, 626, 631, 635, 640, 652, 656, 671, 673, 684, 709, 723, 740, 741, 752, 753, 759, 760, 762, 802, 809, 810, 811, 892, 1079, 1084, 1128, 1149, 1206, 1207, 1208
shift_case: 1288, 1291
shift_down: 746, 747, 748, 749, 750, 752, 754, 759, 760, 762

shift_up: 746, 747, 748, 749, 750, 752, 754, 759, 761, 762
ship_out: 211, 595, 641, 647, 1026, 1078, 1398
`\shipout` primitive: 1074
ship_out_flag: 1074, 1078
short_display: 173, 174, 175, 193, 666, 860, 1342
short_real: 109, 110
shortcut: 450, 451
shortfall: 833, 854, 855, 856
shorthand_def: 209, 1213, 1225, 1226, 1227
`\show` primitive: 1294
show_activities: 218, 1296
show_box: 180, 182, 198, 218, 219, 236, 641, 644, 666, 678, 989, 995, 1124, 1299, 1342
`\showbox` primitive: 1294
show_box_breadth: 236, 1342
`\showboxbreadth` primitive: 238
show_box_breadth_code: 236, 237, 238
show_box_code: 1294, 1295, 1296
show_box_depth: 236, 1342
`\showboxdepth` primitive: 238
show_box_depth_code: 236, 237, 238
show_code: 1294, 1296
show_context: 54, 78, 82, 88, 310, 311, 318, 533, 538, 540
show_cur_cmd_chr: 299, 370, 1034
show_eqtb: 252, 284
show_info: 695, 696
show_lists: 1294, 1295, 1296
`\showlists` primitive: 1294
show_node_list: 173, 176, 180, 181, 182, 195, 198, 233, 693, 695, 696, 698, 1342
`\showthe` primitive: 1294
show_the_code: 1294, 1295
show_token_list: 176, 223, 233, 292, 295, 306, 319, 320, 403, 1342, 1371
show_whatever: 1293, 1296
shown_mode: 213, 215, 299
shrink: 150, 151, 164, 178, 434, 465, 628, 637, 659, 674, 719, 812, 828, 830, 841, 871, 979, 1007, 1012, 1045, 1047, 1151, 1232, 1242, 1243, 1505, 1506, 1558
shrink_order: 150, 164, 178, 465, 628, 637, 659, 674, 719, 812, 828, 829, 979, 1007, 1012, 1151, 1242
shrink_value: 1551, 1552, 1553, 1554, 1563
shrinking: 135, 186, 622, 632, 667, 679, 812, 813, 814, 1151
si: 38, 42, 69, 967, 1313, 1340
sign: 473, 1433, 1434, 1436
simple: 1422, 1423, 1424
simple_group: 269, 1066, 1071
Single-character primitives: 267
`\-`: 1117
`\/`: 265
`\␣`: 265
single_base: 222, 262, 263, 264, 354, 377, 445, 1260, 1292, 1482
size: 1508, 1509, 1511, 1512, 1513, 1514
skew_char: 429, 552, 579, 744, 1256, 1325, 1326, 1340
`\skewchar` primitive: 1257
skip: 224, 430, 1012
`\skip` primitive: 414
skip_base: 224, 227, 229, 1227, 1240
skip_blanks: 303, 344, 345, 347, 349, 354
skip_byte: 548, 560, 744, 755, 756, 912, 1042
skip_code: 1061, 1062, 1063
`\skipdef` primitive: 1225
skip_def_code: 1225, 1226, 1227
skip_line: 336, 496, 497
skipping: 305, 306, 336, 497
slant: 550, 561, 578, 1126, 1128, 1402
slant_code: 550, 561
slow_make_string: 520, 944, 1392
slow_print: 60, 61, 63, 539, 540, 584, 1264, 1283, 1286, 1331, 1342, 1396, 1400, 1401, 1515
small_char: 686, 694, 700, 709, 1163
small_fam: 686, 694, 700, 709, 1163
small_node_size: 141, 144, 145, 147, 152, 153, 156, 158, 202, 206, 658, 724, 906, 913, 917, 1040, 1103, 1104, 1360, 1361, 1379, 1380, 1450
small_number: 101, 102, 147, 152, 154, 264, 369, 392, 416, 441, 443, 453, 464, 473, 485, 492, 497, 500, 501, 526, 610, 652, 671, 691, 709, 722, 723, 729, 759, 765, 832, 895, 896, 908, 909, 924, 937, 947, 963, 973, 990, 1063, 1089, 1094, 1179, 1184, 1194, 1201, 1214, 1239, 1250, 1260, 1338, 1352, 1353, 1373, 1376, 1482, 1524, 1526, 1535
small_op: 946
so: 38, 45, 59, 60, 69, 70, 264, 410, 467, 521, 522, 606, 620, 769, 934, 956, 958, 959, 962, 966, 1312, 1371, 1373
`Sorry, I can't find...`: 527
sort_avail: 131, 1314
source_filename_stack: 304, 328, 331, 540, 1335, 1406
sp: 104, 590
`sp`: 461
space: 550, 561, 755, 758, 1045
space_code: 550, 561, 581, 1045
space_factor: 212, 213, 421, 789, 790, 802, 1037, 1046, 1047, 1059, 1079, 1086, 1094, 1096, 1120, 1122, 1126, 1199, 1203, 1245, 1246, 1463, 1467

\spacefactor primitive: 419
space_shrink: 550, 561, 1045
space_shrink_code: 550, 561, 581
space_skip: 224, 1044, 1046
\spaceskip primitive: 226
space_skip_code: 224, 225, 226, 1044
space_stretch: 550, 561, 1045
space_stretch_code: 550, 561
space_token: 289, 396, 467, 1218
spacer: 207, 208, 232, 289, 291, 294, 298, 303, 337, 345, 347, 348, 349, 354, 407, 409, 410, 446, 447, 455, 467, 786, 938, 964, 1048, 1224, 1461, 1462
\span primitive: 783
span_code: 783, 784, 785, 792, 794
span_count: 136, 159, 185, 799, 804, 811
span_node_size: 800, 801, 806
spec_code: 648
\special primitive: 1347
special_node: 1344, 1347, 1349, 1351, 1357, 1359, 1360, 1361, 1376, 1406
special_out: 1371, 1376
split: 1014
split_bot_mark: 385, 386, 980, 982
\splitbotmark primitive: 387
split_bot_mark_code: 385, 387, 388, 1338
split_first_mark: 385, 386, 980, 982
\splitfirstmark primitive: 387
split_first_mark_code: 385, 387, 388
split_max_depth: 140, 247, 980, 1071, 1103
\splitmaxdepth primitive: 248
split_max_depth_code: 247, 248
split_number: 1430, 1431, 1432, 1433
split_top_ptr: 140, 188, 202, 206, 1024, 1025, 1103
split_top_skip: 140, 224, 971, 980, 1015, 1017, 1024, 1103
\splittopskip primitive: 226
split_top_skip_code: 224, 225, 226, 972
split_up: 984, 989, 1011, 1013, 1023, 1024
spotless: 76, 77, 81, 245, 1335, 1338
spread: 648
sprint_cs: 223, 263, 338, 398, 399, 401, 475, 482, 487, 564, 1297
square roots: 740
src_specials: 32
src_specials_p: 32, 61, 539
ss_code: 1061, 1062, 1063
ss_glue: 162, 164, 718, 1063
ssup_error_line: 11, 54, 1335
ssup_hyph_size: 11, 928
ssup_max_strings: 11, 38
ssup_trie_opcode: 11, 923
ssup_trie_size: 11, 923, 1335
stack conventions: 300
stack_into_box: 714, 716
stack_size: 32, 301, 310, 321, 1335, 1337
start: 300, 302, 303, 307, 318, 319, 323, 324, 325, 328, 329, 331, 360, 362, 363, 372, 486, 541
start_cs: 341, 354, 355
start_eq_no: 1143, 1145
start_field: 300, 302
start_font_error_message: 564, 570
start_here: 5, 1335
start_input: 369, 379, 381, 540, 1340
start_of_TEX: 6, 1335
start_par: 208, 1091, 1092, 1093, 1095
stat: 7, 117, 120, 121, 122, 123, 125, 130, 252, 260, 283, 284, 642, 832, 848, 858, 866, 990, 1008, 1013, 1336
state: 87, 300, 302, 303, 307, 311, 312, 323, 325, 328, 330, 331, 337, 341, 343, 344, 346, 347, 349, 352, 353, 354, 393, 486, 529, 540, 1338
state_field: 300, 302, 1134
stderr: 1309, 1385
stdin: 32
stdout: 32, 61, 527
stomach: 405
stop: 207, 1048, 1049, 1055, 1056, 1057, 1097
stop_at_space: 519, 528, 1382, 1383
stop_flag: 548, 560, 744, 755, 756, 912, 1042
store_background: 867
store_break_width: 846
store_fmt_file: 1305, 1338
store_four_quarters: 567, 571, 572, 576, 577
store_new_token: 374, 375, 396, 400, 402, 410, 467, 469, 476, 477, 479, 480, 485, 486
store_scaled: 574, 576, 578
str_eq_buf: 45, 259
str_eq_str: 46, 1263, 1391, 1481, 1485, 1523
str_number: 38, 39, 43, 45, 46, 47, 62, 63, 79, 93, 94, 95, 177, 178, 264, 284, 304, 410, 515, 520, 522, 528, 530, 532, 533, 535, 540, 552, 563, 929, 932, 937, 1260, 1282, 1302, 1326, 1335, 1340, 1358, 1384, 1391, 1392, 1407, 1418, 1474, 1481, 1482, 1508, 1518, 1523, 1524, 1525, 1526, 1530, 1540
str_pool: 38, 39, 42, 43, 45, 46, 47, 59, 60, 69, 70, 256, 260, 264, 303, 410, 467, 520, 521, 522, 605, 606, 620, 641, 767, 769, 932, 934, 937, 944, 1260, 1311, 1312, 1313, 1335, 1336, 1337, 1371, 1373, 1383, 1385, 1510, 1511, 1525
str_ptr: 38, 39, 41, 43, 44, 47, 59, 60, 70, 260, 262, 520, 528, 540, 620, 1312, 1313, 1326, 1328, 1330, 1335, 1337, 1371, 1373, 1418, 1510

str_room: 42, 180, 260, 467, 519, 520, 528, 942, 1260, 1282, 1331, 1336, 1371, 1373, 1482
str_start: 38, 39, 40, 41, 43, 44, 45, 46, 47, 59, 60, 69, 70, 84, 256, 260, 264, 410, 520, 521, 522, 605, 606, 620, 768, 932, 934, 937, 944, 1260, 1311, 1312, 1313, 1335, 1371, 1373, 1385, 1418, 1510, 1525
str_toks: 467, 468, 473, 1406
strcmp: 1311
strcpy: 51, 1310
stretch: 150, 151, 164, 178, 434, 465, 628, 637, 659, 674, 719, 812, 830, 841, 871, 979, 1007, 1012, 1045, 1047, 1151, 1232, 1242, 1243, 1505, 1506, 1558
stretch_order: 150, 164, 178, 465, 628, 637, 659, 674, 719, 812, 830, 841, 871, 979, 1007, 1012, 1151, 1242
stretch_value: 1551, 1552, 1553, 1554, 1563
stretching: 135, 628, 637, 661, 676, 812, 813, 814, 1151
strikeout: 1474, 1493
string pool: 47, 1311
`\string` primitive: 471
string_code: 471, 472, 474, 475
string_vacancies: 32, 52, 1335
stringcast: 51, 527, 537, 540, 1278, 1310, 1311, 1377
strings_free: 32, 1313, 1335
strlen: 51, 620, 1310
style: 729, 763, 764, 765, 1482, 1483, 1493, 1494, 1496, 1497
style_node: 160, 691, 693, 701, 733, 734, 764, 1172
style_node_size: 691, 692, 701, 766
sub_box: 684, 690, 695, 701, 723, 737, 738, 740, 741, 752, 757, 1079, 1096, 1171
sub_drop: 703, 759
sub_mark: 207, 294, 298, 347, 1049, 1178
sub_mlist: 684, 686, 695, 723, 745, 757, 1184, 1188, 1189, 1194
sub_style: 705, 753, 760, 762
sub_sup: 1178, 1179
subscr: 684, 686, 689, 690, 693, 699, 701, 741, 745, 752, 753, 754, 755, 756, 757, 758, 759, 760, 762, 1154, 1166, 1168, 1178, 1179, 1180, 1189
subscripts: 757, 1178
subtype: 133, 134, 135, 136, 139, 140, 143, 144, 145, 146, 147, 149, 150, 152, 153, 154, 155, 156, 158, 159, 188, 189, 190, 191, 192, 193, 427, 492, 498, 499, 628, 630, 637, 639, 652, 659, 671, 674, 684, 685, 689, 691, 692, 693, 699, 720, 733, 734, 735, 736, 752, 766, 769, 771, 789, 798, 812, 822, 823, 825, 840, 846, 847, 869, 871, 882, 884, 899, 900, 901, 902, 906, 913, 984, 989, 991, 1011, 1012, 1021, 1023, 1024, 1038, 1063, 1064, 1081, 1103, 1104, 1116, 1128, 1151, 1162, 1166, 1168, 1174, 1184, 1338, 1344, 1352, 1359, 1360, 1361, 1365, 1376, 1377, 1450
sub1: 703, 760
sub2: 703, 762
succumb: 93, 94, 95, 1307
sup: 1335
sup_buf_size: 11
sup_drop: 703, 759
sup_dvi_buf_size: 11
sup_expand_depth: 11
sup_font_max: 11
sup_font_mem_size: 11, 1324
sup_hash_extra: 11, 1311
sup_hyph_size: 11
sup_main_memory: 11, 111, 1335
sup_mark: 207, 294, 298, 344, 355, 1049, 1178, 1179, 1180
sup_max_in_open: 11
sup_max_strings: 11, 1313
sup_mem_bot: 11
sup_nest_size: 11
sup_param_size: 11
sup_pool_free: 11
sup_pool_size: 11, 1313
sup_save_size: 11
sup_stack_size: 11
sup_string_vacancies: 11
sup_strings_free: 11
sup_style: 705, 753, 761
sup_trie_size: 11
superscripts: 757, 1178
supscr: 684, 686, 689, 690, 693, 699, 701, 741, 745, 753, 754, 755, 756, 757, 759, 761, 1154, 1166, 1168, 1178, 1179, 1180, 1189
sup1: 703, 761
sup2: 703, 761
sup3: 703, 761
sw: 563, 574, 578, 1441
switch: 341, 343, 344, 346, 350
synch_h: 619, 623, 627, 631, 636, 640, 1371, 1402
synch_v: 619, 623, 627, 631, 635, 636, 640, 1371, 1402
system: 1373
system dependencies: 2, 3, 9, 10, 11, 12, 19, 21, 23, 26, 32, 34, 35, 37, 38, 49, 56, 59, 72, 81, 84, 96, 109, 110, 112, 113, 161, 186, 241, 304, 313, 328, 367, 488, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 526, 528, 541,

