March 14, 2024 at 18:52

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].

define *banner* ≡ `´This␣is␣TeX,␣Version␣3.141592653´` { printed when TEX starts }

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 and 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.

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

```
  define mtype ≡ t@&y@&p@&e   { this is a WEB coding trick: }
  format mtype ≡ type   { 'mtype' will be equivalent to 'type' }
  format type ≡ true   { but 'type' will not be treated as a reserved word }
⟨Compiler directives 9⟩
program TEX;   { all file names are defined dynamically }
  label ⟨Labels in the outer block 6⟩
  const ⟨Constants in the outer block 11⟩
  mtype ⟨Types in the outer block 18⟩
  var ⟨Global variables 13⟩
  procedure initialize;   { this procedure gets things started properly }
    var ⟨Local variables for initialization 19⟩
    begin ⟨Initialize whatever TEX might access 8⟩
    end;
  ⟨Basic printing procedures 57⟩
  ⟨Error handling procedures 78⟩
```

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

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

⟨ Labels in the outer block 6 ⟩ ≡
start_of_TEX, *end_of_TEX*, *final_end*; { key control points }

This code is used in section 4.

7. Some of the code below is intended to be used only when diagnosing the strange behavior that sometimes occurs when 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`, `\tracingpages`, and `\tracingrestores`.

define *debug* ≡ @{ { change this to '*debug* ≡ ' when debugging }
define *gubed* ≡ @} { change this to '*gubed* ≡ ' when debugging }
format *debug* ≡ *begin*
format *gubed* ≡ *end*

define *stat* ≡ @{ { change this to '*stat* ≡ ' when gathering usage statistics }
define *tats* ≡ @} { change this to '*tats* ≡ ' when gathering usage statistics }
format *stat* ≡ *begin*
format *tats* ≡ *end*

8. This program has two important variations: (1) There is a long and slow version called INITEX, which does the extra calculations needed to initialize 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**'.

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

⟨ Initialize whatever 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.

⟨ Constants in the outer block 11 ⟩ ≡

mem_max = 30000;
  { greatest index in TeX's internal *mem* array; must be strictly less than *max_halfword*; must be equal to *mem_top* in `INITEX`, otherwise ≥ *mem_top* }
mem_min = 0; { smallest index in TeX's internal *mem* array; must be *min_halfword* or more; must be equal to *mem_bot* in `INITEX`, otherwise ≤ *mem_bot* }
buf_size = 500; { maximum number of characters simultaneously present in current lines of open files and in control sequences between `\csname` and `\endcsname`; must not exceed *max_halfword* }
error_line = 72; { width of context lines on terminal error messages }
half_error_line = 42; { width of first lines of contexts in terminal error messages; should be between 30 and *error_line* − 15 }
max_print_line = 79; { width of longest text lines output; should be at least 60 }
stack_size = 200; { maximum number of simultaneous input sources }
max_in_open = 6;
  { maximum number of input files and error insertions that can be going on simultaneously }
font_max = 75; { maximum internal font number; must not exceed *max_quarterword* and must be at most *font_base* + 256 }
font_mem_size = 20000; { number of words of *font_info* for all fonts }
param_size = 60; { maximum number of simultaneous macro parameters }
nest_size = 40; { maximum number of semantic levels simultaneously active }
max_strings = 3000; { maximum number of strings; must not exceed *max_halfword* }
string_vacancies = 8000; { the minimum number of characters that should be available for the user's control sequences and font names, after TeX's own error messages are stored }
pool_size = 32000; { maximum number of characters in strings, including all error messages and help texts, and the names of all fonts and control sequences; must exceed *string_vacancies* by the total length of TeX's own strings, which is currently about 23000 }
save_size = 600; { space for saving values outside of current group; must be at most *max_halfword* }
trie_size = 8000; { space for hyphenation patterns; should be larger for `INITEX` than it is in production versions of TeX }
trie_op_size = 500; { space for "opcodes" in the hyphenation patterns }
dvi_buf_size = 800; { size of the output buffer; must be a multiple of 8 }
file_name_size = 40; { file names shouldn't be longer than this }
pool_name = ´`TeXformats:TEX.POOL␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣`´;
  { string of length *file_name_size*; tells where the string pool appears }

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

13. In case somebody has inadvertently made bad settings of the "constants," 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, 382, 387, 388, 410, 438, 447, 480, 489, 493, 512, 513, 520, 527, 532, 539, 549, 550, 555, 592, 595, 605, 616, 646, 647, 661, 684, 719, 724, 764, 770, 814, 821, 823, 825, 828, 833, 839, 847, 872, 892, 900, 905, 907, 921, 926, 943, 947, 950, 971, 980, 982, 989, 1032, 1074, 1266, 1281, 1299, 1305, 1331, 1342, and 1345.

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, 522, and 1249.

This code is used in section 1332.

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

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

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

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

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

⟨ 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, 548, 594, 920, and 925.

This code is used in section 4.

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

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

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

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

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

See also sections 163 and 927.

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 }

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

⟨ Set initial values of key variables 21 ⟩ ≡

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

See also sections 23, 24, 74, 77, 80, 97, 166, 215, 254, 257, 272, 287, 383, 439, 481, 490, 521, 551, 556, 593, 596, 606, 648, 662, 685, 771, 928, 990, 1033, 1267, 1282, 1300, and 1343.

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 ⟩ +≡
  for $i \leftarrow 0$ **to** ´37 **do** $xchr[i] \leftarrow$ ´␣´;
  for $i \leftarrow$ ´177 **to** ´377 **do** $xchr[i] \leftarrow$ ´␣´;

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** $xord[chr(i)] \leftarrow$ *invalid_code*;
  for $i \leftarrow$ ´200 **to** ´377 **do** $xord[xchr[i]] \leftarrow i$;
  for $i \leftarrow 0$ **to** ´176 **do** $xord[xchr[i]] \leftarrow i$;

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

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

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

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

reset(f, *name*, ´/O´)	for input;
rewrite(f, *name*, ´/O´)	for output.

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

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

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

define $reset\_OK(\#) \equiv erstat(\#) = 0$
define $rewrite\_OK(\#) \equiv erstat(\#) = 0$

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

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

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

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

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

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

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

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

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

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

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

29. Binary input and output are done with Pascal's ordinary *get* and *put* procedures, so we don't have to make any other special arrangements for binary I/O. Text output is also easy to do with standard Pascal routines. The treatment of text input is more difficult, however, because of the necessary translation to *ASCII_code* values. 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: **array** [0 .. *buf_size*] **of** *ASCII_code*; { lines of characters being read }
first: 0 .. *buf_size*; { the first unused position in *buffer* }
last: 0 .. *buf_size*; { end of the line just input to *buffer* }
max_buf_stack: 0 .. *buf_size*; { largest index used in *buffer* }

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

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

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

If the *bypass_eoln* parameter is *true*, *input_ln* will do a *get* before looking at the first character of the line; this skips over an *eoln* that was in $f\uparrow$. 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.

```
function input_ln(var f : alpha_file; bypass_eoln : boolean): boolean;
        { inputs the next line or returns false }
  var last_nonblank: 0 .. buf_size;  { last with trailing blanks removed }
  begin if bypass_eoln then
    if ¬eof(f) then get(f);  { input the first character of the line into f↑ }
  last ← first;  { cf. Matthew 19:30 }
  if eof(f) then input_ln ← false
  else begin last_nonblank ← first;
    while ¬eoln(f) do
      begin if last ≥ max_buf_stack then
        begin max_buf_stack ← last + 1;
        if max_buf_stack = buf_size then ⟨Report overflow of the input buffer, and abort 35⟩;
        end;
      buffer[last] ← xord[f↑]; get(f); incr(last);
      if buffer[last − 1] ≠ "␣" then last_nonblank ← last;
      end;
    last ← last_nonblank; input_ln ← true;
    end;
  end;
```