560, 567, 594, 598, 600, 801, 923, 1334, 1335, 1336, 1341, 1343, 1405
s1: 82, 88
s2: 82, 88
s3: 82, 88
s4: 82, 88
t: 46, 107, 108, 125, 218, 277, 279, 280, 281, 323, 341, 369, 392, 467, 476, 520, 707, 708, 729, 759, 803, 833, 880, 909, 969, 973, 1033, 1126, 1179, 1194, 1201, 1260, 1291, 1392, 1482, 1524
t_open_in: 33, 37
t_open_out: 33, 1335
tab_mark: 207, 289, 294, 342, 347, 783, 784, 785, 786, 787, 791, 1129
tab_skip: 224
`\tabskip` primitive: 226
tab_skip_code: 224, 225, 226, 781, 785, 789, 798, 812
tab_token: 289, 1131
table: 1529
tag: 546, 547, 557
tail: 212, 213, 214, 215, 216, 427, 682, 721, 779, 789, 798, 799, 802, 815, 819, 891, 893, 998, 1020, 1026, 1029, 1037, 1038, 1039, 1040, 1043, 1044, 1046, 1057, 1063, 1064, 1079, 1081, 1083, 1084, 1094, 1099, 1103, 1104, 1108, 1113, 1116, 1120, 1122, 1123, 1126, 1128, 1148, 1153, 1158, 1161, 1162, 1166, 1168, 1171, 1174, 1177, 1179, 1180, 1184, 1187, 1189, 1190, 1194, 1199, 1208, 1209, 1311, 1352, 1353, 1354, 1355, 1356, 1357, 1378, 1379, 1380, 1406, 1451, 1453, 1454, 1456, 1457, 1462, 1463, 1464
tail_append: 214, 789, 798, 819, 1038, 1040, 1043, 1057, 1059, 1063, 1064, 1094, 1096, 1103, 1106, 1115, 1116, 1120, 1153, 1161, 1166, 1168, 1171, 1174, 1175, 1180, 1194, 1199, 1206, 1208, 1209, 1450, 1451, 1452, 1453, 1457, 1458, 1459, 1463, 1467
tail_append_glue: 1450, 1451, 1452, 1453, 1457, 1459, 1463, 1464
tail_field: 212, 213, 998, 1451, 1454
tail_forbidden: 207, 1412, 1447
tally: 54, 55, 57, 58, 292, 312, 315, 316, 317
tats: 7
temp_head: 162, 306, 394, 399, 403, 467, 469, 470, 473, 481, 722, 723, 757, 763, 819, 865, 866, 867, 880, 882, 883, 884, 890, 971, 1067, 1068, 1197, 1199, 1202, 1300, 1406
temp_ptr: 115, 154, 621, 622, 626, 631, 632, 635, 640, 643, 682, 695, 696, 972, 1004, 1024, 1040, 1044, 1338
temp_str: 520, 540
ten_thousand_wchar_offset: 1421, 1422
ten_wchar_offset: 1421
term_and_log: 54, 57, 58, 71, 75, 92, 245, 537, 1301, 1331, 1338, 1373, 1377
term_in: 32, 36, 37, 71, 1341, 1342
term_input: 71, 78
term_offset: 54, 55, 57, 58, 61, 62, 71, 540, 641, 1283
term_only: 54, 55, 57, 58, 71, 75, 92, 538, 1301, 1336, 1338, 1373
term_out: 32, 34, 36, 37, 51, 56
terminal_input: 304, 313, 328, 330, 360
test_char: 909, 912
TEX: 4
`TeX capacity exceeded ...`: 94
  buffer size: 35, 328, 377
  exception dictionary: 943
  font memory: 583
  grouping levels: 274
  hash size: 260
  input stack size: 321
  main memory size: 120, 125
  number of strings: 43, 520
  parameter stack size: 393
  pattern memory: 957, 967
  pool size: 42
  save size: 273
  semantic nest size: 216
  text input levels: 328
`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: 517
TeX_banner: 2
TeX_banner_k: 2
TEX_font_area: 517
TEX_format_default: 523, 526, 527
tex_input_type: 540, 1278
tex_int_pars: 236
tex_remainder: 104
The TEXbook: 1, 23, 49, 108, 207, 418, 449, 459, 462, 686, 691, 767, 1218, 1334
`TeXformats`: 11, 524
TEXMF_ENGINE_NAME: 11
texmf_log_name: 535
TEXMF_POOL_NAME: 11
`texput`: 35, 537, 1260
text: 256, 258, 259, 260, 262, 263, 264, 265, 494, 556, 783, 1191, 1219, 1260, 1311, 1321, 1335, 1347, 1372, 1385, 1474, 1482, 1524
`Text line contains...`: 346

text_char: 19, 20, 25, 26, 47, 1305, 1306, 1310, 1311
\textfont primitive: 1233
text_mlist: 692, 698, 701, 734, 1177
text_size: 702, 706, 735, 765, 1198, 1202
text_style: 691, 697, 706, 734, 740, 747, 748, 749, 751, 752, 761, 765, 1172, 1197, 1199
\textstyle primitive: 1172
TEX82: 1, 99
TFM files: 542
tfm_file: 542, 563, 566, 567, 578
tfm_temp: 567
TFtoPL: 564
That makes 100 errors...: 82
the: 210, 265, 266, 366, 370, 481
The following...deleted: 644, 995, 1124
\the primitive: 265
the_toks: 468, 469, 470, 481, 1300
thick_mu_skip: 224
\thickmuskip primitive: 226
thick_mu_skip_code: 224, 225, 226, 769
thickness: 686, 700, 728, 746, 747, 749, 750, 1185
thin_mu_skip: 224
\thinmuskip primitive: 226
thin_mu_skip_code: 224, 225, 226, 229, 769
thirty_wchar_offset: 1421, 1423
This can't happen: 95
align: 803
copying: 206
curlevel: 281
disc1: 844
disc2: 845
disc3: 873
disc4: 874
display: 1203
endv: 794
ext1: 1351
ext2: 1360
ext3: 1361
ext4: 1376
flushing: 202
if: 500
line breaking: 880
mlist1: 731
mlist2: 757
mlist3: 764
mlist4: 769
page: 1003
paragraph: 869
prefix: 1214
pruning: 971
right: 1188
rightbrace: 1071
vcenter: 739
vertbreak: 976
vlistout: 633
vpack: 672
256 spans: 801
this_box: 622, 627, 628, 632, 636, 637
this_if: 501, 504, 506, 508, 509
thousand_wchar_offset: 1421, 1424
three_codes: 648
threshold: 831, 854, 857, 866
Tight \hbox...: 670
Tight \vbox...: 681
tight_fit: 820, 822, 833, 836, 837, 839, 856
time: 236, 241, 539, 620
\time primitive: 238
time_code: 236, 237, 238
tini: 8
Tini: 8
to: 648, 1085, 1228
tok_val: 413, 418, 421, 431, 468
token: 289
token_list: 307, 311, 312, 323, 325, 330, 337, 341, 346, 393, 529, 1134, 1338
token_ref_count: 200, 203, 291, 476, 485, 982, 1406
token_show: 295, 296, 323, 404, 1282, 1287, 1300, 1373
token_type: 307, 311, 312, 314, 319, 323, 324, 325, 327, 382, 393, 1029, 1098
toklist: 1406
toks: 230
\toks primitive: 265
toks_base: 230, 231, 232, 233, 418, 1227, 1229, 1230
\toksdef primitive: 1225
toks_def_code: 1225, 1226, 1227
toks_register: 209, 265, 266, 416, 418, 1213, 1229, 1230
tolerance: 236, 240, 831, 866
\tolerance primitive: 238
tolerance_code: 236, 237, 238
Too many }'s: 1071
too_small: 1306, 1309
top: 549
top_bot_mark: 210, 296, 366, 370, 387, 388, 389
top_edge: 632, 639
top_mark: 385, 386, 1015
\topmark primitive: 387
top_mark_code: 385, 387, 389, 1338
top_skip: 224
\topskip primitive: 226
top_skip_code: 224, 225, 226, 1004
total_demerits: 822, 848, 849, 858, 867, 877, 878
total height: 989

total_mathex_params: 704, 1198
total_mathsy_params: 703, 1198
total_pages: 595, 596, 620, 643, 645
total_shrink: 649, 653, 659, 667, 668, 669, 670, 674, 679, 680, 681, 799, 1204
total_stretch: 649, 653, 659, 661, 662, 663, 674, 676, 677, 799
Trabb Pardo, Luis Isidoro: 2
tracing_char_sub_def: 236, 240, 1227
`\tracingcharsubdef` primitive: 238
tracing_char_sub_def_code: 236, 237, 238
tracing_commands: 236, 370, 501, 512, 1034
`\tracingcommands` primitive: 238
tracing_commands_code: 236, 237, 238
tracing_lost_chars: 236, 584, 1401
`\tracinglostchars` primitive: 238
tracing_lost_chars_code: 236, 237, 238
tracing_macros: 236, 323, 392, 403
`\tracingmacros` primitive: 238
tracing_macros_code: 236, 237, 238
tracing_online: 236, 245, 1296, 1301, 1373, 1377
`\tracingonline` primitive: 238
tracing_online_code: 236, 237, 238
tracing_output: 236, 641, 644
`\tracingoutput` primitive: 238
tracing_output_code: 236, 237, 238
tracing_pages: 236, 990, 1008, 1013
`\tracingpages` primitive: 238
tracing_pages_code: 236, 237, 238
tracing_paragraphs: 236, 848, 858, 866
`\tracingparagraphs` primitive: 238
tracing_paragraphs_code: 236, 237, 238
tracing_restores: 236, 283
`\tracingrestores` primitive: 238
tracing_restores_code: 236, 237, 238
tracing_stats: 117, 236, 642, 1329, 1336
`\tracingstats` primitive: 238
tracing_stats_code: 236, 237, 238
tracinglostchars: 1401
`Transcript written...`: 1336
translate_filename: 24, 61, 539, 1390
trap_zero_glue: 1231, 1232, 1239
trick_buf: 54, 58, 315, 317
trick_count: 54, 58, 315, 316, 317
Trickey, Howard Wellington: 2
trie: 923, 924, 925, 953, 955, 956, 957, 961, 962, 969
trie_back: 953, 957, 959
trie_c: 950, 951, 956, 958, 959, 962, 966, 967, 1340
trie_char: 923, 924, 926, 961, 962
trie_fix: 961, 962
trie_hash: 950, 951, 952, 953, 955, 1340
trie_l: 950, 951, 952, 960, 962, 963, 966, 967, 1340
trie_link: 923, 924, 926, 953, 955, 956, 957, 958, 959, 961, 962
trie_max: 953, 955, 957, 961, 1327, 1328
trie_min: 953, 955, 956, 959
trie_node: 951, 952
trie_not_ready: 894, 953, 954, 963, 969, 1327, 1328, 1340
trie_o: 950, 951, 962, 966, 967, 1340
trie_op: 923, 924, 926, 927, 946, 961, 962
trie_op_hash: 11, 946, 947, 948, 949, 951, 955
trie_op_lang: 946, 947, 948, 955
trie_op_ptr: 946, 947, 948, 949, 1327, 1328
trie_op_size: 11, 924, 946, 947, 949, 1327, 1328
trie_op_val: 946, 947, 948, 955
trie_opcode: 923, 924, 946, 947, 950, 963, 1340
trie_pack: 960, 969
trie_pointer: 923, 924, 925, 950, 951, 952, 953, 956, 960, 962, 963, 969, 1328, 1340
trie_ptr: 950, 955, 967, 1340
trie_r: 950, 951, 952, 958, 959, 960, 962, 966, 967, 1340
trie_ref: 953, 955, 956, 959, 960, 962
trie_root: 950, 952, 955, 961, 969, 1340
trie_size: 32, 951, 955, 957, 967, 1328, 1335, 1340
trie_taken: 953, 955, 956, 957, 959, 1340
trie_trc: 924, 1327, 1328, 1340
trie_trl: 924, 1327, 1328, 1340
trie_tro: 924, 953, 1327, 1328, 1340
trie_used: 946, 947, 948, 949, 1327, 1328
true: 4, 16, 31, 37, 45, 46, 49, 51, 53, 71, 77, 88, 97, 98, 104, 105, 106, 107, 168, 169, 238, 256, 257, 259, 311, 327, 328, 336, 346, 355, 361, 362, 365, 377, 381, 410, 416, 417, 433, 443, 447, 450, 456, 464, 465, 475, 489, 504, 511, 515, 519, 527, 528, 529, 537, 557, 566, 581, 595, 624, 631, 640, 641, 644, 666, 678, 709, 722, 794, 830, 831, 832, 854, 857, 866, 883, 885, 887, 906, 908, 913, 914, 954, 959, 965, 966, 995, 1023, 1024, 1028, 1033, 1038, 1040, 1043, 1054, 1057, 1086, 1093, 1104, 1124, 1166, 1197, 1198, 1221, 1244, 1256, 1261, 1273, 1282, 1286, 1301, 1306, 1339, 1345, 1357, 1373, 1374, 1377, 1383, 1396, 1404, 1422, 1424, 1487, 1493, 1494, 1524, 1528, 1529, 1574
`true`: 456
try_break: 831, 832, 842, 854, 861, 865, 869, 871, 872, 876, 882
twenty_wchar_offset: 1421, 1423
two: 101, 102
two_choices: 113
two_halves: 118, 124, 172, 221, 256, 687, 1311, 1335