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

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

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

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

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

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

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

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

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

⟨ Report overflow of the input buffer, and abort 35 ⟩ ≡
if *format_ident* = 0 **then**
  begin *write_ln*(*term_out*, ´Buffer␣size␣exceeded!´); **goto** *final_end*;
  end
else begin *cur_input.loc_field* ← *first*; *cur_input.limit_field* ← *last* − 1;
  overflow("buffer␣size", *buf_size*);
  end

This code is used in 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 without retrieving a possible command line. It should be clear how to modify this routine to deal with command lines, if the system permits them.

```
function init_terminal: boolean;   { gets the terminal input started }
  label exit;
  begin t_open_in;
  loop begin wake_up_terminal; write(term_out, ´**´); update_terminal;
    if ¬input_ln(term_in, true) then   { this shouldn't happen }
      begin write_ln(term_out); write(term_out, ´!␣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 $-128 \mathrel{..} 127$. To accommodate such systems we access the string pool only via macros that can easily be redefined.

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

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

39. ⟨ Global variables 13 ⟩ +≡
str_pool: **packed array** [*pool_pointer*] **of** *packed_ASCII_code*; { the characters }
str_start: **array** [*str_number*] **of** *pool_pointer*; { the starting pointers }
pool_ptr: *pool_pointer*; { first unused position in *str_pool* }
str_ptr: *str_number*; { number of the current string being created }
init_pool_ptr: *pool_pointer*; { the starting value of *pool_ptr* }
init_str_ptr: *str_number*; { the starting value of *str_ptr* }

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

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

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

define *cur_length* ≡ (*pool_ptr* − *str_start*[*str_ptr*])

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

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

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

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

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

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

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

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

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

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

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

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

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

48. **define** *app_lc_hex*(#) ≡ *l* ← #;
  if $l < 10$ **then** *append_char*(*l* + "0") **else** *append_char*(*l* − 10 + "a")

```
⟨Make the first 256 strings 48⟩ ≡
  for k ← 0 to 255 do
    begin if (⟨Character k cannot be printed 49⟩) then
      begin append_char("^"); append_char("^");
      if k < ´100 then append_char(k + ´100)
      else if k < ´200 then append_char(k − ´100)
        else begin app_lc_hex(k 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] = ´`≠`´, would like string ´32 to be the single character ´32 instead of the three characters ´136, ´136, ´132 (`^^Z`). On the other hand, even people with an extended character set will want to represent string ´15 by `^^M`, since ´15 is *carriage_return*; the idea is to produce visible strings instead of tabs or line-feeds or carriage-returns or bell-rings or characters that are treated anomalously in text files.

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

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

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

This code is used in section 48.

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

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

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

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

This code is used in section 47.

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

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

This code is used in section 51.

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

⟨ Check the pool check sum 53 ⟩ ≡

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

This code is used in section 52.

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

term_and_log, the normal setting, prints on the terminal and on the transcript file.
log_only, prints only on the transcript file.
term_only, prints only on the terminal.
no_print, doesn't print at all. This is used only in rare cases before the transcript file is open.
pseudo, puts output into a cyclic buffer that is used by the *show_context* routine; when we get to that routine we shall discuss the reasoning behind this curious mode.
new_string, appends the output to the current string in the string pool.
0 to 15, prints on one of the sixteen files for `\write` output.

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

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

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

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

55. ⟨ Initialize the output routines 55 ⟩ ≡
selector ← *term_only*; *tally* ← 0; *term_offset* ← 0; *file_offset* ← 0;

See also sections 61, 528, and 533.

This code is used in section 1332.

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, 518, 699, and 1355.

This code is used in section 4.

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

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_char*(*s* : *ASCII_code*); { prints a single character }
  label *exit*;
  begin if ⟨ Character *s* is the current new-line character 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 this part of the program is system dependent.

⟨Initialize the output routines 55⟩ +≡
  wterm(*banner*);
  if *format_ident* = 0 **then** *wterm_ln*(´␣(no␣format␣preloaded)´)
  else begin *slow_print*(*format_ident*); *print_ln*;
    end;
  update_terminal;

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

⟨Basic printing procedures 57⟩ +≡
procedure *print_nl*(*s* : *str_number*); { prints string *s* at beginning of line }
  begin if $((term\_offset > 0) \land (odd(selector))) \lor ((file\_offset > 0) \land (selector \geq log\_only))$ **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 $c \geq 0$ **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 *print_err*(#) ≡
  begin if *interaction* = *error_stop_mode* **then** *wake_up_terminal*;
  print_nl("!␣"); *print*(#);
  end

⟨ Global variables 13 ⟩ +≡
interaction: *batch_mode* .. *error_stop_mode*; { current level of interaction }

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

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

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

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

This code is used in sections 1265 and 1337.

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

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

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

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

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

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

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

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

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

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

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

This code is used in section 4.

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

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

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

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

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

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

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

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

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

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

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

This code is used in section 82.

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

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

⟨ Interpret code *c* and **return** if done 84 ⟩ ≡

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

This code is used in section 83.

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

```
begin print("Type␣<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
  if input_stack[base_ptr].name_field ≥ 256 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 ⟩ +≡
procedure fatal_error(s : str_number); { prints s, and that's it }
  begin normalize_selector;
  print_err("Emergency␣stop"); help1(s); succumb;
  end;
```

94. Here is the most dreaded error message.

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

95. The program might sometime run completely amok, at which point there is no choice but to stop. If no previous error has been detected, that's bad news; a message is printed that is really intended for the 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 ⟩ +≡
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 ≠ 0 then pause_for_instructions;
    end

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

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

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

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

99. Arithmetic with scaled dimensions. The principal computations performed by 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.

```
⟨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 ≥ 0 then
      begin x_over_n ← x div n; remainder ← x mod n;
      end
    else begin x_over_n ← −((−x) div n); remainder ← −((−x) mod n);
      end;
    end;
  if negative then negate(remainder);
  end;
```

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

```
function xn_over_d(x : scaled; n, d : integer): scaled;
  var positive: boolean; {was x ≥ 0?}
    t, u, v: nonnegative_integer; {intermediate quantities}
  begin if x ≥ 0 then positive ← true
  else begin negate(x); positive ← false;
    end;
  t ← (x mod ´100000) ∗ n; u ← (x div ´100000) ∗ n + (t div ´100000);
  v ← (u mod d) ∗ ´100000 + (t mod ´100000);
  if u div d ≥ ´100000 then arith_error ← true
  else u ← ´100000 ∗ (u div d) + (v div d);
  if positive then
    begin xn_over_d ← u; remainder ← v mod d;
    end
  else begin xn_over_d ← −u; remainder ← −(v mod d);
    end;
  end;
```