type: 4, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 152, 153, 155, 156, 157, 158, 159, 160, 175, 183, 184, 202, 206, 427, 492, 498, 499, 500, 508, 625, 626, 629, 631, 634, 635, 638, 640, 643, 652, 654, 656, 658, 671, 672, 673, 683, 684, 685, 686, 689, 690, 691, 692, 699, 701, 716, 718, 723, 724, 729, 730, 731, 732, 734, 735, 739, 750, 753, 755, 764, 765, 770, 771, 799, 802, 804, 808, 810, 812, 813, 814, 819, 822, 823, 825, 833, 835, 840, 844, 845, 846, 847, 848, 859, 861, 862, 863, 864, 865, 867, 868, 869, 871, 873, 874, 877, 878, 882, 884, 899, 900, 902, 906, 917, 971, 973, 975, 976, 979, 981, 982, 984, 989, 991, 996, 999, 1000, 1003, 1007, 1011, 1012, 1013, 1014, 1016, 1017, 1024, 1077, 1083, 1084, 1090, 1103, 1104, 1108, 1113, 1116, 1124, 1150, 1158, 1161, 1162, 1166, 1168, 1171, 1184, 1188, 1189, 1194, 1205, 1206, 1344, 1352, 1450
Type <return> to proceed...: 85
type_code: 207, 230, 1447
u: 69, 107, 392, 563, 709, 794, 803, 932, 937, 947, 1260, 1482, 1524
u_flag: 1486, 1487, 1493
u_part: 771, 772, 782, 791, 797, 804
u_template: 307, 314, 324, 791
uc_code: 230, 232, 410
\uccode primitive: 1233
uc_code_base: 230, 235, 1233, 1234, 1289, 1291
uc_hyph: 236, 894, 899
\uchyph primitive: 238
uc_hyph_code: 236, 237, 238
ucharcast: 526
\PUXucnumber primitive: 471
ucnumber_code: 471, 472, 474, 475
uexit: 81
un_hbox: 208, 1093, 1110, 1111, 1112
\unhbox primitive: 1110
\unhcopy primitive: 1110
\unkern primitive: 1110
\unpenalty primitive: 1110
\unskip primitive: 1110
un_vbox: 208, 1049, 1097, 1110, 1111, 1112
\unvbox primitive: 1110
\unvcopy primitive: 1110
unbalance: 392, 394, 399, 402, 476, 480
Unbalanced output routine: 1030
Unbalanced write...: 1375
Undefined control sequence: 373
undefined_cfont: 1512, 1527, 1538
undefined_control_sequence: 222, 232, 259, 262, 268, 282, 1311, 1321, 1322, 1335
undefined_cs: 210, 222, 366, 375, 1229, 1230, 1298, 1311
under_noad: 690, 693, 699, 701, 736, 764, 1159, 1160
Underfull \hbox...: 663
Underfull \vbox...: 677
underline: 1474, 1493
\underline primitive: 1159
undet_glue_ptr: 1448, 1454, 1455, 1456
undump: 1309, 1315, 1317, 1322, 1328, 1330
undump_checked_things: 1313, 1326
undump_end: 1309
undump_end_end: 1309
undump_four_ASCII: 1313
undump_hh: 1322
undump_int: 1309, 1311, 1315, 1320, 1322, 1328, 1330, 1404, 1577, 1581
undump_qqqq: 1313
undump_size: 1309, 1313, 1324, 1328, 1577, 1579, 1581
undump_size_end: 1309
undump_size_end_end: 1309
undump_things: 1311, 1313, 1315, 1320, 1322, 1324, 1326, 1328, 1390
undump_upper_check_things: 1326, 1328
unfloat: 109, 661, 667, 676, 679, 813, 814
unhyphenated: 822, 832, 840, 867, 869, 871
unity: 101, 103, 114, 164, 186, 456, 571, 1262, 1441, 1511
unpackage: 1112, 1113
unsave: 281, 283, 794, 803, 1029, 1066, 1071, 1089, 1103, 1122, 1136, 1171, 1177, 1189, 1194, 1197, 1199, 1203
unset_node: 136, 159, 175, 183, 184, 202, 206, 654, 672, 685, 691, 692, 771, 799, 802, 804, 808
unsigned: 1326
unspecified_mode: 73, 74, 1330
update_active: 864
update_heights: 973, 975, 976, 997, 1000, 1003
update_terminal: 34, 37, 61, 71, 81, 86, 362, 527, 540, 641, 1283, 1341
update_width: 835, 863
upper_cdigit_base: 471, 475
\uppercase primitive: 1289
Use of x doesn't match...: 401
use_err_help: 79, 80, 89, 90, 1286
v: 69, 107, 392, 453, 709, 718, 739, 746, 752, 803, 833, 925, 937, 947, 963, 980, 1141
v_flag: 1486, 1487, 1493
v_offset: 247, 643, 644
\voffset primitive: 248
v_offset_code: 247, 248

v_part: 771, 772, 782, 792, 797, 804
v_template: 307, 314, 325, 393, 792, 1134
vacuous: 443, 447, 448
vadjust: 208, 265, 266, 1100, 1101, 1102, 1103
\vadjust primitive: 265
valign: 208, 265, 266, 1049, 1093, 1133
\valign primitive: 265
var_code: 232, 1154, 1158, 1168
var_delimiter: 709, 740, 751, 765
var_used: 117, 125, 130, 164, 642, 1314, 1315
vbadness: 236, 677, 680, 681, 1015, 1020
\vbadness primitive: 238
vbadness_code: 236, 237, 238
\vbox primitive: 1074
vbox_group: 269, 1086, 1088
vcenter: 208, 265, 266, 1049, 1170
\vcenter primitive: 265
vcenter_group: 269, 1170, 1171
vcenter_noad: 690, 693, 699, 701, 736, 764, 1171
version_string: 61, 539
vert_break: 973, 974, 979, 980, 983, 985, 1013
very_loose_fit: 820, 822, 833, 836, 837, 839, 855
vet_glue: 628, 637
\vfil primitive: 1061
\vfilneg primitive: 1061
\vfill primitive: 1061
vfuzz: 247, 680, 1015, 1020
\vfuzz primitive: 248
vfuzz_code: 247, 248
VIRTEX: 1334
virtual memory: 126
Vitter, Jeffrey Scott: 261
vlist_node: 137, 148, 159, 175, 183, 184, 202, 206, 508, 621, 625, 626, 631, 632, 634, 635, 640, 643, 647, 654, 671, 672, 684, 716, 718, 723, 739, 750, 753, 810, 812, 814, 844, 845, 869, 873, 874, 971, 976, 981, 1003, 1077, 1083, 1090, 1113, 1150
vlist_out: 595, 618, 619, 621, 622, 626, 631, 632, 635, 640, 641, 643, 696, 1376
vmode: 211, 215, 419, 420, 421, 425, 427, 504, 778, 788, 789, 807, 810, 811, 812, 815, 1028, 1032, 1048, 1049, 1051, 1059, 1060, 1074, 1075, 1076, 1079, 1081, 1082, 1083, 1086, 1093, 1094, 1097, 1101, 1102, 1106, 1108, 1112, 1113, 1114, 1133, 1170, 1246, 1247
vmove: 208, 1051, 1074, 1075, 1076
vpack: 236, 647, 648, 649, 671, 708, 738, 741, 762, 802, 807, 980, 1024, 1103, 1171
vpackage: 671, 799, 980, 1020, 1089
vrule: 208, 265, 266, 466, 1059, 1087, 1093
\vrule primitive: 265
vsize: 247, 983, 990
\vsize primitive: 248
vsize_code: 247, 248
vskip: 208, 1049, 1060, 1061, 1062, 1081, 1097
\vskip primitive: 1061
vsplit: 970, 980, 981, 983, 1085
\vsplit needs a \vbox: 981
\vsplit primitive: 1074
vsplit_code: 1074, 1075, 1082
\vss primitive: 1061
\vtop primitive: 1074
vtop_code: 1074, 1075, 1086, 1088, 1089
vtop_group: 269, 1086, 1088
w: 114, 147, 156, 275, 278, 279, 610, 652, 671, 709, 718, 741, 794, 803, 909, 997, 1126, 1141, 1201, 1352, 1353, 1482
w_close: 1332, 1340
w_make_name_string: 528, 1331
w_open_in: 527
w_open_out: 1331
wait: 1015, 1023, 1024, 1025
wake_up_terminal: 34, 37, 51, 71, 73, 363, 487, 527, 533, 1297, 1300, 1306, 1311, 1336, 1341
warning_index: 305, 331, 338, 392, 393, 398, 399, 401, 404, 476, 482, 485, 777, 780
warning_issued: 76, 81, 245, 1338
was_free: 165, 167, 171
was_hi_min: 165, 166, 167, 171
was_lo_max: 165, 166, 167, 171
was_mem_end: 165, 166, 167, 171
wchr_cmd: 298
\wd primitive: 419
WEB: 1, 4, 38, 40, 50, 1311
web2c_int_base: 236
web2c_int_pars: 236
weight: 1482, 1483, 1492, 1496, 1497
what_lang: 1344, 1359, 1365, 1379, 1380
what_lhm: 1344, 1359, 1365, 1379, 1380
what_rhm: 1344, 1359, 1365, 1379, 1380
whatsit_node: 146, 148, 175, 183, 202, 206, 625, 634, 654, 672, 733, 764, 869, 899, 902, 971, 976, 1003, 1150, 1344, 1352
widow_penalty: 236, 1099
\widowpenalty primitive: 238
widow_penalty_code: 236, 237, 238
width: 466
width: 135, 136, 138, 139, 147, 150, 151, 155, 156, 178, 184, 187, 191, 192, 427, 432, 434, 454, 465, 466, 557, 608, 610, 614, 625, 626, 628, 629, 634, 636, 637, 638, 644, 654, 656, 659, 660, 669, 671, 672, 673, 674, 682, 686, 691, 709, 712, 717, 718, 719, 720, 734, 741, 747, 750, 752, 753, 760, 761, 762, 771, 782, 796, 799, 800, 801, 804, 805,

806, 807, 809, 810, 811, 812, 813, 814, 830, 840, 841, 844, 845, 869, 871, 873, 874, 884, 972, 979, 999, 1004, 1007, 1012, 1045, 1047, 1057, 1094, 1096, 1150, 1151, 1202, 1204, 1208, 1232, 1242, 1243, 1505, 1506, 1558
width_base: 553, 557, 569, 572, 574, 579, 1325, 1326, 1340
width_index: 546, 553
width_offset: 135, 419, 420, 1250
width_value: 1551, 1552, 1553, 1554, 1563
Wirth, Niklaus: 10
wlog: 56, 58, 537, 539, 1337
wlog_cr: 56, 57, 58, 537, 539, 1336
wlog_ln: 56, 1337
word_define: 1217, 1227, 1231, 1235, 1239, 1477, 1544, 1546, 1547
word_file: 25, 113, 528, 1308
words: 204, 205, 206, 1360
wrap_lig: 913, 914
wrapup: 1038, 1043
write: 37, 56, 58, 600, 1309
\write primitive: 1347
write_dvi: 600, 601, 602, 643
write_file: 57, 58, 1345, 1377, 1381
write_ln: 37, 51, 56, 57, 1309, 1385
write_loc: 1316, 1317, 1347, 1348, 1374
write_node: 1344, 1347, 1349, 1351, 1359, 1360, 1361, 1376, 1377
write_node_size: 1344, 1353, 1355, 1356, 1357, 1360, 1361, 1406
write_open: 1345, 1346, 1373, 1377, 1381
write_out: 1373, 1377
write_stream: 1344, 1353, 1357, 1358, 1373, 1377, 1406
write_text: 307, 314, 323, 1343, 1374
write_tokens: 1344, 1355, 1356, 1357, 1359, 1360, 1361, 1371, 1374, 1406
writing: 581
wt: 588
wterm: 56, 58, 61, 527
wterm_cr: 56, 57, 58
wterm_ln: 56, 61, 527, 1306, 1311, 1335, 1340
Wyatt, Douglas Kirk: 2
w0: 588, 589, 607, 612
w1: 588, 589, 610
w2: 588
w3: 588
w4: 588
x: 100, 105, 106, 107, 590, 603, 652, 671, 709, 723, 729, 738, 740, 741, 746, 752, 759, 1126, 1305, 1306, 1441
x_height: 550, 561, 562, 741, 1126, 1402
x_height_code: 550, 561
x_leaders: 149, 190, 630, 1074, 1075
\xleaders primitive: 1074
x_over_n: 106, 706, 719, 720, 989, 1011, 1012, 1013, 1243
x_token: 364, 384, 481, 1155, 1453, 1460
xchr: 20, 21, 23, 24, 38, 49, 58, 522, 1373, 1389, 1390
xclause: 16
\xdef primitive: 1211
xeq_level: 253, 254, 268, 278, 279, 283, 1307
xmalloc_array: 51, 522, 526, 1310, 1311, 1313, 1324, 1326, 1328, 1335, 1340
xn_over_d: 107, 458, 460, 461, 571, 719, 1047, 1263, 1505, 1506, 1511
xord: 20, 24, 52, 53, 526, 528, 1389, 1390
xpand: 476, 480, 482
xprn: 20, 24, 1389, 1390
xray: 208, 1293, 1294, 1295
xspace_skip: 224, 1046
\xspaceskip primitive: 226
xspace_skip_code: 224, 225, 226, 1046
xxx1: 588, 589, 1371
xxx2: 588
xxx3: 588
xxx4: 588, 589, 1371
x0: 588, 589, 607, 612
x1: 588, 589, 610
x2: 588
x3: 588
x4: 588
y: 105, 709, 729, 738, 740, 741, 746, 752, 759
y_here: 611, 612, 614, 615, 616
y_OK: 611, 612, 615
y_seen: 614, 615
year: 236, 241, 539, 620, 1331
\year primitive: 238
year_code: 236, 237, 238
yhash: 256, 1311, 1335
You already have nine...: 479
You can't \insert255: 1102
You can't dump...: 1307
You can't use \hrule...: 1098
You can't use \long...: 1216
You can't use a prefix with x: 1215
You can't use x after ...: 431, 1240
You can't use x in y mode: 1052
You have to increase POOLSIZE: 52
you_cant: 1052, 1053, 1083, 1109
yz_OK: 611, 612, 613, 615
yzmem: 116, 1311, 1335
y0: 588, 589, 597, 607, 612

y1: 588, 589, 610, 616
y2: 588, 597
y3: 588
y4: 588
z: 563, 709, 729, 746, 752, 759, 925, 930, 956, 962, 1201, 1441
z_here: 611, 612, 614, 615, 617
z_OK: 611, 612, 615
z_seen: 614, 615
Zabala Salelles, Ignacio Andrés: 2
zeqtb: 253, 1311, 1335, 1340
zero_glue: 162, 175, 224, 228, 427, 465, 735, 805, 890, 1044, 1045, 1046, 1174, 1232, 1452, 1453, 1457, 1507, 1514
zero_token: 448, 455, 476, 479, 482
zmem: 116, 1311, 1335
z0: 588, 589, 607, 612
z1: 588, 589, 610, 617
z2: 588
z3: 588
z4: 588