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

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

It is not difficult to prove that

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

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

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

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

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

```
  define set_glue_ratio_zero(#) ≡ # ← 0.0  { store the representation of zero ratio }
  define set_glue_ratio_one(#) ≡ # ← 1.0  { store the representation of unit ratio }
  define float(#) ≡ #  { convert from glue_ratio to type real }
  define unfloat(#) ≡ #  { convert from real to type glue_ratio }
  define float_constant(#) ≡ #.0  { convert integer constant to real }
⟨ Types in the outer block 18 ⟩ +≡
  glue_ratio = real;  { one-word representation of a glue expansion factor }
```

110. 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* ≡ 0 { smallest allowable value in a *halfword* }
define *max_halfword* ≡ 65535 { 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_min* < *min_halfword*) ∨ (*mem_max* ≥ *max_halfword*) ∨
(*mem_bot* − *mem_min* > *max_halfword* + 1) **then** *bad* ← 14;
if (*font_base* < *min_quarterword*) ∨ (*font_max* > *max_quarterword*) **then** *bad* ← 15;
if *font_max* > *font_base* + 256 **then** *bad* ← 16;
if (*save_size* > *max_halfword*) ∨ (*max_strings* > *max_halfword*) **then** *bad* ← 17;
if *buf_size* > *max_halfword* **then** *bad* ← 18;
if *max_quarterword* − *min_quarterword* < 255 **then** *bad* ← 19;

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* $= 0$.

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

113. The reader should study the following definitions closely:

define *sc* ≡ *int* { *scaled* data is equivalent to *integer* }

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

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

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

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 ≠ null then
    begin r ← p;
    repeat q ← r; r ← link(r);
      stat decr(dyn_used); tats
    until r = null; { now q is the last node on the list }
    link(q) ← avail; avail ← p;
    end;
  end;
```

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

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

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* ← *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 ← *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* ← *max_halfword*; **return**;
    end;
  if *lo_mem_max* + 2 < *hi_mem_min* **then**
    if *lo_mem_max* + 2 ≤ *mem_bot* + *max_halfword* **then**
      ⟨ Grow more variable-size memory and **goto** *restart* 126 ⟩;
  overflow("main␣memory␣size", *mem_max* + 1 − *mem_min*); { sorry, nothing satisfactory is left }
found: *link*(*r*) ← *null*; { this node is now nonempty }
  stat *var_used* ← *var_used* + *s*; { maintain usage statistics }
  tats
  get_node ← *r*;
exit: **end**;

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* ≥ 1998 **then** *t* ← *lo_mem_max* + 1000
  else *t* ← *lo_mem_max* + 1 + (*hi_mem_min* − *lo_mem_max*) **div** 2; { *lo_mem_max* + 2 ≤ *t* < *hi_mem_min* }
  p ← *llink*(*rover*); *q* ← *lo_mem_max*; *rlink*(*p*) ← *q*; *llink*(*rover*) ← *q*;
  if *t* > *mem_bot* + *max_halfword* **then** *t* ← *mem_bot* + *max_halfword*;
  rlink(*q*) ← *rover*; *llink*(*q*) ← *p*; *link*(*q*) ← *empty_flag*; *node_size*(*q*) ← *t* − *lo_mem_max*;
  lo_mem_max ← *t*; *link*(*lo_mem_max*) ← *null*; *info*(*lo_mem_max*) ← *null*; *rover* ← *q*; **goto** *restart*;
  end

This code is used in section 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 $r > p + 1$ **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* }

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*, for historic reasons that are no longer relevant.)

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

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

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*(#) ≡ *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* ≡ *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*(#) ≡ # + 1 { the word where the ligature is to be found }
  define *lig_ptr*(#) ≡ *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*, *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* ≡ *subtype* { how many subsequent nodes to replace }
  define *pre_break* ≡ *llink* { text that precedes a discretionary break }
  define *post_break* ≡ *rlink* { text that follows a discretionary break }

function *new_disc*: *pointer*; { creates an empty *disc_node* }
  var *p*: *pointer*; { the new node }
  begin $p \leftarrow$ *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* ← *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*(#) ≡ *link*(#) { reference count of a glue specification }
define *stretch*(#) ≡ *mem*[# + 2].*sc* { the stretchability of this glob of glue }
define *shrink*(#) ≡ *mem*[# + 3].*sc* { the shrinkability of this glob of glue }
define *stretch_order* ≡ *type* { order of infinity for stretching }
define *shrink_order* ≡ *subtype* { order of infinity for shrinking }
define *fil* = 1 { first-order infinity }
define *fill* = 2 { second-order infinity }
define *filll* = 3 { third-order infinity }

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

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* ← *get_node*(*glue_spec_size*);
mem[*q*] ← *mem*[*p*]; *glue_ref_count*(*q*) ← *null*;
width(*q*) ← *width*(*p*); *stretch*(*q*) ← *stretch*(*p*); *shrink*(*q*) ← *shrink*(*p*); *new_spec* ← *q*;
end;

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* ← *get_node*(*small_node_size*); *type*(*p*) ← *glue_node*; *subtype*(*p*) ← *n* + 1; *leader_ptr*(*p*) ← *null*;
q ← ⟨ Current *mem* equivalent of glue parameter number *n* 224 ⟩; *glue_ptr*(*p*) ← *q*;
incr(*glue_ref_count*(*q*)); *new_param_glue* ← *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* ← *get_node*(*small_node_size*); *type*(*p*) ← *glue_node*; *subtype*(*p*) ← *normal*;
leader_ptr(*p*) ← *null*; *glue_ptr*(*p*) ← *q*; *incr*(*glue_ref_count*(*q*)); *new_glue* ← *p*;
end;

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*(#) ≡ *mem*[# + *glue_offset*].*sc* { total stretch in an unset node }
define *glue_shrink* ≡ *shift_amount* { total shrink in an unset node }
define *span_count* ≡ *subtype* { indicates the number of spanned columns }

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 $k \leftarrow mem\_bot + 1$ **to** $lo\_mem\_stat\_max$ **do** $mem[k].sc \leftarrow 0$; { all glue dimensions are zeroed }
$k \leftarrow mem\_bot$; **while** $k \le lo\_mem\_stat\_max$ **do** { set first words of glue specifications }
begin $glue\_ref\_count(k) \leftarrow null + 1$; $stretch\_order(k) \leftarrow normal$; $shrink\_order(k) \leftarrow normal$;
$k \leftarrow k + glue\_spec\_size$;
end;
$stretch(fil\_glue) \leftarrow unity$; $stretch\_order(fil\_glue) \leftarrow fil$;
$stretch(fill\_glue) \leftarrow unity$; $stretch\_order(fill\_glue) \leftarrow fill$;
$stretch(ss\_glue) \leftarrow unity$; $stretch\_order(ss\_glue) \leftarrow fil$;
$shrink(ss\_glue) \leftarrow unity$; $shrink\_order(ss\_glue) \leftarrow fil$;
$stretch(fil\_neg\_glue) \leftarrow -unity$; $stretch\_order(fil\_neg\_glue) \leftarrow fil$;
$rover \leftarrow lo\_mem\_stat\_max + 1$; $link(rover) \leftarrow empty\_flag$; { now initialize the dynamic memory }
$node\_size(rover) \leftarrow 1000$; { which is a 1000-word available node }
$llink(rover) \leftarrow rover$; $rlink(rover) \leftarrow rover$;
$lo\_mem\_max \leftarrow rover + 1000$; $link(lo\_mem\_max) \leftarrow null$; $info(lo\_mem\_max) \leftarrow null$;
for $k \leftarrow hi\_mem\_stat\_min$ **to** mem_top **do** $mem[k] \leftarrow mem[lo\_mem\_max]$; { clear list heads }
⟨Initialize the special list heads and constant nodes 790⟩;
$avail \leftarrow null$; $mem\_end \leftarrow mem\_top$; $hi\_mem\_min \leftarrow hi\_mem\_stat\_min$;
{ initialize the one-word memory }
$var\_used \leftarrow lo\_mem\_stat\_max + 1 - mem\_bot$; $dyn\_used \leftarrow hi\_mem\_stat\_usage$; { initialize statistics }

See also sections 222, 228, 232, 240, 250, 258, 552, 946, 951, 1216, 1301, and 1369.

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.)

⟨Global variables 13⟩ +≡
debug $free$: **packed array** $[mem\_min \mathrel{..} mem\_max]$ **of** $boolean$; { free cells }
was_free: **packed array** $[mem\_min \mathrel{..} mem\_max]$ **of** $boolean$; { previously free cells }
$was\_mem\_end, was\_lo\_max, was\_hi\_min$: $pointer$; { previous *mem_end*, *lo_mem_max*, and *hi_mem_min* }
$panicking$: $boolean$; { do we want to check memory constantly? }
gubed

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

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 ← mem_min to lo_mem_max do free[p] ← false; { you can probably do this faster }
for p ← hi_mem_min to mem_end do free[p] ← 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 ← mem_min to lo_mem_max do was_free[p] ← free[p];
for p ← hi_mem_min to mem_end do was_free[p] ← free[p]; { was_free ← free might be faster }
was_mem_end ← mem_end; was_lo_max ← lo_mem_max; was_hi_min ← hi_mem_min;
end;
gubed
```