⟨ Accumulate the constant until *cur_tok* is not a suitable digit 448 ⟩ Used in section 447.
⟨ Add the width of node *s* to *act_width* 874 ⟩ Used in section 872.
⟨ Add the width of node *s* to *break_width* 845 ⟩ Used in section 843.
⟨ Add the width of node *s* to *disc_width* 873 ⟩ Used in section 872.
⟨ Add this face matching 1529 ⟩ Used in section 1526.
⟨ Adjust for the magnification ratio 460 ⟩ Used in section 456.
⟨ Adjust for the setting of `\globaldefs` 1217 ⟩ Used in section 1214.
⟨ Adjust *shift_up* and *shift_down* for the case of a fraction line 749 ⟩ Used in section 746.
⟨ Adjust *shift_up* and *shift_down* for the case of no fraction line 748 ⟩ Used in section 746.
⟨ Advance *cur_p* to the node following the present string of characters 870 ⟩ Used in section 869.
⟨ Advance past a whatsit node in the *line_break* loop 1365 ⟩ Used in section 869.
⟨ Advance past a whatsit node in the pre-hyphenation loop 1366 ⟩ Used in section 899.
⟨ Advance r; **goto** *found* if the parameter delimiter has been fully matched, otherwise **goto** *continue* 397 ⟩ Used in section 395.
⟨ Allocate entire node p and **goto** *found* 129 ⟩ Used in section 127.
⟨ Allocate from the top of node p and **goto** *found* 128 ⟩ Used in section 127.
⟨ Apologize for inability to do the operation now, unless `\unskip` follows non-glue 1109 ⟩ Used in section 1108.
⟨ Apologize for not loading the font, **goto** *done* 570 ⟩ Used in section 569.
⟨ Append a ligature and/or kern to the translation; **goto** *continue* if the stack of inserted ligatures is nonempty 913 ⟩ Used in section 909.
⟨ Append a new leader node that uses *cur_box* 1081 ⟩ Used in section 1078.
⟨ Append a new letter or a hyphen level 965 ⟩ Used in section 964.
⟨ Append a new letter or hyphen 940 ⟩ Used in section 938.
⟨ Append a normal inter-word space to the current list, then **goto** *big_switch* 1044 ⟩ Used in section 1033.
⟨ Append a penalty node, if a nonzero penalty is appropriate 893 ⟩ Used in section 883.
⟨ Append an insertion to the current page and **goto** *contribute* 1011 ⟩ Used in section 1003.
⟨ Append any *new_hlist* entries for q, and any appropriate penalties 770 ⟩ Used in section 763.
⟨ Append box *cur_box* to the current list, shifted by *box_context* 1079 ⟩ Used in section 1078.
⟨ 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 1037 ⟩ Used in section 1033.
⟨ Append characters of $hu[j\,..]$ to *major_tail*, advancing j 920 ⟩ Used in section 919.
⟨ Append double-byte character *cur_chr* and the following double-byte characters (if any) to the current hlist in the current font; **goto** *main_loop* when a single-byte character has been fetched; **goto** *reswitch* when a non-character has been fetched 1453 ⟩ Used in section 1033.
⟨ Append inter-element spacing based on *r_type* and t 769 ⟩ Used in section 763.
⟨ Append tabskip glue and an empty box to list u, and update s and t as the prototype nodes are passed 812 ⟩ Used in section 811.
⟨ Append the accent with appropriate kerns, then set $p \leftarrow q$ 1128 ⟩ Used in section 1126.
⟨ Append the current tabskip glue to the preamble list 781 ⟩ Used in section 780.
⟨ Append the display and perhaps also the equation number 1207 ⟩ Used in section 1202.
⟨ Append the glue or equation number following the display 1208 ⟩ Used in section 1202.
⟨ Append the glue or equation number preceding the display 1206 ⟩ Used in section 1202.
⟨ Append the new box to the current vertical list, followed by the list of special nodes taken out of the box by the packager 891 ⟩ Used in section 883.
⟨ Append the value n to list p 941 ⟩ Used in section 940.
⟨ Assign the values *depth_threshold* ← *show_box_depth* and *breadth_max* ← *show_box_breadth* 236 ⟩ Used in section 198.
⟨ Assignments 1220, 1221, 1224, 1227, 1228, 1229, 1231, 1235, 1237, 1238, 1244, 1245, 1251, 1255, 1256, 1259, 1267, 1416, 1427, 1430, 1471, 1477, 1522, 1534, 1539, 1541, 1544, 1546, 1550, 1558, 1563, 1567 ⟩ Used in section 1214.
⟨ Attach list p to the current list, and record its length; then finish up and **return** 1123 ⟩ Used in section 1122.
⟨ Attach the limits to y and adjust *height*(v), *depth*(v) to account for their presence 754 ⟩ Used in section 753.
⟨ Back up an outer control sequence so that it can be reread 337 ⟩ Used in section 336.

⟨ Basic printing procedures 57, 58, 59, 60, 62, 63, 64, 65, 262, 263, 521, 702, 1358, 1385, 1387, 1422, 1423, 1424, 1434 ⟩ 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 1020 ⟩ Used in section 1017.
⟨ Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list 879 ⟩ Used in section 818.
⟨ Build a CJK font according to *cur_chr* and *cface_num* if it is not exist 1538 ⟩ Used in sections 1537 and 1537.
⟨ Calculate the length, l, and the shift amount, s, of the display lines 1152 ⟩ Used in section 1148.
⟨ 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 1149 ⟩ Used in section 1148.
⟨ Call the packaging subroutine, setting *just_box* to the justified box 892 ⟩ Used in section 883.
⟨ 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 869 ⟩ Used in section 866.
⟨ Carry out a ligature replacement, updating the cursor structure and possibly advancing j; **goto** *continue* if the cursor doesn't advance, otherwise **goto** *done* 914 ⟩ Used in section 912.
⟨ Case statement to copy different types and set *words* to the number of initial words not yet copied 206 ⟩ Used in section 205.
⟨ Cases for noads that can follow a *bin_noad* 736 ⟩ Used in section 731.
⟨ Cases for nodes that can appear in an mlist, after which we **goto** *done_with_node* 733 ⟩ Used in section 731.
⟨ Cases of *flush_node_list* that arise in mlists only 701 ⟩ Used in section 202.
⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1088, 1103, 1121, 1135, 1136, 1171, 1176, 1189 ⟩ Used in section 1071.
⟨ Cases of *main_control* that are for extensions to TEX 1350 ⟩ Used in section 1048.
⟨ Cases of *main_control* that are not part of the inner loop 1048 ⟩ Used in section 1033.
⟨ Cases of *main_control* that build boxes and lists 1059, 1060, 1066, 1070, 1076, 1093, 1095, 1097, 1100, 1105, 1107, 1112, 1115, 1119, 1125, 1129, 1133, 1137, 1140, 1143, 1153, 1157, 1161, 1165, 1167, 1170, 1174, 1178, 1183, 1193, 1196, 1445 ⟩ Used in section 1048.
⟨ Cases of *main_control* that don't depend on *mode* 1213, 1271, 1274, 1277, 1279, 1288, 1293 ⟩ Used in section 1048.
⟨ Cases of *main_control* that handle spacer 1461 ⟩ Used in section 1033.
⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227, 231, 239, 249, 266, 335, 380, 388, 415, 420, 472, 491, 495, 784, 987, 1056, 1062, 1075, 1092, 1111, 1118, 1146, 1160, 1173, 1182, 1192, 1212, 1223, 1226, 1234, 1254, 1258, 1264, 1266, 1276, 1281, 1290, 1295, 1298, 1349, 1426, 1443, 1446, 1466, 1470, 1476, 1515, 1521, 1543, 1549, 1557, 1562, 1566 ⟩ Used in section 298.
⟨ Cases of *show_node_list* that arise in mlists only 693 ⟩ Used in section 183.
⟨ Cases where character is ignored 345 ⟩ Used in section 344.
⟨ Change buffered instruction to y or w and **goto** *found* 616 ⟩ Used in section 615.
⟨ Change buffered instruction to z or x and **goto** *found* 617 ⟩ Used in section 615.
⟨ Change current mode to $-vmode$ for `\halign`, $-hmode$ for `\valign` 778 ⟩ Used in section 777.
⟨ Change discretionary to compulsory and set $disc\_break \leftarrow true$ 885 ⟩ Used in section 884.
⟨ Change font *dvi_cf* to f 1574 ⟩ Used in section 623.
⟨ Change font *dvi_f* to f 624 ⟩ Used in section 623.
⟨ Change state if necessary, and **goto** *switch* if the current character should be ignored, or **goto** *reswitch* if the current character changes to another 344 ⟩ Used in section 343.
⟨ Change the case of the token in p, if a change is appropriate 1292 ⟩ Used in section 1291.
⟨ Change the current style and **goto** *delete_q* 766 ⟩ Used in section 764.
⟨ 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* 734 ⟩ Used in section 733.
⟨ Character k cannot be printed 49 ⟩ Used in section 48.
⟨ Character s is the current new-line character 244 ⟩ Used in sections 58 and 59.

⟨ Check flags of unavailable nodes 170 ⟩ Used in section 167.
⟨ Check for charlist cycle 573 ⟩ Used in section 572.
⟨ Check for improper alignment in displayed math 779 ⟩ Used in section 777.
⟨ 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 977 ⟩ Used in section 975.
⟨ 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* 1008 ⟩ Used in section 1000.
⟨ Check single-word *avail* list 168 ⟩ Used in section 167.
⟨ Check that another `$` follows 1200 ⟩ Used in sections 1197, 1197, and 1209.
⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1198 ⟩ Used in sections 1197 and 1197.
⟨ Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto** *done1* 902 ⟩ Used in section 897.
⟨ Check the "constant" values for consistency 14, 111, 290, 525, 1252 ⟩ Used in section 1335.
⟨ Check the lookahead character 1459 ⟩ Used in sections 1453 and 1453.
⟨ Check the pool check sum 53 ⟩ Used in section 52.
⟨ Check variable-size *avail* list 169 ⟩ Used in section 167.
⟨ Clean up the memory by removing the break nodes 868 ⟩ Used in sections 818 and 866.
⟨ Clear dimensions to zero 653 ⟩ Used in sections 652 and 671.
⟨ Clear off top level from *save_stack* 282 ⟩ Used in section 281.
⟨ Close the format file 1332 ⟩ Used in section 1305.
⟨ Coerce glue to a dimension 454 ⟩ Used in sections 452 and 458.
⟨ Compiler directives 9 ⟩ Used in section 4.
⟨ Complain about an undefined family and set *cur_i* null 726 ⟩ Used in section 725.
⟨ Complain about an undefined macro 373 ⟩ Used in section 370.
⟨ Complain about missing `\endcsname` 376 ⟩ Used in section 375.
⟨ Complain about unknown unit and **goto** *done2* 462 ⟩ Used in section 461.
⟨ Complain that `\the` can't do this; give zero result 431 ⟩ Used in section 416.
⟨ Complain that the user should have said `\mathaccent` 1169 ⟩ Used in section 1168.
⟨ Compleat the incompleat noad 1188 ⟩ Used in section 1187.
⟨ Complete a potentially long `\show` command 1301 ⟩ Used in section 1296.
⟨ Compute result of *multiply* or *divide*, put it in *cur_val* 1243 ⟩ Used in section 1239.
⟨ Compute result of *register* or *advance*, put it in *cur_val* 1241 ⟩ Used in section 1239.
⟨ Compute the amount of skew 744 ⟩ Used in section 741.
⟨ Compute the badness, b, of the current page, using *awful_bad* if the box is too full 1010 ⟩ Used in section 1008.
⟨ Compute the badness, b, using *awful_bad* if the box is too full 978 ⟩ Used in section 977.
⟨ Compute the demerits, d, from r to *cur_p* 862 ⟩ Used in section 858.
⟨ Compute the discretionary *break_width* values 843 ⟩ Used in section 840.
⟨ Compute the hash code h 261 ⟩ Used in section 259.
⟨ Compute the magic offset 768 ⟩ Used in section 1340.
⟨ Compute the minimum suitable height, w, and the corresponding number of extension steps, n; also set *width*(b) 717 ⟩ Used in section 716.
⟨ Compute the new line width 853 ⟩ Used in section 838.
⟨ Compute the register location l and its type p; but **return** if invalid 1240 ⟩ Used in section 1239.
⟨ Compute the sum of two glue specs 1242 ⟩ Used in section 1241.
⟨ Compute the trie op code, v, and set $l \leftarrow 0$ 968 ⟩ Used in section 966.
⟨ Compute the values of *break_width* 840 ⟩ Used in section 839.
⟨ Consider a node with matching width; **goto** *found* if it's a hit 615 ⟩ Used in section 614.
⟨ 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 854 ⟩

Used in section 832.
⟨ Constants in the outer block 11, 1472, 1499 ⟩ Used in section 4.
⟨ Construct a box with limits above and below it, skewed by *delta* 753 ⟩ Used in section 752.
⟨ Construct a sub/superscript combination box x, with the superscript offset by *delta* 762 ⟩ Used in section 759.
⟨ Construct a subscript box x when there is no superscript 760 ⟩ Used in section 759.
⟨ Construct a superscript box x 761 ⟩ Used in section 759.
⟨ Construct a vlist box for the fraction, according to *shift_up* and *shift_down* 750 ⟩ Used in section 746.
⟨ Construct an extensible character in a new box b, using recipe *rem_byte*(q) and font f 716 ⟩ Used in section 713.
⟨ Contribute an entire group to the current parameter 402 ⟩ Used in section 395.
⟨ Contribute the recently matched tokens to the current parameter, and **goto** *continue* if a partial match is still in effect; but abort if $s = null$ 400 ⟩ Used in section 395.
⟨ Convert a final *bin_noad* to an *ord_noad* 732 ⟩ Used in sections 729 and 731.
⟨ Convert *cur_val* to a lower level 432 ⟩ Used in section 416.
⟨ Convert math glue to ordinary glue 735 ⟩ Used in section 733.
⟨ Convert *nucleus*(q) to an hlist and attach the sub/superscripts 757 ⟩ Used in section 731.
⟨ Copy the tabskip glue between columns 798 ⟩ Used in section 794.
⟨ Copy the templates from node *cur_loop* into node p 797 ⟩ Used in section 796.
⟨ Copy the token list 469 ⟩ Used in section 468.
⟨ 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 758 ⟩ Used in section 757.
⟨ Create a character node q for the next character, but set $q \leftarrow null$ if problems arise 1127 ⟩ Used in section 1126.
⟨ Create a new glue specification whose width is *cur_val*; scan for its stretch and shrink components 465 ⟩ Used in section 464.
⟨ Create a page insertion node with $subtype(r) = qi(n)$, and include the glue correction for box n in the current page state 1012 ⟩ Used in section 1011.
⟨ Create an active breakpoint representing the beginning of the paragraph 867 ⟩ Used in section 866.
⟨ Create and append a discretionary node as an alternative to the unhyphenated word, and continue to develop both branches until they become equivalent 917 ⟩ Used in section 916.
⟨ 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 747 ⟩ Used in section 746.
⟨ Create new active nodes for the best feasible breaks just found 839 ⟩ Used in section 838.
⟨ Create the *format_ident*, open the format file, and inform the user that dumping has begun 1331 ⟩ Used in section 1305.
⟨ Current *mem* equivalent of glue parameter number n 224 ⟩ Used in sections 152 and 154.
⟨ Deactivate node r 863 ⟩ Used in section 854.
⟨ Declare PUTeX subprocedures for *prefixed_command* 1505, 1506, 1525 ⟩ Used in section 1214.
⟨ Declare action procedures for use by *main_control* 1046, 1050, 1052, 1053, 1054, 1057, 1063, 1064, 1067, 1072, 1073, 1078, 1082, 1087, 1089, 1094, 1096, 1098, 1099, 1102, 1104, 1106, 1108, 1113, 1116, 1120, 1122, 1126, 1130, 1132, 1134, 1138, 1139, 1141, 1145, 1154, 1158, 1162, 1163, 1166, 1168, 1175, 1177, 1179, 1184, 1194, 1197, 1203, 1214, 1273, 1278, 1282, 1291, 1296, 1305, 1351, 1379, 1406 ⟩ Used in section 1033.
⟨ Declare additional functions for MLTEX 1396, 1397 ⟩ Used in section 563.
⟨ Declare additional routines for string recycling 1391, 1392 ⟩ Used in section 47.
⟨ Declare math construction procedures 737, 738, 739, 740, 741, 746, 752, 755, 759, 765 ⟩ Used in section 729.
⟨ Declare procedures for preprocessing hyphenation patterns 947, 951, 952, 956, 960, 962, 963, 969 ⟩ Used in section 945.
⟨ Declare procedures needed for displaying the elements of mlists 694, 695, 697 ⟩ Used in section 179.
⟨ Declare procedures needed in *do_extension* 1352, 1353 ⟩ Used in section 1351.
⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1371, 1373, 1376 ⟩ Used in section 622.
⟨ Declare procedures that scan font-related stuff 580, 581 ⟩ Used in section 412.