168. ⟨Check single-word *avail* list 168⟩ ≡

```
p ← avail; q ← null; clobbered ← false;
while p ≠ null do
  begin if (p > mem_end) ∨ (p < hi_mem_min) then clobbered ← true
  else if free[p] then clobbered ← true;
  if clobbered then
    begin print_nl("AVAIL␣list␣clobbered␣at␣"); print_int(q); goto done1;
    end;
  free[p] ← true; q ← p; p ← link(q);
  end;
done1:
```

This code is used in section 167.

169. ⟨Check variable-size *avail* list 169⟩ ≡

```
p ← rover; q ← null; clobbered ← false;
repeat if (p ≥ lo_mem_max) ∨ (p < mem_min) then clobbered ← true
  else if (rlink(p) ≥ lo_mem_max) ∨ (rlink(p) < mem_min) then clobbered ← true
    else if ¬(is_empty(p)) ∨ (node_size(p) < 2) ∨ (p + node_size(p) > lo_mem_max) ∨
          (llink(rlink(p)) ≠ p) then clobbered ← true;
  if clobbered then
    begin print_nl("Double-AVAIL␣list␣clobbered␣at␣"); print_int(q); goto done2;
    end;
  for q ← 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] ← true;
    end;
  q ← p; p ← 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 933 ⟩;
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 }

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 ≤ mem_end then
        begin if font(p) ≠ font_in_short_display then
          begin if (font(p) < font_base) ∨ (font(p) > font_max) then print_char("*")
          else ⟨ Print the font identifier for font(p) 267 ⟩;
          print_char("␣"); font_in_short_display ← font(p);
          end;
        print_ASCII(qo(character(p)));
        end;
      end
    else ⟨ Print a short indication of the contents of node p 175 ⟩;
    p ← 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) < font_base) ∨ (font(p) > font_max) then print_char("*")
    else ⟨Print the font identifier for font(p) 267⟩;
    print_char("␣"); 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 691⟩
⟨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 1356 ⟩;
  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 690 ⟩
  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("-␣");
    if abs(mem[p + glue_offset].int) < '4000000 then print("?.?")
    else if abs(g) > float_constant(20000) then
        begin if g > float_constant(0) then print_char(">")
        else print("<␣-");
        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) ≠ 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 ≠ null do
    begin q ← link(p);
    if is_char_node(p) then free_avail(p)
    else begin case type(p) of
      hlist_node, vlist_node, unset_node: begin flush_node_list(list_ptr(p)); free_node(p, box_node_size);
        goto done;
        end;
      rule_node: begin free_node(p, rule_node_size); goto done;
        end;
      ins_node: begin flush_node_list(ins_ptr(p)); delete_glue_ref(split_top_ptr(p));
        free_node(p, ins_node_size); goto done;
        end;
      whatsit_node: ⟨Wipe out the whatsit node p and goto done 1358⟩;
      glue_node: begin fast_delete_glue_ref(glue_ptr(p));
        if leader_ptr(p) ≠ null then flush_node_list(leader_ptr(p));
        end;
      kern_node, math_node, penalty_node: do_nothing;
      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 698⟩
      othercases confusion("flushing")
      endcases;
      free_node(p, small_node_size);
    done: end;
    p ← 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 1357 ⟩;
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 }

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`'.

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 *def_font* = 88 { define a font file (`\font`) }
define *register* = 89 { internal register (`\count`, `\dimen`, etc.) }
define *max_internal* = 89 { the largest code that can follow `\the` }
define *advance* = 90 { advance a register or parameter (`\advance`) }
define *multiply* = 91 { multiply a register or parameter (`\multiply`) }
define *divide* = 92 { divide a register or parameter (`\divide`) }
define *prefix* = 93 { qualify a definition (`\global`, `\long`, `\outer`) }
define *let* = 94 { assign a command code (`\let`, `\futurelet`) }
define *shorthand_def* = 95 { code definition (`\chardef`, `\countdef`, etc.) }
define *read_to_cs* = 96 { read into a control sequence (`\read`) }
define *def* = 97 { macro definition (`\def`, `\gdef`, `\xdef`, `\edef`) }
define *set_box* = 98 { set a box (`\setbox`) }
define *hyph_data* = 99 { hyphenation data (`\hyphenation`, `\patterns`) }
define *set_interaction* = 100 { define level of interaction (`\batchmode`, etc.) }
define *max_command* = 100 { the largest command code seen at *big_switch* }

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

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

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

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

procedure print_mode(m : integer); { prints the mode represented by m }
  begin if m > 0 then
    case m div (max_command + 1) of
    0: print("vertical");
    1: print("horizontal");
    2: print("display␣math");
    end
  else if m = 0 then print("no")
    else case (−m) div (max_command + 1) of
      0: print("internal␣vertical");
      1: print("restricted␣horizontal");
      2: print("math");
      end;
  print("␣mode");
  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: **array** [0 .. *nest_size*] **of** *list_state_record*;
nest_ptr: 0 .. *nest_size*; { first unused location of *nest* }
max_nest_stack: 0 .. *nest_size*; { maximum of *nest_ptr* when pushing }
cur_list: *list_state_record*; { the "top" semantic state }
shown_mode: −*mmode* .. *mmode*; { most recent mode shown by `\tracingcommands` }

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;
  ⟨ Start a new current page 991 ⟩;

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

procedure *push_nest*; { enter a new semantic level, save the old }
  begin if *nest_ptr* > *max_nest_stack* **then**
    begin *max_nest_stack* ← *nest_ptr*;
    if *nest_ptr* = *nest_size* **then** *overflow*("semantic␣nest␣size", *nest_size*);
    end;
  nest[*nest_ptr*] ← *cur_list*; { stack the record }
  incr(*nest_ptr*); *head* ← *get_avail*; *tail* ← *head*; *prev_graf* ← 0; *mode_line* ← *line*;
  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 986⟩;
      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); print("␣line");
      if nest[p].pg_field ≠ 1 then print_char("s");
      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␣begin␣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* .. (*hash_base* − 1)] holds the current equivalents of single-character control sequences.
2) *eqtb*[*hash_base* .. (*glue_base* − 1)] holds the current equivalents of multiletter control sequences.
3) *eqtb*[*glue_base* .. (*local_base* − 1)] holds the current equivalents of glue parameters like the current baselineskip.
4) *eqtb*[*local_base* .. (*int_base* − 1)] holds the current equivalents of local halfword quantities like the current box registers, the current "catcodes," the current font, and a pointer to the current paragraph shape.
5) *eqtb*[*int_base* .. (*dimen_base* − 1)] holds the current equivalents of fullword integer parameters like the current hyphenation penalty.
6) *eqtb*[*dimen_base* .. *eqtb_size*] holds the current equivalents of fullword dimension parameters like the current hsize or amount of hanging indentation.

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

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

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 256 equivalents for "active characters" that act as control sequences, followed by 256 equivalents for single-character control sequences.