⟨ Declare procedures that scan restricted classes of integers 436, 437, 438, 439, 440, 1388, 1419 ⟩ Used in section 412.
⟨ Declare subprocedures for *line_break* 829, 832, 880, 898, 945 ⟩ Used in section 818.
⟨ Declare subprocedures for *prefixed_command* 1218, 1232, 1239, 1246, 1247, 1248, 1249, 1250, 1260, 1268, 1482, 1524, 1526, 1530, 1535 ⟩ Used in section 1214.
⟨ Declare subprocedures for *var_delimiter* 712, 714, 715 ⟩ Used in section 709.
⟨ Declare the function called *fin_mlist* 1187 ⟩ Used in section 1177.
⟨ Declare the function called *find_cface_num* 1481 ⟩ Used in section 1260.
⟨ Declare the function called *find_ec_num* 1523 ⟩ Used in section 1526.
⟨ Declare the function called *fw_times_sd* 1441 ⟩ Used in section 1260.
⟨ Declare the function called *open_fmt_file* 527 ⟩ Used in section 1306.
⟨ Declare the function called *print_fixword* 1440 ⟩ Used in section 1482.
⟨ Declare the function called *reconstitute* 909 ⟩ Used in section 898.
⟨ Declare the procedure called *align_peek* 788 ⟩ Used in section 803.
⟨ Declare the procedure called *check_cfont* 1512 ⟩ Used in section 1260.
⟨ Declare the procedure called *fire_up* 1015 ⟩ Used in section 997.
⟨ Declare the procedure called *get_preamble_token* 785 ⟩ Used in section 777.
⟨ Declare the procedure called *handle_right_brace* 1071 ⟩ Used in section 1033.
⟨ Declare the procedure called *init_span* 790 ⟩ Used in section 789.
⟨ Declare the procedure called *insert_relax* 382 ⟩ Used in section 369.
⟨ Declare the procedure called *macro_call* 392 ⟩ Used in section 369.
⟨ Declare the procedure called *make_cfont* 1514 ⟩ Used in section 1260.
⟨ Declare the procedure called *print_cmd_chr* 298 ⟩ Used in section 252.
⟨ Declare the procedure called *print_skip_param* 225 ⟩ Used in section 179.
⟨ Declare the procedure called *restore_trace* 284 ⟩ Used in section 281.
⟨ Declare the procedure called *runaway* 306 ⟩ Used in section 119.
⟨ Declare the procedure called *show_token_list* 292 ⟩ Used in section 119.
⟨ Decry the invalid character and **goto** *restart* 346 ⟩ Used in section 344.
⟨ Define a CJK font and then goto *common_ending* 1509 ⟩ Used in section 1260.
⟨ Define the *cur_cfont* according to *cur_font* and *cface_num* 1527 ⟩ Used in sections 1526, 1535, and 1541.
⟨ Delete $c -$ "0" tokens and **goto** *continue* 88 ⟩ Used in section 84.
⟨ Delete the page-insertion nodes 1022 ⟩ Used in section 1017.
⟨ Destroy the t nodes following q, and make r point to the following node 886 ⟩ Used in section 885.
⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 667 ⟩ Used in section 660.
⟨ Determine horizontal glue stretch setting, then **return** or **goto** *common_ending* 661 ⟩ Used in section 660.
⟨ Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming that $l = false$ 1205 ⟩ Used in section 1202.
⟨ Determine the shrink order 668 ⟩ Used in sections 667, 679, and 799.
⟨ Determine the stretch order 662 ⟩ Used in sections 661, 676, and 799.
⟨ Determine the value of *height*(r) and the appropriate glue setting; then **return** or **goto** *common_ending* 675 ⟩ Used in section 671.
⟨ Determine the value of *width*(r) and the appropriate glue setting; then **return** or **goto** *common_ending* 660 ⟩ Used in section 652.
⟨ Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 679 ⟩ Used in section 675.
⟨ Determine vertical glue stretch setting, then **return** or **goto** *common_ending* 676 ⟩ Used in section 675.
⟨ Discard erroneous prefixes and **return** 1215 ⟩ Used in section 1214.
⟨ Discard the prefixes \long and \outer if they are irrelevant 1216 ⟩ Used in section 1214.
⟨ Dispense with trivial cases of void or bad boxes 981 ⟩ Used in section 980.
⟨ Display adjustment p 197 ⟩ Used in section 183.
⟨ Display box p 184 ⟩ Used in section 183.
⟨ Display choice node p 698 ⟩ Used in section 693.
⟨ Display discretionary p 195 ⟩ Used in section 183.

⟨ Display fraction noad p 700 ⟩ Used in section 693.
⟨ Display glue p 189 ⟩ Used in section 183.
⟨ Display insertion p 188 ⟩ Used in section 183.
⟨ Display kern p 191 ⟩ Used in section 183.
⟨ Display leaders p 190 ⟩ Used in section 189.
⟨ Display ligature p 193 ⟩ Used in section 183.
⟨ Display mark p 196 ⟩ Used in section 183.
⟨ Display math node p 192 ⟩ Used in section 183.
⟨ Display node p 183 ⟩ Used in section 182.
⟨ Display normal noad p 699 ⟩ Used in section 693.
⟨ Display penalty p 194 ⟩ Used in section 183.
⟨ Display rule p 187 ⟩ Used in section 183.
⟨ Display special fields of the unset node p 185 ⟩ Used in section 184.
⟨ Display the current context 312 ⟩ Used in section 311.
⟨ Display the insertion split cost 1014 ⟩ Used in section 1013.
⟨ Display the page break cost 1009 ⟩ Used in section 1008.
⟨ Display the token (m, c) 294 ⟩ Used in section 293.
⟨ Display the value of b 505 ⟩ Used in section 501.
⟨ Display the value of *glue_set*(p) 186 ⟩ Used in section 184.
⟨ Display the whatsit node p 1359 ⟩ Used in section 183.
⟨ Display token p, and **return** if there are problems 293 ⟩ Used in section 292.
⟨ 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 731 ⟩ Used in section 730.
⟨ Do ligature or kern command, returning to *main_lig_loop* or *main_loop_wrapup* or *main_loop_move* 1043 ⟩ Used in section 1042.
⟨ Do magic computation 320 ⟩ Used in section 292.
⟨ Do some work that has been queued up for `\write` 1377 ⟩ Used in section 1376.
⟨ Drop current token and complain that it was unmatched 1069 ⟩ Used in section 1067.
⟨ Dump MLTEX-specific data 1403 ⟩ Used in section 1305.
⟨ Dump a couple more things and the closing check word 1329 ⟩ Used in section 1305.
⟨ Dump constants for consistency check 1310 ⟩ Used in section 1305.
⟨ Dump regions 1 to 4 of *eqtb* 1318 ⟩ Used in section 1316.
⟨ Dump regions 5 and 6 of *eqtb* 1319 ⟩ Used in section 1316.
⟨ Dump the CJK font face information 1576 ⟩ Used in section 1305.
⟨ Dump the CJK font information 1580 ⟩ Used in section 1305.
⟨ Dump the array info for internal font number k 1325 ⟩ Used in section 1323.
⟨ Dump the dynamic memory 1314 ⟩ Used in section 1305.
⟨ Dump the face matching table 1578 ⟩ Used in section 1305.
⟨ Dump the font information 1323 ⟩ Used in section 1305.
⟨ Dump the hash table 1321 ⟩ Used in section 1316.
⟨ Dump the hyphenation tables 1327 ⟩ Used in section 1305.
⟨ Dump the string pool 1312 ⟩ Used in section 1305.
⟨ Dump the table of equivalents 1316 ⟩ Used in section 1305.
⟨ Dump *xord*, *xchr*, and *xprn* 1389 ⟩ Used in section 1310.
⟨ Either append the insertion node p after node q, and remove it from the current page, or delete *node*(p) 1025 ⟩ Used in section 1023.
⟨ 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 1023 ⟩ Used in section 1017.
⟨ Either process `\ifcase` or set b to the value of a boolean condition 504 ⟩ Used in section 501.
⟨ Empty the last bytes out of *dvi_buf* 602 ⟩ Used in section 645.
⟨ Ensure that box 255 is empty after output 1031 ⟩ Used in section 1029.

⟨ Ensure that box 255 is empty before output 1018 ⟩ Used in section 1017.
⟨ Ensure that $trie\_max \ge h + 256$ 957 ⟩ Used in section 956.
⟨ Enter a hyphenation exception 942 ⟩ Used in section 938.
⟨ Enter all of the patterns into a linked trie, until coming to a right brace 964 ⟩ Used in section 963.
⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then **return** 938 ⟩ Used in section 937.
⟨ Enter *skip_blanks* state, emit a space 349 ⟩ Used in section 347.
⟨ Error handling procedures 78, 81, 82, 93, 94, 95 ⟩ Used in section 4.
⟨ 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 654 ⟩ Used in section 652.
⟨ 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 672 ⟩ Used in section 671.
⟨ Expand a nonmacro 370 ⟩ Used in section 369.
⟨ Expand macros in the token list and make *link*(*def_ref*) point to the result 1374 ⟩ Used in section 1373.
⟨ Expand the next part of the input 481 ⟩ Used in section 480.
⟨ Expand the token after the next token 371 ⟩ Used in section 370.
⟨ Explain that too many dead cycles have occurred in a row 1027 ⟩ Used in section 1015.
⟨ Express astonishment that no number was here 449 ⟩ Used in section 447.
⟨ Express consternation over the fact that no alignment is in progress 1131 ⟩ Used in section 1130.
⟨ Express shock at the missing left brace; **goto** *found* 478 ⟩ Used in section 477.
⟨ Feed the macro body and its parameters to the scanner 393 ⟩ Used in section 392.
⟨ Fetch a Chinese face id 1528 ⟩ Used in section 1526.
⟨ Fetch a box dimension 423 ⟩ Used in section 416.
⟨ Fetch a character code from some table 417 ⟩ Used in section 416.
⟨ Fetch a font dimension 428 ⟩ Used in section 416.
⟨ Fetch a font integer 429 ⟩ Used in section 416.
⟨ Fetch a register 430 ⟩ Used in section 416.
⟨ Fetch a token list or font identifier, provided that $level = tok\_val$ 418 ⟩ Used in section 416.
⟨ Fetch an internal dimension and **goto** *attach_sign*, or fetch an internal integer 452 ⟩ Used in section 451.
⟨ Fetch an item in the current node, if appropriate 427 ⟩ Used in section 416.
⟨ Fetch something on the *page_so_far* 424 ⟩ Used in section 416.
⟨ Fetch the Chinese face name 1510 ⟩ Used in section 1509.
⟨ Fetch the font design size and compute font 'at' size 1511 ⟩ Used in section 1509.
⟨ Fetch the *dead_cycles* or the *insert_penalties* 422 ⟩ Used in section 416.
⟨ Fetch the *par_shape* size 426 ⟩ Used in section 416.
⟨ Fetch the *prev_graf* 425 ⟩ Used in section 416.
⟨ Fetch the *space_factor* or the *prev_depth* 421 ⟩ Used in section 416.
⟨ Find an active node with fewest demerits 877 ⟩ Used in section 876.
⟨ Find hyphen locations for the word in *hc*, or **return** 926 ⟩ Used in section 898.
⟨ Find optimal breakpoints 866 ⟩ Used in section 818.
⟨ Find the best active node for the desired looseness 878 ⟩ Used in section 876.
⟨ Find the best way to split the insertion, and change *type*(r) to *split_up* 1013 ⟩ Used in section 1011.
⟨ Find the glue specification, *main_p*, for text spaces in the current font 1045 ⟩ Used in sections 1044 and 1046.
⟨ Finish an alignment in a display 1209 ⟩ Used in section 815.
⟨ Finish displayed math 1202 ⟩ Used in section 1197.
⟨ Finish issuing a diagnostic message for an overfull or underfull hbox 666 ⟩ Used in section 652.
⟨ Finish issuing a diagnostic message for an overfull or underfull vbox 678 ⟩ Used in section 671.
⟨ Finish line, emit a `\par` 351 ⟩ Used in section 347.
⟨ Finish line, emit a space 348 ⟩ Used in section 347.
⟨ Finish line, **goto** *switch* 350 ⟩ Used in section 347.
⟨ Finish math in text 1199 ⟩ Used in section 1197.
⟨ Finish the `DVI` file 645 ⟩ Used in section 1336.