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

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

⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
eq_type(*undefined_control_sequence*) ← *undefined_cs*; *equiv*(*undefined_control_sequence*) ← *null*;
eq_level(*undefined_control_sequence*) ← *level_zero*;
for *k* ← *active_base* **to** *undefined_control_sequence* − 1 **do** *eqtb*[*k*] ← *eqtb*[*undefined_control_sequence*];

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*) ≥ *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, 376, 384, 411, 416, 468, 487, 491, 553, 780, 983, 1052, 1058, 1071, 1088, 1107, 1114, 1141, 1156, 1169, 1178, 1188, 1208, 1219, 1222, 1230, 1250, 1254, 1262, 1272, 1277, 1286, 1291, and 1344.

This code is used in section 1336.

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, 377, 385, 412, 417, 469, 488, 492, 781, 984, 1053, 1059, 1072, 1089, 1108, 1115, 1143, 1157, 1170, 1179, 1189, 1209, 1220, 1223, 1231, 1251, 1255, 1261, 1263, 1273, 1278, 1287, 1292, 1295, and 1346.

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 $k \leftarrow glue\_base + 1$ **to** $local\_base - 1$ **do** $eqtb[k] \leftarrow eqtb[glue\_base]$;
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.

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 *math_font_base* = *cur_font_loc* + 1 { table of 48 math font numbers }
define *cat_code_base* = *math_font_base* + 48 { table of 256 command codes (the "catcodes") }
define *lc_code_base* = *cat_code_base* + 256 { table of 256 lowercase mappings }
define *uc_code_base* = *lc_code_base* + 256 { table of 256 uppercase mappings }
define *sf_code_base* = *uc_code_base* + 256 { table of 256 spacefactor mappings }
define *math_code_base* = *sf_code_base* + 256 { table of 256 math mode mappings }
define *int_base* = *math_code_base* + 256 { beginning of region 5 }

define *par_shape_ptr* ≡ *equiv*(*par_shape_loc*)
define *output_routine* ≡ *equiv*(*output_routine_loc*)
define *every_par* ≡ *equiv*(*every_par_loc*)
define *every_math* ≡ *equiv*(*every_math_loc*)
define *every_display* ≡ *equiv*(*every_display_loc*)
define *every_hbox* ≡ *equiv*(*every_hbox_loc*)
define *every_vbox* ≡ *equiv*(*every_vbox_loc*)
define *every_job* ≡ *equiv*(*every_job_loc*)
define *every_cr* ≡ *equiv*(*every_cr_loc*)
define *err_help* ≡ *equiv*(*err_help_loc*)
define *toks*(#) ≡ *equiv*(*toks_base* + #)
define *box*(#) ≡ *equiv*(*box_base* + #)
define *cur_font* ≡ *equiv*(*cur_font_loc*)
define *fam_fnt*(#) ≡ *equiv*(*math_font_base* + #)
define *cat_code*(#) ≡ *equiv*(*cat_code_base* + #)
define *lc_code*(#) ≡ *equiv*(*lc_code_base* + #)
define *uc_code*(#) ≡ *equiv*(*uc_code_base* + #)
define *sf_code*(#) ≡ *equiv*(*sf_code_base* + #)
define *math_code*(#) ≡ *equiv*(*math_code_base* + #)
{ Note: *math_code*(*c*) is the true math code plus *min_halfword* }

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

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

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

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

 define *null_font* ≡ *font_base*
 define *var_code* ≡ ´*70000* { math code meaning "use the current family" }

⟨ Initialize table entries (done by INITEX only) 164 ⟩ +≡
 par_shape_ptr ← *null*; *eq_type*(*par_shape_loc*) ← *shape_ref*; *eq_level*(*par_shape_loc*) ← *level_one*;
 for *k* ← *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 < 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 $n < lc\_code\_base$ **then**
    begin *print_esc*("`catcode`"); *print_int*($n - cat\_code\_base$);
    end
  else if $n < uc\_code\_base$ **then**
      begin *print_esc*("`lccode`"); *print_int*($n - lc\_code\_base$);
      end
    else if $n < sf\_code\_base$ **then**
        begin *print_esc*("`uccode`"); *print_int*($n - uc\_code\_base$);
        end
      else begin *print_esc*("`sfcode`"); *print_int*($n - sf\_code\_base$);
        end;
  print_char("`=`"); *print_int*(*equiv*(*n*));
  end
else begin *print_esc*("`mathcode`"); *print_int*($n - math\_code\_base$); *print_char*("`=`");
  print_int(*ho*(*equiv*(*n*)));
  end

This code is used in section 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 held over after a split }
define *global_defs_code* = 43 { override `\global` specifications }
define *cur_fam_code* = 44 { current family }
define *escape_char_code* = 45 { escape character for token output }
define *default_hyphen_char_code* = 46 { value of `\hyphenchar` when a font is loaded }

define *default_skew_char_code* = 47 { value of `\skewchar` when a font is loaded }
define *end_line_char_code* = 48 { character placed at the right end of the buffer }
define *new_line_char_code* = 49 { character that prints as *print_ln* }
define *language_code* = 50 { current hyphenation table }
define *left_hyphen_min_code* = 51 { minimum left hyphenation fragment size }
define *right_hyphen_min_code* = 52 { minimum right hyphenation fragment size }
define *holding_inserts_code* = 53 { do not remove insertion nodes from `\box255` }
define *error_context_lines_code* = 54 { maximum intermediate line pairs shown }
define *int_pars* = 55 { 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*)

⟨ 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");
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*);

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;
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 }

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

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

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* ← *selector*;
  if (*tracing_online* ≤ 0) ∧ (*selector* = *term_and_log*) **then**
    begin *decr*(*selector*);
    if *history* = *spotless* **then** *history* ← *warning_issued*;
    end;
  end;

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

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

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

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 $n < glue\_base$ **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 $n \le eqtb\_size$ **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 ⟩ +≡
eqtb: **array** [*active_base* .. *eqtb_size*] **of** *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: **array** [*hash_base* .. *undefined_control_sequence* − 1] **of** *two_halves*; { the hash table }
hash_used: *pointer*; { allocation pointer for *hash* }
no_new_control_sequence: *boolean*; { are new identifiers legal? }
cs_count: *integer*; { total number of known identifiers }

257. ⟨ Set initial values of key variables 21 ⟩ +≡
no_new_control_sequence ← *true*; { new identifiers are usually forbidden }
next(*hash_base*) ← 0; *text*(*hash_base*) ← 0;
for k ← *hash_base* + 1 **to** *undefined_control_sequence* − 1 **do** *hash*[k] ← *hash*[*hash_base*];

258. ⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
hash_used ← *frozen_control_sequence*; { nothing is used }
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 $buffer[j\ ..\ (j + l - 1)]$. 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 repeat if hash_is_full then overflow("hash␣size", hash_size);
      decr(hash_used);
    until text(hash_used) = 0; { search for an empty location in hash }
    next(p) ← hash_used; p ← hash_used;
    end;
  str_room(l); d ← cur_length;
  while pool_ptr > str_start[str_ptr] do
    begin decr(pool_ptr); str_pool[pool_ptr + l] ← str_pool[pool_ptr];
    end; { move current string up to make room for another }
  for k ← j to j + l − 1 do append_char(buffer[k]);
  text(p) ← make_string; pool_ptr ← pool_ptr + d;
  stat incr(cs_count); tats
  end
```

This code is used in section 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"); print_char("␣");
        end
      else begin print_esc(p − single_base);
        if cat_code(p − single_base) = letter then print_char("␣");
        end
    else if p < active_base then print_esc("IMPOSSIBLE.")
      else print(p − active_base)
  else if p ≥ undefined_control_sequence then print_esc("IMPOSSIBLE.")
    else if (text(p) < 0) ∨ (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** $cur\_val \leftarrow s + single\_base$
else begin $k \leftarrow str\_start[s]$; $l \leftarrow str\_start[s+1] - k$; { we will move *s* into the (empty) *buffer* }
  for $j \leftarrow 0$ **to** $l - 1$ **do** $buffer[j] \leftarrow so(str\_pool[k+j])$;
  $cur\_val \leftarrow id\_lookup(0, l)$; { *no_new_control_sequence* is *false* }
  flush_string; $text(cur\_val) \leftarrow s$; { we don't want to have the string twice }
  end;
$eq\_level(cur\_val) \leftarrow level\_one$; $eq\_type(cur\_val) \leftarrow c$; $equiv(cur\_val) \leftarrow o$;
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");
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*) $\geq$ *int_base*, then *save_level*(*p*) should replace the corresponding entry in *xeq_level*.

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

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

4) If *save_type*(*p*) = *level_boundary*, then *save_level*(*p*) is a code explaining what kind of group we were previously in, and *save_index*(*p*) points to the level boundary word at the bottom of the entries for that group.

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

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*;
  $p \leftarrow$ *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 $l \leftarrow$ *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* **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 $2^8m + c$; the command code is in the range $1 \le m \le 14$. (2) A control sequence whose *eqtb* address is p is represented as the number $cs\_token\_flag + p$. Here $cs\_token\_flag = 2^{12} - 1$ is larger than $2^8m + c$, yet it is small enough that $cs\_token\_flag + p < max\_halfword$; thus, a token fits comfortably in a halfword.

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

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

290. ⟨ Check the "constant" values for consistency 14 ⟩ +≡
if $cs\_token\_flag + undefined\_control\_sequence > max\_halfword$ **then** $bad \leftarrow 21$;

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 an output routine will be enclosed in braces.

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

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

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

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

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

Examples such as

`\def\m{\def\m{a}␣b}`

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

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

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 ≠ 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 ≠ 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 '400; c ← info(p) mod '400;
    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
left_brace, right_brace, math_shift, tab_mark, sup_mark, sub_mark, spacer, letter, other_char: print(c);
mac_param: begin print(c); print(c);
  end;
out_param: begin print(match_chr);
  if c ≤ 9 then print_char(c + "0")
  else begin print_char("!"); return;
    end;
  end;
match: begin match_chr ← c; print(c); incr(n); print_char(n);
  if n > "9" then return;
  end;
end_match: 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

⟨ 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: *chr_cmd*("the␣letter␣");
other_char: *chr_cmd*("the␣character␣");
⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩
othercases *print*("[unknown␣command␣code!]")
endcases;
end;

This code is used in section 252.

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

procedure *show_cur_cmd_chr*;
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: **array** [0 .. *stack_size*] **of** *in_state_record*;
input_ptr: 0 .. *stack_size*; { first unused location of *input_stack* }
max_in_stack: 0 .. *stack_size*; { largest value of *input_ptr* when pushing }
cur_input: *in_state_record*; { the "top" input state, according to convention (1) }

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: **array** [1 .. *max_in_open*] **of** *alpha_file*;
line: *integer*; { current line number in the current source file }
line_stack: **array** [1 .. *max_in_open*] **of** *integer*;

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 *print_nl*("Runaway␣");
    case *scanner_status* **of**
    defining: **begin** *print*("definition"); *p* ← *def_ref*;
      end;
    matching: **begin** *print*("argument"); *p* ← *temp_head*;
      end;
    aligning: **begin** *print*("preamble"); *p* ← *hold_head*;
      end;
    absorbing: **begin** *print*("text"); *p* ← *def_ref*;
      end;
    end; { there are no other cases }
    print_char("?"); *print_ln*; *show_token_list*(*link*(*p*), *null*, *error_line* − 10);
    end;
  end;

This code is used in section 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* ≡ *index* { type of current token list }
define *param_start* ≡ *limit* { base of macro parameters in *param_stack* }
define *parameter* = 0 { *token_type* code for parameter }
define *u_template* = 1 { *token_type* code for $\langle u_j\rangle$ template }
define *v_template* = 2 { *token_type* code for $\langle v_j\rangle$ template }
define *backed_up* = 3 { *token_type* code for text to be reread }
define *inserted* = 4 { *token_type* code for inserted texts }
define *macro* = 5 { *token_type* code for defined control sequences }
define *output_text* = 6 { *token_type* code for output routines }
define *every_par_text* = 7 { *token_type* code for `\everypar` }
define *every_math_text* = 8 { *token_type* code for `\everymath` }
define *every_display_text* = 9 { *token_type* code for `\everydisplay` }
define *every_hbox_text* = 10 { *token_type* code for `\everyhbox` }
define *every_vbox_text* = 11 { *token_type* code for `\everyvbox` }
define *every_job_text* = 12 { *token_type* code for `\everyjob` }
define *every_cr_text* = 13 { *token_type* code for `\everycr` }

define *mark_text* = 14 { *token_type* code for `\topmark`, etc. }
define *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: **array** [0 . . *param_size*] **of** *pointer*; { token list pointers for parameters }
param_ptr: 0 . . *param_size*; { first unused entry in *param_stack* }
max_param_stack: *integer*; { largest value of *param_ptr*, will be $\leq$ *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* $[k \mathbf{\ mod\ } error\_line]$ if $k < trick\_count$, otherwise character k is dropped. Initially we set $tally \leftarrow 0$ and $trick\_count \leftarrow 1000000$; then when we reach the point where transition from line 1 to line 2 should occur, we set $first\_count \leftarrow tally$ and $trick\_count \leftarrow \max(error\_line, tally + 1 + error\_line - half\_error\_line)$. 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 $k = first\_count$, and the length of the context information gathered for line 2 is $m = \min(tally, trick\_count) - k$. If $l + k \le h$, where $h = half\_error\_line$, we print $trick\_buf[0 \mathrel{..} k-1]$ 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

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

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

⟨Local variables for formatting calculations 315⟩ ≡
i: 0 .. *buf_size*; { index into *buffer* }
j: 0 .. *buf_size*; { end of current line in *buffer* }
l: 0 .. *half_error_line*; { length of descriptive information on line 1 }
m: *integer*; { context information gathered for line 2 }
n: 0 .. *error_line*; { length of line 1 }
p: *integer*; { starting or ending place in *trick_buf* }
q: *integer*; { temporary index }

This code is used in section 311.