⟨ Finish the extensions 1381 ⟩ Used in section 1336.
⟨ Fire up the user's output routine and **return** 1028 ⟩ Used in section 1015.
⟨ Fix the reference count, if any, and negate *cur_val* if *negative* 433 ⟩ Used in section 416.
⟨ Flush the box from memory, showing statistics if requested 642 ⟩ Used in section 641.
⟨ Forbidden cases detected in *main_control* 1051, 1101, 1114, 1147 ⟩ Used in section 1048.
⟨ Generate a *down* or *right* command for w and **return** 613 ⟩ Used in section 610.
⟨ Generate a *y0* or *z0* command in order to reuse a previous appearance of w 612 ⟩ Used in section 610.
⟨ Get ready to compress the trie 955 ⟩ Used in section 969.
⟨ Get ready to start line breaking 819, 830, 837, 851 ⟩ Used in section 818.
⟨ Get substitution information, check it, goto *found* if all is ok, otherwise goto *continue* 1400 ⟩ Used in section 1398.
⟨ Get the first line of input and prepare to start 1340 ⟩ Used in section 1335.
⟨ Get the next non-blank non-call token 409 ⟩ Used in sections 408, 444, 458, 506, 529, 580, 788, 794, 1048, 1418, 1477, 1487, 1493, 1528, 1535, 1535, 1550, and 1558.
⟨ Get the next non-blank non-relax non-call token 407 ⟩ Used in sections 406, 1081, 1087, 1154, 1163, 1214, 1229, and 1273.
⟨ Get the next non-blank non-sign token; set *negative* appropriately 444 ⟩ Used in sections 443, 451, and 464.
⟨ Get the next token, suppressing expansion 358 ⟩ Used in section 357.
⟨ Get user's advice and **return** 83 ⟩ Used in section 82.
⟨ Give diagnostic information, if requested 1034 ⟩ Used in section 1033.
⟨ Give improper `\hyphenation` error 939 ⟩ Used in section 938.
⟨ Give improper hyphenation error for Chinese characters inside 1444 ⟩ Used in section 938.
⟨ Global variables 13, 20, 26, 30, 32, 39, 50, 54, 73, 76, 79, 96, 104, 115, 116, 117, 118, 124, 165, 173, 181, 213, 246, 253, 256, 271, 286, 297, 301, 304, 305, 308, 309, 310, 333, 361, 367, 385, 390, 391, 413, 441, 450, 483, 492, 496, 515, 516, 523, 530, 535, 542, 552, 553, 558, 595, 598, 608, 619, 649, 650, 664, 687, 722, 727, 767, 773, 817, 824, 826, 828, 831, 836, 842, 850, 875, 895, 903, 908, 910, 924, 929, 946, 950, 953, 974, 983, 985, 992, 1035, 1077, 1269, 1284, 1302, 1308, 1334, 1345, 1348, 1382, 1384, 1386, 1393, 1394, 1399, 1409, 1420, 1448, 1474, 1502, 1518, 1559, 1572 ⟩ Used in section 4.
⟨ Go into display math mode 1148 ⟩ Used in section 1141.
⟨ Go into ordinary math mode 1142 ⟩ Used in sections 1141 and 1145.
⟨ Go through the preamble list, determining the column widths and changing the alignrecords to dummy unset boxes 804 ⟩ Used in section 803.
⟨ Grow more variable-size memory and **goto** *restart* 126 ⟩ Used in section 125.
⟨ Handle PUTEXspace command 1467 ⟩ Used in section 1461.
⟨ Handle situations involving spaces, braces, changes of state 347 ⟩ Used in section 344.
⟨ Handle the command *puxgRotateCtext* 1545 ⟩ Used in section 1544.
⟨ 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* 838 ⟩ Used in section 832.
⟨ If all characters of the family fit relative to h, then **goto** *found*, otherwise **goto** *not_found* 958 ⟩ Used in section 956.
⟨ If an alignment entry has just ended, take appropriate action 342 ⟩ Used in section 341.
⟨ If an expanded code is present, reduce it and **goto** *start_cs* 355 ⟩ Used in sections 354 and 356.
⟨ If dumping is not allowed, abort 1307 ⟩ Used in section 1305.
⟨ 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* 756 ⟩ Used in section 755.
⟨ If no hyphens were found, **return** 905 ⟩ Used in section 898.
⟨ 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*) 871 ⟩ Used in section 869.
⟨ 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 975 ⟩ Used in section 973.
⟨ 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 764 ⟩ Used in section 763.
⟨ If node r is of type *delta_node*, update *cur_active_width*, set *prev_r* and *prev_prev_r*, then **goto** *continue* 835 ⟩ Used in section 832.
⟨ 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* 1083 ⟩ Used in section 1082.
⟨ 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 1003 ⟩ Used in section 1000.
⟨ If the current wchar is at the beginning of a restricted hlist that is after a undetermined spacer, then we have to determine that space. When it is done **goto** *save_cur_wchar* 1454 ⟩ Used in section 1453.
⟨ 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* 1039 ⟩ Used in section 1037.
⟨ If the face name is missing, then ignore this face deinition 1495 ⟩ Used in section 1482.
⟨ 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* 479 ⟩ Used in section 477.
⟨ If the preamble list has been traversed, check that the row has ended 795 ⟩ Used in section 794.
⟨ If the preceding node is wchar node, then append a cespace 1451 ⟩ Used in section 1037.
⟨ If the previous node is an undetermined glue, then make it certain and **goto** *save_cur_wchar* 1456 ⟩ Used in section 1453.
⟨ If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1230 ⟩ Used in section 1229.
⟨ 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* 934 ⟩ Used in section 933.
⟨ If the string *hyph_word*[h] is less than or equal to s, interchange (*hyph_word*[h], *hyph_list*[h]) with (s, p) 944 ⟩ Used in section 943.
⟨ If the token is a wide character, then append a cspace 1452 ⟩ Used in section 1196.
⟨ If there's a ligature or kern at the cursor position, update the data structures, possibly advancing j; continue until the cursor moves 912 ⟩ Used in section 909.
⟨ If there's a ligature/kern command relevant to *cur_l* and *cur_r*, adjust the text appropriately; exit to *main_loop_wrapup* 1042 ⟩ Used in section 1037.
⟨ If this CJK font has already been loaded, set f to the internal CJK font number and **goto** common_ending 1513 ⟩ Used in section 1509.
⟨ If this Chinese face has already been loaded, then **goto** *common_ending* 1496 ⟩ Used in section 1482.
⟨ If this font has already been loaded, set f to the internal font number and **goto** *common_ending* 1263 ⟩ Used in section 1260.
⟨ If this *sup_mark* starts an expanded character like `^^A` or `^^df`, then **goto** *reswitch*, otherwise set *state* ← *mid_line* 352 ⟩ Used in section 344.
⟨ Ignore the fraction operation and complain about this ambiguous case 1186 ⟩ Used in section 1184.
⟨ Implement `\closeout` 1356 ⟩ Used in section 1351.
⟨ Implement `\immediate` 1378 ⟩ Used in section 1351.
⟨ Implement `\openout` 1354 ⟩ Used in section 1351.
⟨ Implement `\setlanguage` 1380 ⟩ Used in section 1351.
⟨ Implement `\special` 1357 ⟩ Used in section 1351.
⟨ Implement `\write` 1355 ⟩ Used in section 1351.
⟨ Incorporate a whatsit node into a vbox 1362 ⟩ Used in section 672.
⟨ Incorporate a whatsit node into an hbox 1363 ⟩ Used in section 654.
⟨ Incorporate box dimensions into the dimensions of the hbox that will contain it 656 ⟩ Used in section 654.
⟨ Incorporate box dimensions into the dimensions of the vbox that will contain it 673 ⟩ Used in section 672.
⟨ Incorporate character dimensions into the dimensions of the hbox that will contain it, then move to the next node 657 ⟩ Used in section 654.

⟨ Incorporate glue into the horizontal totals 659 ⟩ Used in section 654.
⟨ Incorporate glue into the vertical totals 674 ⟩ Used in section 672.
⟨ Increase the number of parameters in the last font 583 ⟩ Used in section 581.
⟨ Initialization of global variables done in the *main_control* procedure 1449, 1507 ⟩ Used in section 1033.
⟨ Initialize for hyphenating a paragraph 894 ⟩ Used in section 866.
⟨ Initialize table entries (done by INITEX only) 164, 222, 228, 232, 240, 250, 258, 555, 949, 954, 1219, 1304, 1372, 1411, 1412, 1417, 1480, 1501, 1504, 1519, 1532 ⟩ Used in section 8.
⟨ Initialize the current page, insert the \topskip glue ahead of p, and **goto** *continue* 1004 ⟩ Used in section 1003.
⟨ Initialize the input routines 331 ⟩ Used in section 1340.
⟨ Initialize the output routines 55, 61, 531, 536 ⟩ Used in section 1335.
⟨ Initialize the print *selector* based on *interaction* 75 ⟩ Used in sections 1268 and 1340.
⟨ Initialize the special list heads and constant nodes 793, 800, 823, 984, 991 ⟩ Used in section 164.
⟨ Initialize variables as *ship_out* begins 620 ⟩ Used in section 643.
⟨ Initialize whatever TEX might access 8 ⟩ Used in section 4.
⟨ Initiate or terminate input from a file 381 ⟩ Used in section 370.
⟨ Initiate the construction of an hbox or vbox, then **return** 1086 ⟩ Used in section 1082.
⟨ Input and store tokens from the next line of the file 486 ⟩ Used in section 485.
⟨ Input for \read from the terminal 487 ⟩ Used in section 486.
⟨ Input from external file, **goto** *restart* if no input found 343 ⟩ Used in section 341.
⟨ Input from token list, **goto** *restart* if end of list or if a parameter needs to be expanded 357 ⟩ Used in section 341.
⟨ Input the first line of *read_file*[m] 488 ⟩ Used in section 486.
⟨ Input the next line of *read_file*[m] 489 ⟩ Used in section 486.
⟨ Insert a delta node to prepare for breaks at *cur_p* 846 ⟩ Used in section 839.
⟨ Insert a delta node to prepare for the next active node 847 ⟩ Used in section 839.
⟨ Insert a dummy noad to be sub/superscripted 1180 ⟩ Used in section 1179.
⟨ Insert a new active node from *best_place*[*fit_class*] to *cur_p* 848 ⟩ Used in section 839.
⟨ Insert a new control sequence after p, then make p point to it 260 ⟩ Used in section 259.
⟨ Insert a new pattern into the linked trie 966 ⟩ Used in section 964.
⟨ Insert a new trie node between q and p, and make p point to it 967 ⟩ Used in section 966.
⟨ Insert a token containing *frozen_endv* 378 ⟩ Used in section 369.
⟨ Insert a token saved by \afterassignment, if any 1272 ⟩ Used in section 1214.
⟨ Insert glue for *split_top_skip* and set $p \leftarrow null$ 972 ⟩ Used in section 971.
⟨ Insert hyphens as specified in *hyph_list*[h] 935 ⟩ Used in section 934.
⟨ Insert macro parameter and **goto** *restart* 359 ⟩ Used in section 357.
⟨ Insert the appropriate mark text into the scanner 389 ⟩ Used in section 370.
⟨ Insert the current list into its environment 815 ⟩ Used in section 803.
⟨ Insert the pair (s,p) into the exception table 943 ⟩ Used in section 942.
⟨ Insert the $\langle v_j \rangle$ template and **goto** *restart* 792 ⟩ Used in section 342.
⟨ Insert token p into TEX's input 326 ⟩ Used in section 282.
⟨ 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 $cur\_val = fmem\_ptr$ 582 ⟩ Used in section 581.
⟨ Justify the line ending at breakpoint *cur_p*, and append it to the current vertical list, together with associated penalties and other insertions 883 ⟩ Used in section 880.
⟨ Last-minute procedures 1336, 1338, 1339, 1341 ⟩ Used in section 1333.
⟨ Lengthen the preamble periodically 796 ⟩ Used in section 795.
⟨ Let *cur_h* be the position of the first box, and set $leader\_wd + lx$ to the spacing between corresponding parts of boxes 630 ⟩ Used in section 629.
⟨ Let *cur_v* be the position of the first box, and set $leader\_ht + lx$ to the spacing between corresponding parts of boxes 639 ⟩ Used in section 638.

⟨ 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 $v \leftarrow max\_dimen$ 1150 ⟩ Used in section 1149.
⟨ 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 1151 ⟩ Used in section 1150.
⟨ Let d be the width of the whatsit p 1364 ⟩ Used in section 1150.
⟨ Let n be the largest legal code value, based on *cur_chr* 1236 ⟩ Used in section 1235.
⟨ Link node p into the current page and **goto** *done* 1001 ⟩ Used in section 1000.
⟨ Local variables for dimension calculations 453 ⟩ Used in section 451.
⟨ Local variables for finishing a displayed formula 1201 ⟩ Used in section 1197.
⟨ Local variables for formatting calculations 315 ⟩ Used in section 311.
⟨ Local variables for hyphenation 904, 915, 925, 932 ⟩ Used in section 898.
⟨ Local variables for initialization 19, 163, 930 ⟩ Used in section 4.
⟨ Local variables for line breaking 865, 896 ⟩ Used in section 818.
⟨ Look ahead for another character, or leave *lig_stack* empty if there's none there 1041 ⟩ Used in section 1037.
⟨ Look ahead for next character. If it is a wide character then append a cespace, or leave *lig_stack* empty if there's no character there 1460 ⟩ Used in section 1041.
⟨ Look at all the marks in nodes before the break, and set the final link to *null* at the break 982 ⟩ Used in section 980.
⟨ 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 711 ⟩ Used in section 710.
⟨ Look at the other stack entries until deciding what sort of `DVI` command to generate; **goto** *found* if node p is a "hit" 614 ⟩ Used in section 610.
⟨ 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 710 ⟩ Used in section 709.
⟨ Look for parameter number or `##` 482 ⟩ Used in section 480.
⟨ Look for the word $hc[1 \mathinner{\ldotp\ldotp} hn]$ in the exception table, and **goto** *found* (with *hyf* containing the hyphens) if an entry is found 933 ⟩ Used in section 926.
⟨ Look up the characters of list r in the hash table, and set *cur_cs* 377 ⟩ Used in section 375.
⟨ Lookahead and determine the type of spacer to append 1463 ⟩ Used in section 1461.
⟨ Lookahead and determine the type of *ex_spacer* to append 1464 ⟩ Used in section 1461.
⟨ Make a copy of node p in node r 205 ⟩ Used in section 204.
⟨ Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1038 ⟩ Used in section 1037.
⟨ 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 1360 ⟩ Used in section 206.
⟨ Make a second pass over the mlist, removing all noads and inserting the proper spacing and penalties 763 ⟩ Used in section 729.
⟨ Make final adjustments and **goto** *done* 579 ⟩ Used in section 565.
⟨ Make node p look like a *char_node* and **goto** *reswitch* 655 ⟩ Used in sections 625, 654, and 1150.
⟨ Make sure that *page_max_depth* is not exceeded 1006 ⟩ Used in section 1000.
⟨ Make sure that *pi* is in the proper range 834 ⟩ Used in section 832.
⟨ Make the contribution list empty by setting its tail to *contrib_head* 998 ⟩ Used in section 997.
⟨ Make the first 256 strings 48 ⟩ Used in section 47.
⟨ Make the height of box y equal to h 742 ⟩ Used in section 741.
⟨ Make the running dimensions in rule q extend to the boundaries of the alignment 809 ⟩ Used in section 808.
⟨ Make the unset node r into a *vlist_node* of height w, setting the glue as if the height were t 814 ⟩ Used in section 811.
⟨ Make the unset node r into an *hlist_node* of width w, setting the glue as if the width were t 813 ⟩ Used in section 811.
⟨ Make variable b point to a box for (f, c) 713 ⟩ Used in section 709.
⟨ Manufacture a control sequence name 375 ⟩ Used in section 370.
⟨ Math-only cases in non-math modes, or vice versa 1049 ⟩ Used in section 1048.