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

define *begin_pseudoprint* ≡
  begin $l \leftarrow tally$; $tally \leftarrow 0$; $selector \leftarrow pseudo$; $trick\_count \leftarrow 1000000$;
  end
define *set_trick_count* ≡
  begin $first\_count \leftarrow tally$; $trick\_count \leftarrow tally + 1 + error\_line - half\_error\_line$;
  if $trick\_count < error\_line$ **then** $trick\_count \leftarrow error\_line$;
  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* ← *tally* − *first_count*
else *m* ← *trick_count* − *first_count*; { context on line 2 }
if *l* + *first_count* ≤ *half_error_line* **then**
begin *p* ← 0; *n* ← *l* + *first_count*;
end
else begin *print*("..."); *p* ← *l* + *first_count* − *half_error_line* + 3; *n* ← *half_error_line*;
end;
for *q* ← *p* **to** *first_count* − 1 **do** *print_char*(*trick_buf*[*q* **mod** *error_line*]);
print_ln;
for *q* ← 1 **to** *n* **do** *print_char*("␣"); { print *n* spaces to begin line 2 }
if *m* + *n* ≤ *error_line* **then** *p* ← *first_count* + *m*
else *p* ← *first_count* + (*error_line* − *n* − 3);
for *q* ← *first_count* **to** *p* − 1 **do** *print_char*(*trick_buf*[*q* **mod** *error_line*]);
if *m* + *n* > *error_line* **then** *print*("...")

This code is used in section 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* ← *limit*
else *j* ← *limit* + 1; { determine the effective end of the line }
if *j* > 0 **then**
for *i* ← *start* **to** *j* − 1 **do**
begin if *i* = *loc* **then** *set_trick_count*;
print(*buffer*[*i*]);
end

This code is used in section 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 ≥ backed_up then   { token list to be deleted }
    begin if token_type ≤ inserted then flush_list(start)
    else begin delete_token_ref(start);   { update reference count }
      if token_type = macro then   { parameters must be flushed }
        while param_ptr > param_start do
          begin decr(param_ptr); flush_list(param_stack[param_ptr]);
          end;
      end;
    end
  else if token_type = u_template then
      if align_state > 500000 then align_state ← 0
      else fatal_error("(interwoven␣alignment␣preambles␣are␣not␣allowed)");
  pop_input; check_interrupt;
  end;
```

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* ← *in_open*; *line_stack*[*index*] ← *line*; *start* ← *first*; *state* ← *mid_line*;
  name ← 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* ← *start*; *line* ← *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* ≠ *token_list*) ∧ *terminal_input* ∧ (*input_ptr* > 0) ∧ (*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* ← 0; *max_in_stack* ← 0; *in_open* ← 0; *open_parens* ← 0; *max_buf_stack* ← 0;
  param_ptr ← 0; *max_param_stack* ← 0; *first* ← *buf_size*;
  repeat *buffer*[*first*] ← 0; *decr*(*first*);
  until *first* = 0;
  scanner_status ← *normal*; *warning_index* ← *null*; *first* ← 1; *state* ← *new_line*; *start* ← 1; *index* ← 0;
  line ← 0; *name* ← 0; *force_eof* ← *false*; *align_state* ← 1000000;
  if ¬*init_terminal* **then goto** *final_end*;
  limit ← *last*; *first* ← *last* + 1; { *init_terminal* has set *loc* and *last* }
  end

This code is used in section 1337.

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("␣while␣scanning␣"); ⟨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 or a runaway balanced text we will insert a right brace; for a runaway preamble, we will insert a special `\cr` token and a right brace; and for a runaway argument, we will set *long_state* to *outer_call* and insert `\par`.

⟨Print either '`definition`' or '`use`' or '`preamble`' or '`text`', and insert tokens that should lead to recovery 339⟩ ≡
p ← *get_avail*;
case *scanner_status* **of**
defining: **begin** *print*("definition"); *info*(*p*) ← *right_brace_token* + "}";
end;
matching: **begin** *print*("use"); *info*(*p*) ← *par_token*; *long_state* ← *outer_call*;
end;
aligning: **begin** *print*("preamble"); *info*(*p*) ← *right_brace_token* + "}"; *q* ← *p*; *p* ← *get_avail*;
link(*p*) ← *q*; *info*(*p*) ← *cs_token_flag* + *frozen_cr*; *align_state* ← −1000000;
end;
absorbing: **begin** *print*("text"); *info*(*p*) ← *right_brace_token* + "}";
end;
end; { there are no other cases }
ins_list(*p*)

This code is used in section 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 }
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* 789⟩

This code is used in section 341.

343. ⟨Input from external file, **goto** *restart* if no input found 343⟩ ≡
begin *switch*: **if** *loc* ≤ *limit* **then** { current line not yet finished }
begin *cur_chr* ← *buffer*[*loc*]; *incr*(*loc*);
reswitch: *cur_cmd* ← *cat_code*(*cur_chr*); ⟨Change state if necessary, and **goto** *switch* if the current character should be ignored, or **goto** *reswitch* if the current character changes to another 344⟩;
end
else begin *state* ← *new_line*;
⟨Move to next line of file, or **goto** *restart* if there is no next line, or **return** if a `\read` line has finished 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* ← *skip_blanks* or *mid_line* 354 ⟩;
any_state_plus(*active_char*): ⟨ Process an active-character control sequence and set *state* ← *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* ← *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*(#) ≡ (((# ≥ "0") ∧ (# ≤ "9")) ∨ ((# ≥ "a") ∧ (# ≤ "f")))
define *hex_to_cur_chr* ≡
    if *c* ≤ "9" **then** *cur_chr* ← *c* − "0" **else** *cur_chr* ← *c* − "a" + 10;
  if *cc* ≤ "9" **then** *cur_chr* ← 16 ∗ *cur_chr* + *cc* − "0"
  else *cur_chr* ← 16 ∗ *cur_chr* + *cc* − "a" + 10

⟨ If this *sup_mark* starts an expanded character like `^^A` or `^^df`, then **goto** *reswitch*, otherwise set *state* ← *mid_line* 352 ⟩ ≡

```
begin if cur_chr = buffer[loc] then
  if loc < limit then
    begin c ← buffer[loc + 1]; if c < '200 then   { yes we have an expanded char }
      begin loc ← loc + 2;
      if is_hex(c) then
        if loc ≤ limit then
          begin cc ← buffer[loc]; if is_hex(cc) then
            begin incr(loc); hex_to_cur_chr; goto reswitch;
            end;
          end;
      if c < '100 then cur_chr ← c + '100 else cur_chr ← c − '100;
      goto reswitch;
      end;
    end;
  state ← mid_line;
end
```

This code is used in section 344.

353. ⟨Process an active-character control sequence and set *state* ← *mid_line* 353⟩ ≡
begin *cur_cs* ← *cur_chr* + *active_base*; *cur_cmd* ← *eq_type*(*cur_cs*); *cur_chr* ← *equiv*(*cur_cs*);
state ← *mid_line*;
if *cur_cmd* ≥ *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.

⟨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 *start_cs*: *k* ← *loc*; *cur_chr* ← *buffer*[*k*]; *cat* ← *cat_code*(*cur_chr*); *incr*(*k*);
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⟩;
cur_cs ← *single_base* + *buffer*[*loc*]; *incr*(*loc*);
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 $buffer[(k-1)\,..\,(k+1)]$ or $buffer[(k-1)\,..\,(k+2)]$, we will store the corresponding code in $buffer[k-1]$ and shift the rest of the buffer left two or three places.

⟨If an expanded code is present, reduce it and **goto** *start_cs* 355⟩ ≡