⟨ Merge the widths in the span nodes of q with those of p, destroying the span nodes of q 806 ⟩ Used in section 804.
⟨ Modify the cespace factor of the specified chinese face 1554 ⟩ Used in section 1550.
⟨ Modify the cspace factor of the specified chinese face 1553 ⟩ Used in section 1550.
⟨ Modify the depth factor of the specified chinese face 1555 ⟩ Used in section 1550.
⟨ 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* 884 ⟩ Used in section 883.
⟨ Modify the glue specification in *main_p* according to the space factor 1047 ⟩ Used in section 1046.
⟨ Modify the undetermined glue according the type of pre-glue character 1455 ⟩ Used in sections 1454 and 1456.
⟨ Move down or output leaders 637 ⟩ Used in section 634.
⟨ 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 1000 ⟩ Used in section 997.
⟨ Move pointer s to the end of the current list, and set *replace_count*(r) appropriately 921 ⟩ Used in section 917.
⟨ Move right or output leaders 628 ⟩ Used in section 625.
⟨ Move the characters of a ligature node to *hu* and *hc*; but **goto** *done3* if they are not all letters 901 ⟩ Used in section 900.
⟨ Move the cursor past a pseudo-ligature, then **goto** *main_loop_lookahead* or *main_lig_loop* 1040 ⟩ Used in section 1037.
⟨ Move the data into *trie* 961 ⟩ Used in section 969.
⟨ Move to next line of file, or **goto** *restart* if there is no next line, or **return** if a `\read` line has finished 360 ⟩ Used in section 343.
⟨ Negate all three glue components of *cur_val* 434 ⟩ Used in section 433.
⟨ Nullify *width*(q) and the tabskip glue following this column 805 ⟩ Used in section 804.
⟨ Numbered cases for *debug_help* 1342 ⟩ Used in section 1341.
⟨ Open *tfm_file* for input 566 ⟩ Used in section 565.
⟨ Other local variables for *try_break* 833 ⟩ Used in section 832.
⟨ Other local variables used by procedure *new_font* 1508 ⟩ Used in section 1260.
⟨ Other variables used by the procedure *prefixed_command* 1415, 1536, 1540, 1551, 1564 ⟩ Used in section 1214.
⟨ Other variables used by *new_cface* 1486 ⟩ Used in section 1482.
⟨ Output a box in a vlist 635 ⟩ Used in section 634.
⟨ Output a box in an hlist 626 ⟩ Used in section 625.
⟨ Output a leader box at *cur_h*, then advance *cur_h* by $leader\_wd + lx$ 631 ⟩ Used in section 629.
⟨ Output a leader box at *cur_v*, then advance *cur_v* by $leader\_ht + lx$ 640 ⟩ Used in section 638.
⟨ Output a rule in a vlist, **goto** *next_p* 636 ⟩ Used in section 634.
⟨ Output a rule in an hlist 627 ⟩ Used in section 625.
⟨ Output a substitution, **goto** *continue* if not possible 1398 ⟩ Used in section 623.
⟨ Output leaders in a vlist, **goto** *fin_rule* if a rule or to *next_p* if done 638 ⟩ Used in section 637.
⟨ Output leaders in an hlist, **goto** *fin_rule* if a rule or to *next_p* if done 629 ⟩ Used in section 628.
⟨ Output node p for *hlist_out* and move to the next node, maintaining the condition $cur\_v = base\_line$ 623 ⟩ Used in section 622.
⟨ Output node p for *vlist_out* and move to the next node, maintaining the condition $cur\_h = left\_edge$ 633 ⟩ Used in section 632.
⟨ Output statistics about this job 1337 ⟩ Used in section 1336.
⟨ Output the CJK font definitions for all fonts that were used 1573 ⟩ Used in section 645.
⟨ Output the font definitions for all fonts that were used 646 ⟩ Used in section 645.
⟨ Output the font name whose internal number is f 606 ⟩ Used in section 605.
⟨ Output the non-*char_node* p for *hlist_out* and move to the next node 625 ⟩ Used in section 623.
⟨ Output the non-*char_node* p for *vlist_out* 634 ⟩ Used in section 633.
⟨ Output the whatsit node p in a vlist 1369 ⟩ Used in section 634.
⟨ Output the whatsit node p in an hlist 1370 ⟩ Used in section 625.
⟨ PUTeX basic scanning routines 1418 ⟩ Used in section 466.

⟨ PUTeX routines that will be used by TeX routines 1413, 1431, 1478, 1479 ⟩ Used in section 4.
⟨ Pack the family into *trie* relative to h 959 ⟩ Used in section 956.
⟨ Package an unset box for the current column and record its width 799 ⟩ Used in section 794.
⟨ Package the preamble list, to determine the actual tabskip glue amounts, and let p point to this prototype box 807 ⟩ Used in section 803.
⟨ Perform the default output routine 1026 ⟩ Used in section 1015.
⟨ Pontificate about improper alignment in display 1210 ⟩ Used in section 1209.
⟨ Pop the condition stack 499 ⟩ Used in sections 501, 503, 512, and 513.
⟨ Prepare a nonbreak space if the current wide character is not allowed to appear at the end of line 1458 ⟩ Used in section 1453.
⟨ Prepare all the boxes involved in insertions to act as queues 1021 ⟩ Used in section 1017.
⟨ Prepare to deactivate node r, and **goto** *deactivate* unless there is a reason to consider lines of text from r to *cur_p* 857 ⟩ Used in section 854.
⟨ Prepare to insert a token that matches *cur_group*, and print what it is 1068 ⟩ Used in section 1067.
⟨ Prepare to move a box or rule node to the current page, then **goto** *contribute* 1005 ⟩ Used in section 1003.
⟨ Prepare to move whatsit p to the current page, then **goto** *contribute* 1367 ⟩ Used in section 1003.
⟨ Print CJK font faces 1569 ⟩ Used in section 1567.
⟨ Print CJK fonts 1570 ⟩ Used in section 1567.
⟨ Print TeX fonts 1568 ⟩ Used in section 1567.
⟨ Print a short indication of the contents of node p 175 ⟩ Used in section 174.
⟨ Print a symbolic description of the new break node 849 ⟩ Used in section 848.
⟨ Print a symbolic description of this feasible break 859 ⟩ Used in section 858.
⟨ Print character substition tracing log 1401 ⟩ Used in section 1398.
⟨ Print either '`definition`' or '`use`' or '`preamble`' or '`text`', and insert tokens that should lead to recovery 339 ⟩ Used in section 338.
⟨ Print font faces matching table 1571 ⟩ Used in section 1567.
⟨ Print location of current line 313 ⟩ Used in section 312.
⟨ Print newly busy locations 171 ⟩ Used in section 167.
⟨ Print string s as an error message 1286 ⟩ Used in section 1282.
⟨ Print string s on the terminal 1283 ⟩ Used in section 1282.
⟨ Print the banner line, including the date and time 539 ⟩ Used in section 537.
⟨ Print the font identifier for *font*(p) 267 ⟩ Used in sections 174 and 176.
⟨ 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* $\gets$ *cur_p* 860 ⟩ Used in section 859.
⟨ Print the menu of available options 85 ⟩ Used in section 84.
⟨ Print the result of command c 475 ⟩ Used in section 473.
⟨ Print two lines using the tricky pseudoprinted information 317 ⟩ Used in section 312.
⟨ Print type of token list 314 ⟩ Used in section 312.
⟨ Process an active-character control sequence and set *state* $\gets$ *mid_line* 353 ⟩ Used in section 344.
⟨ 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 730 ⟩ Used in section 729.
⟨ Process whatsit p in *vert_break* loop, **goto** *not_found* 1368 ⟩ Used in section 976.
⟨ 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 1124 ⟩ Used in section 1122.
⟨ Prune unwanted nodes at the beginning of the next line 882 ⟩ Used in section 880.
⟨ Pseudoprint the line 318 ⟩ Used in section 312.
⟨ Pseudoprint the token list 319 ⟩ Used in section 312.
⟨ Push the condition stack 498 ⟩ Used in section 501.
⟨ Put each of TeX's primitives into the hash table 226, 230, 238, 248, 265, 334, 379, 387, 414, 419, 471, 490, 494, 556, 783, 986, 1055, 1061, 1074, 1091, 1110, 1117, 1144, 1159, 1172, 1181, 1191, 1211, 1222, 1225, 1233, 1253, 1257, 1265,

1275, 1280, 1289, 1294, 1347, 1414, 1425, 1429, 1442, 1465, 1469, 1475, 1520, 1533, 1542, 1548, 1556, 1561, 1565 ⟩ Used in section 1339.
⟨ 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 919 ⟩ Used in section 917.
⟨ Put the characters $hu[l\ ..\ i]$ and a hyphen into *pre_break*(r) 918 ⟩ Used in section 917.
⟨ Put the fraction into a box with its delimiters, and make *new_hlist*(q) point to it 751 ⟩ Used in section 746.
⟨ Put the `\leftskip` glue at the left and detach this line 890 ⟩ Used in section 883.
⟨ 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 1017 ⟩ Used in section 1015.
⟨ Put the (positive) 'at' size into s 1262 ⟩ Used in section 1261.
⟨ Put the `\rightskip` glue after node q 889 ⟩ Used in section 884.
⟨ 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* 565 ⟩ Used in section 563.
⟨ Read box dimensions 574 ⟩ Used in section 565.
⟨ Read character data 572 ⟩ Used in section 565.
⟨ Read extensible character recipes 577 ⟩ Used in section 565.
⟨ Read font parameters 578 ⟩ Used in section 565.
⟨ Read ligature/kern program 576 ⟩ Used in section 565.
⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 362 ⟩ Used in section 360.
⟨ 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 541 ⟩ Used in section 540.
⟨ 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 571 ⟩ Used in section 565.
⟨ Read the `TFM` size fields 568 ⟩ Used in section 565.
⟨ Readjust the height and depth of *cur_box*, for `\vtop` 1090 ⟩ Used in section 1089.
⟨ Rebuild character using substitution information 1402 ⟩ Used in section 1398.
⟨ Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 916 ⟩ Used in section 906.
⟨ Record a new feasible break 858 ⟩ Used in section 854.
⟨ Recover from an unbalanced output routine 1030 ⟩ Used in section 1029.
⟨ Recover from an unbalanced write command 1375 ⟩ Used in section 1374.
⟨ Recycle node p 1002 ⟩ Used in section 1000.
⟨ Remove the last box, unless it's part of a discretionary 1084 ⟩ Used in section 1083.
⟨ Replace nodes $ha\ ..\ hb$ by a sequence of nodes that includes the discretionary hyphens 906 ⟩ Used in section 898.
⟨ Replace the tail of the list by p 1190 ⟩ Used in section 1189.
⟨ Replace z by z' and compute α, β 575 ⟩ Used in sections 574 and 1441.
⟨ Report a runaway argument and abort 399 ⟩ Used in sections 395 and 402.
⟨ Report a tight hbox and **goto** *common_ending*, if this box is sufficiently bad 670 ⟩ Used in section 667.
⟨ Report a tight vbox and **goto** *common_ending*, if this box is sufficiently bad 681 ⟩ Used in section 679.
⟨ Report an extra right brace and **goto** *continue* 398 ⟩ Used in section 395.
⟨ Report an improper use of the macro and abort 401 ⟩ Used in section 400.
⟨ Report an overfull hbox and **goto** *common_ending*, if this box is sufficiently bad 669 ⟩ Used in section 667.
⟨ Report an overfull vbox and **goto** *common_ending*, if this box is sufficiently bad 680 ⟩ Used in section 679.
⟨ Report an underfull hbox and **goto** *common_ending*, if this box is sufficiently bad 663 ⟩ Used in section 661.
⟨ Report an underfull vbox and **goto** *common_ending*, if this box is sufficiently bad 677 ⟩ Used in section 676.
⟨ Report overflow of the input buffer, and abort 35 ⟩ Used in section 31.
⟨ Report that an invalid delimiter code is being changed to null; set *cur_val* $\leftarrow 0$ 1164 ⟩ Used in section 1163.
⟨ Report that the font won't be loaded 564 ⟩ Used in section 563.
⟨ Report that this dimension is out of range 463 ⟩ Used in section 451.

⟨ Resume the page builder after an output routine has come to an end 1029 ⟩ Used in section 1103.
⟨ Reverse the links of the relevant passive nodes, setting *cur_p* to the first breakpoint 881 ⟩ Used in section 880.
⟨ Scan CJK font face identifier 1484 ⟩ Used in section 1482.
⟨ Scan CJK font face name 1485 ⟩ Used in section 1482.
⟨ Scan a control sequence and set *state* ← *skip_blanks* or *mid_line* 354 ⟩ Used in section 344.
⟨ Scan a numeric constant 447 ⟩ Used in section 443.
⟨ Scan a parameter until its delimiter string has been found; or, if $s = null$, simply scan the delimiter string 395 ⟩ Used in section 394.
⟨ Scan a subformula enclosed in braces and **return** 1156 ⟩ Used in section 1154.
⟨ 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* 356 ⟩ Used in section 354.
⟨ Scan an alphabetic character code into *cur_val* 445 ⟩ Used in section 443.
⟨ Scan an optional space 446 ⟩ Used in sections 445, 451, 458, and 1203.
⟨ Scan and build the body of the token list; **goto** *found* when finished 480 ⟩ Used in section 476.
⟨ Scan and build the parameter part of the macro definition 477 ⟩ Used in section 476.
⟨ Scan decimal fraction 455 ⟩ Used in section 451.
⟨ Scan file name in the buffer 534 ⟩ Used in section 533.
⟨ Scan for all other units and adjust *cur_val* and f accordingly; **goto** *done* in the case of scaled points 461 ⟩ Used in section 456.
⟨ Scan for `fil` units; **goto** *attach_fraction* if found 457 ⟩ Used in section 456.
⟨ Scan for `mu` units and **goto** *attach_fraction* 459 ⟩ Used in section 456.
⟨ Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 458 ⟩ Used in section 456.
⟨ Scan optional CJK font face definition parameters 1487 ⟩ Used in section 1482.
⟨ 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 782 ⟩ Used in section 780.
⟨ Scan spacing dimension of CJK font face 1552 ⟩ Used in sections 1550 and 1563.
⟨ Scan the CJK font charset 1488 ⟩ Used in section 1487.
⟨ Scan the CJK font depth 1491 ⟩ Used in section 1487.
⟨ Scan the CJK font height 1490 ⟩ Used in section 1487.
⟨ Scan the CJK font style 1493 ⟩ Used in section 1487.
⟨ Scan the CJK font weight 1492 ⟩ Used in section 1487.
⟨ Scan the CJK font width 1489 ⟩ Used in section 1487.
⟨ Scan the argument for command c 474 ⟩ Used in section 473.
⟨ Scan the font size specification 1261 ⟩ Used in section 1260.
⟨ Scan the parameters and make $link(r)$ point to the macro body; but **return** if an illegal `\par` is detected 394 ⟩ Used in section 392.
⟨ Scan the preamble and record it in the *preamble* list 780 ⟩ Used in section 777.
⟨ Scan the template ⟨u_j⟩, putting the resulting token list in *hold_head* 786 ⟩ Used in section 782.
⟨ Scan the template ⟨v_j⟩, putting the resulting token list in *hold_head* 787 ⟩ Used in section 782.
⟨ 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 456 ⟩ Used in section 451.
⟨ Search *eqtb* for equivalents equal to p 255 ⟩ Used in section 172.
⟨ Search *hyph_list* for pointers to p 936 ⟩ Used in section 172.
⟨ Search *save_stack* for equivalents that point to p 285 ⟩ Used in section 172.
⟨ Select the appropriate case and **return** or **goto** *common_ending* 512 ⟩ Used in section 504.
⟨ Set CJK font rotation style 1494 ⟩ Used in section 1493.
⟨ Set PUTeX global parameter *puxgCfaceDepth* 1547 ⟩ Used in section 1546.
⟨ Set initial values of key variables 21, 23, 24, 74, 77, 80, 97, 166, 215, 254, 257, 272, 287, 368, 386, 442, 484, 493, 554, 559, 596, 599, 609, 651, 665, 688, 774, 931, 993, 1036, 1270, 1285, 1303, 1346, 1383, 1395, 1410, 1503, 1560 ⟩

Used in section 8.
⟨ Set line length parameters in preparation for hanging indentation 852 ⟩ Used in section 851.
⟨ Set the glue in all the unset boxes of the current list 808 ⟩ Used in section 803.
⟨ Set the glue in node r and change it from an unset node 811 ⟩ Used in section 810.
⟨ Set the matching CJK font 1537 ⟩ Used in section 1220.
⟨ Set the unset box q and the unset boxes in it 810 ⟩ Used in section 808.
⟨ Set the value of b to the badness for shrinking the line, and compute the corresponding *fit_class* 856 ⟩ Used in section 854.
⟨ Set the value of b to the badness for stretching the line, and compute the corresponding *fit_class* 855 ⟩ Used in section 854.
⟨ Set the value of *output_penalty* 1016 ⟩ Used in section 1015.
⟨ Set up data structures with the cursor following position j 911 ⟩ Used in section 909.
⟨ Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 706 ⟩ Used in sections 723, 729, 733, 757, 763, and 766.
⟨ Set variable c to the current escape character 243 ⟩ Used in section 63.
⟨ Setup this new Chinese face 1497 ⟩ Used in section 1482.
⟨ Setup variables before scanning CJK font face parameters 1483 ⟩ Used in section 1482.
⟨ Setup *hbox_tail* and package 1462 ⟩ Used in sections 1088 and 1088.
⟨ Ship box p out 643 ⟩ Used in section 641.
⟨ Show equivalent n, in region 1 or 2 223 ⟩ Used in section 252.
⟨ Show equivalent n, in region 3 229 ⟩ Used in section 252.
⟨ Show equivalent n, in region 4 233 ⟩ Used in section 252.
⟨ Show equivalent n, in region 5 242 ⟩ Used in section 252.
⟨ Show equivalent n, in region 6 251 ⟩ Used in section 252.
⟨ Show the auxiliary field, a 219 ⟩ Used in section 218.
⟨ Show the current contents of a box 1299 ⟩ Used in section 1296.
⟨ Show the current meaning of a token, then **goto** *common_ending* 1297 ⟩ Used in section 1296.
⟨ Show the current value of some parameter or register, then **goto** *common_ending* 1300 ⟩ Used in section 1296.
⟨ Show the font identifier in *eqtb*[n] 234 ⟩ Used in section 233.
⟨ Show the halfword code in *eqtb*[n] 235 ⟩ Used in section 233.
⟨ Show the status of the current page 989 ⟩ Used in section 218.
⟨ Show the text of the macro being expanded 404 ⟩ Used in section 392.
⟨ Simplify a trivial box 724 ⟩ Used in section 723.
⟨ Skip to `\else` or `\fi`, then **goto** *common_ending* 503 ⟩ Used in section 501.
⟨ Skip to node *ha*, or **goto** *done1* if no hyphenation should be attempted 899 ⟩ Used in section 897.
⟨ Skip to node *hb*, putting letters into *hu* and *hc* 900 ⟩ Used in section 897.
⟨ Sort p into the list starting at *rover* and advance p to *rlink*(p) 132 ⟩ Used in section 131.
⟨ Sort the hyphenation op tables into proper order 948 ⟩ Used in section 955.
⟨ Split off part of a vertical box, make *cur_box* point to it 1085 ⟩ Used in section 1082.
⟨ Squeeze the equation as much as possible; if there is an equation number that should go on a separate line by itself, set $e \leftarrow 0$ 1204 ⟩ Used in section 1202.
⟨ Start a new current page 994 ⟩ Used in section 1020.
⟨ Store *cur_box* in a box register 1080 ⟩ Used in section 1078.
⟨ Store maximum values in the *hyf* table 927 ⟩ Used in section 926.
⟨ Store *save_stack*[*save_ptr*] in *eqtb*[p], unless *eqtb*[p] holds a global value 283 ⟩ Used in section 282.
⟨ Store the current token, but **goto** *continue* if it is a blank space that would become an undelimited parameter 396 ⟩ Used in section 395.
⟨ Subtract glue from *break_width* 841 ⟩ Used in section 840.
⟨ Subtract the width of node v from *break_width* 844 ⟩ Used in section 843.
⟨ Suppress expansion of the next token 372 ⟩ Used in section 370.
⟨ Swap the subscript and superscript into box x 745 ⟩ Used in section 741.

⟨ Switch to a larger accent if available and appropriate 743 ⟩ Used in section 741.
⟨ Tell the user what has run away and try to recover 338 ⟩ Used in section 336.
⟨ Terminate the current conditional and skip to `\fi` 513 ⟩ Used in section 370.
⟨ Test box register status 508 ⟩ Used in section 504.
⟨ Test if an integer is odd 507 ⟩ Used in section 504.
⟨ Test if two characters match 509 ⟩ Used in section 504.
⟨ Test if two macro texts match 511 ⟩ Used in section 510.
⟨ Test if two tokens match 510 ⟩ Used in section 504.
⟨ Test relation between integers or dimensions 506 ⟩ Used in section 504.
⟨ The em width for *cur_font* 561 ⟩ Used in section 458.
⟨ The x-height for *cur_font* 562 ⟩ Used in section 458.
⟨ Tidy up the parameter just scanned, and tuck it away 403 ⟩ Used in section 395.
⟨ Transfer node p to the adjustment list 658 ⟩ Used in section 654.
⟨ Transplant the post-break list 887 ⟩ Used in section 885.
⟨ Transplant the pre-break list 888 ⟩ Used in section 885.
⟨ Treat *cur_chr* as an active character 1155 ⟩ Used in sections 1154 and 1158.
⟨ Try the final line break at the end of the paragraph, and **goto** *done* if the desired breakpoints have been found 876 ⟩ Used in section 866.
⟨ Try to allocate within node p and its physical successors, and **goto** *found* if allocation was possible 127 ⟩ Used in section 125.
⟨ Try to break after a discretionary fragment, then **goto** *done5* 872 ⟩ Used in section 869.
⟨ Try to get a different log file name 538 ⟩ Used in section 537.
⟨ Try to hyphenate the following word 897 ⟩ Used in section 869.
⟨ Try to recover from mismatched `\right` 1195 ⟩ Used in section 1194.
⟨ Types in the outer block 18, 25, 38, 101, 109, 113, 150, 212, 269, 300, 551, 597, 923, 928, 1439, 1473, 1500, 1517 ⟩ Used in section 4.
⟨ Undump MLTEX-specific data 1404 ⟩ Used in section 1306.
⟨ Undump a couple more things and the closing check word 1330 ⟩ Used in section 1306.
⟨ Undump constants for consistency check 1311 ⟩ Used in section 1306.
⟨ Undump regions 1 to 6 of *eqtb* 1320 ⟩ Used in section 1317.
⟨ Undump the CJK font face information 1577 ⟩ Used in section 1306.
⟨ Undump the CJK font information 1581 ⟩ Used in section 1306.
⟨ Undump the array info for internal font number k 1326 ⟩ Used in section 1324.
⟨ Undump the dynamic memory 1315 ⟩ Used in section 1306.
⟨ Undump the font information 1324 ⟩ Used in section 1306.
⟨ Undump the hash table 1322 ⟩ Used in section 1317.
⟨ Undump the hyphenation tables 1328 ⟩ Used in section 1306.
⟨ Undump the string pool 1313 ⟩ Used in section 1306.
⟨ Undump the table of equivalents 1317 ⟩ Used in section 1306.
⟨ Undump *xord*, *xchr*, and *xprn* 1390 ⟩ Used in section 1311.
⟨ Unump the face matching table 1579 ⟩ Used in section 1306.
⟨ Update the active widths, since the first active node has been deleted 864 ⟩ Used in section 863.
⟨ Update the current height and depth measurements with respect to a glue or kern node p 979 ⟩ Used in section 975.
⟨ Update the current page measurements with respect to the glue or kern specified by node p 1007 ⟩ Used in section 1000.
⟨ Update the value of *printed_node* for symbolic displays 861 ⟩ Used in section 832.
⟨ Update the values of *first_mark* and *bot_mark* 1019 ⟩ Used in section 1017.
⟨ Update the values of *last_glue*, *last_penalty*, and *last_kern* 999 ⟩ Used in section 997.
⟨ Update the values of *max_h* and *max_v*; but if the page is too large, **goto** *done* 644 ⟩ Used in section 643.
⟨ Update width entry for spanned columns 801 ⟩ Used in section 799.
⟨ Use code c to distinguish between generalized fractions 1185 ⟩ Used in section 1184.

⟨ 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 976 ⟩ Used in section 975.
⟨ Use size fields to allocate font information 569 ⟩ Used in section 565.
⟨ Wipe out the whatsit node p and **goto** *done* 1361 ⟩ Used in section 202.
⟨ 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 1024 ⟩ Used in section 1023.
⟨ print a CJK name sequence member 1438 ⟩ Used in section 475.
⟨ print a CJK number with specified format 1436 ⟩ Used in section 475.
⟨ scan PUTEX internal values 1428 ⟩ Used in section 416.
⟨ scan a CJK name sequence number 1437 ⟩ Used in section 474.
⟨ scan a CJK number with a possible selector and then split it 1433 ⟩ Used in section 474.
⟨ scan and split the number 1432 ⟩ Used in section 474.
⟨ the previous node is a character node, so we have to append a glue first 1457 ⟩ Used in section 1453.
⟨ using full-width arabic characters to print a CJK number 1435 ⟩ Used in section 475.

		Section	Page
Changes to 1.	Introduction	1	3
Changes to 2.	The character set	17	11
Changes to 3.	Input and output	25	14
Changes to 4.	String handling	38	20
Changes to 5.	On-line and off-line printing	54	25
Changes to 6.	Reporting errors	72	32
Changes to 7.	Arithmetic with scaled dimensions	99	41
Changes to 8.	Packed data	110	45
Changes to 9.	Dynamic memory allocation	115	47
Changes to 10.	Data structures for boxes and their friends	133	53
Changes to 11.	Memory layout	162	61
Changes to 12.	Displaying boxes	173	65
Changes to 13.	Destroying boxes	199	73
Changes to 14.	Copying boxes	203	75
Changes to 15.	The command codes	207	77
Changes to 16.	The semantic nest	211	82
Changes to 17.	The table of equivalents	220	86
Changes to 18.	The hash table	256	109
Changes to 19.	Saving and restoring equivalents	268	116
Changes to 20.	Token lists	289	122
Changes to 21.	Introduction to the syntactic routines	297	126
Changes to 22.	Input stacks and states	300	128
Changes to 23.	Maintaining the input stacks	321	138
Changes to 24.	Getting the next token	332	141
Changes to 25.	Expanding the next token	366	152
Changes to 26.	Basic scanning subroutines	405	163
Changes to 27.	Building token lists	467	182
Changes to 28.	Conditional processing	490	191
Changes to 29.	File names	514	198
Changes to 30.	Font metric data	542	211
Changes to 31.	Device-independent file format	586	228
Changes to 32.	Shipping pages out	595	236
Changes to 33.	Packaging	647	257
Changes to 34.	Data structures for math mode	683	267
Changes to 35.	Subroutines for math mode	702	276
Changes to 36.	Typesetting math formulas	722	283
Changes to 37.	Alignment	771	303
Changes to 38.	Breaking paragraphs into lines	816	320
Changes to 39.	Breaking paragraphs into lines, continued	865	337
Changes to 40.	Pre-hyphenation	894	349
Changes to 41.	Post-hyphenation	903	353
Changes to 42.	Hyphenation	922	363
Changes to 43.	Initializing the hyphenation tables	945	370
Changes to 44.	Breaking vertical lists into pages	970	381
Changes to 45.	The page builder	983	387
Changes to 46.	The chief executive	1032	404
Changes to 47.	Building boxes and lists	1058	415
Changes to 48.	Building math lists	1139	437
Changes to 49.	Mode-independent processing	1211	456
Changes to 50.	Dumping and undumping the tables	1302	478
Changes to 51.	The main program	1333	491
Changes to 52.	Debugging	1341	498

Changes to 53. Extensions	1343	500
Changes to 54/web2c. System-dependent changes for Web2c	1382	510
Changes to 54/web2c-string. The string recycling routines	1391	512
Changes to 54/MLTEX. System-dependent changes for MLTEX	1393	513
Changes to 54. System-dependent changes	1405	519
Changes to 55. Introduction to PUTEX	1408	520
Changes to 56. CJK Numbers	1420	522
Changes to 58. All about spaces	1447	528
Changes to 59. CJK font face definition table	1468	537
Changes to 59. CJK font definition table	1498	546
Changes to 57. Matching faces	1516	550
Changes to 60. Font matching	1531	554
Changes to 61. Dump Font Info	1564	561
Changes to 62. Dump/undump PUTEX internal information	1575	563
Changes to 63. Index	1582	565