```
begin if buffer[k] = cur_chr then if cat = sup_mark then if k < limit then
    begin c ← buffer[k + 1]; if c < ´200 then   { yes, one is indeed present }
      begin d ← 2;
      if is_hex(c) then if k + 2 ≤ limit then
          begin cc ← buffer[k + 2]; if is_hex(cc) then incr(d);
          end;
      if d > 2 then
        begin hex_to_cur_chr; buffer[k − 1] ← cur_chr;
        end
      else if c < ´100 then buffer[k − 1] ← c + ´100
        else buffer[k − 1] ← c − ´100;
      limit ← limit − d; first ← first − d;
      while k ≤ limit do
        begin buffer[k] ← buffer[k + d]; incr(k);
        end;
      goto start_cs;
      end;
    end;
end
```

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 cur_chr ← buffer[k]; cat ← cat_code(cur_chr); incr(k);
until (cat ≠ letter) ∨ (k > limit);
⟨If an expanded code is present, reduce it and goto start_cs 355⟩;
if cat ≠ letter then decr(k);   { now k points to first nonletter }
if k > loc + 1 then   { multiletter control sequence has been scanned }
  begin cur_cs ← id_lookup(loc, k − loc); loc ← k; goto found;
  end;
end
```

This code is used in section 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** ´400; $cur\_chr \leftarrow t$ **mod** ´400;
    case *cur_cmd* **of**
    left_brace: *incr*(*align_state*);
    right_brace: *decr*(*align_state*);
    out_param: ⟨Insert macro parameter and **goto** *restart* 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* ≡ (*end_line_char* < 0) ∨ (*end_line_char* > 255)

⟨Move to next line of file, or **goto** *restart* if there is no next line, or **return** if a `\read` line has finished 360⟩ ≡

```
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⟩ +≡
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⟩ ≡

```
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 ∗ '400) + 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*.

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

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

```
⟨Declare the procedure called macro_call 389⟩
⟨Declare the procedure called insert_relax 379⟩
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 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 367⟩
  else if cur_cmd < end_template then macro_call
    else ⟨Insert a token containing frozen_endv 375⟩;
  cur_val ← cv_backup; cur_val_level ← cvl_backup; radix ← radix_backup; cur_order ← co_backup;
  link(backup_head) ← backup_backup;
  end;
```

367. ⟨Expand a nonmacro 367⟩ ≡

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

This code is used in section 366.

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

⟨ Expand the token after the next token 368 ⟩ ≡
begin *get_token*; $t \gets cur\_tok$; *get_token*;
if $cur\_cmd > max\_command$ **then** *expand* **else** *back_input*;
$cur\_tok \gets t$; *back_input*;
end

This code is used in section 367.

369. 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 369 ⟩ ≡
begin $save\_scanner\_status \gets scanner\_status$; $scanner\_status \gets normal$; *get_token*;
$scanner\_status \gets save\_scanner\_status$; $t \gets cur\_tok$; *back_input*;
{ now *start* and *loc* point to the backed-up token *t* }
if $t \geq cs\_token\_flag$ **then**
begin $p \gets get\_avail$; $info(p) \gets cs\_token\_flag + frozen\_dont\_expand$; $link(p) \gets loc$; $start \gets p$;
$loc \gets p$;
end;
end

This code is used in section 367.

370. ⟨ Complain about an undefined macro 370 ⟩ ≡
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 367.

371. 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

372. ⟨ Manufacture a control sequence name 372 ⟩ ≡

```
begin r ← get_avail; p ← r; { head of the list of characters }
repeat get_x_token;
  if cur_cs = 0 then store_new_token(cur_tok);
until cur_cs ≠ 0;
if cur_cmd ≠ end_cs_name then ⟨ Complain about missing \endcsname 373 ⟩;
⟨ Look up the characters of list r in the hash table, and set cur_cs 374 ⟩;
flush_list(r);
if eq_type(cur_cs) = undefined_cs then
  begin eq_define(cur_cs, relax, 256); { N.B.: The save_stack might change }
  end; { the control sequence will now match '\relax' }
cur_tok ← cur_cs + cs_token_flag; back_input;
end
```

This code is used in section 367.

373. ⟨ Complain about missing \endcsname 373 ⟩ ≡

```
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 372.

374. ⟨ Look up the characters of list r in the hash table, and set cur_cs 374 ⟩ ≡

```
j ← first; p ← link(r);
while p ≠ null do
  begin if j ≥ max_buf_stack then
    begin max_buf_stack ← j + 1;
    if max_buf_stack = buf_size then overflow("buffer␣size", buf_size);
    end;
  buffer[j] ← info(p) mod '400; incr(j); p ← link(p);
  end;
if j > first + 1 then
  begin no_new_control_sequence ← false; cur_cs ← id_lookup(first, j − first);
  no_new_control_sequence ← true;
  end
else if j = first then cur_cs ← null_cs { the list is empty }
  else cur_cs ← single_base + buffer[first] { the list has length one }
```

This code is used in section 372.

375. 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* 375 ⟩ ≡

```
begin cur_tok ← cs_token_flag + frozen_endv; back_input;
end
```

This code is used in section 366.

376. 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);

377. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
input: **if** *chr_code* = 0 **then** *print_esc*("input") **else** *print_esc*("endinput");

378. ⟨Initiate or terminate input from a file 378⟩ ≡
if *cur_chr* > 0 **then** *force_eof* ← *true*
else if *name_in_progress* **then** *insert_relax*
else *start_input*

This code is used in section 367.

379. 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* 379⟩ ≡
procedure *insert_relax*;
begin *cur_tok* ← *cs_token_flag* + *cur_cs*; *back_input*; *cur_tok* ← *cs_token_flag* + *frozen_relax*; *back_input*;
token_type ← *inserted*;
end;

This code is used in section 366.

380. Here is a recursive procedure that is TEX's usual way to get the next token of input. It has been slightly optimized to take account of common cases.

procedure *get_x_token*; { sets *cur_cmd*, *cur_chr*, *cur_tok*, and expands macros }
label *restart*, *done*;
begin *restart*: *get_next*;
if *cur_cmd* ≤ *max_command* **then goto** *done*;
if *cur_cmd* ≥ *call* **then**
if *cur_cmd* < *end_template* **then** *macro_call*
else begin *cur_cs* ← *frozen_endv*; *cur_cmd* ← *endv*; **goto** *done*; { *cur_chr* = *null_list* }
end
else *expand*;
goto *restart*;
done: **if** *cur_cs* = 0 **then** *cur_tok* ← (*cur_cmd* ∗ ´400) + *cur_chr*
else *cur_tok* ← *cs_token_flag* + *cur_cs*;
end;

381. The *get_x_token* procedure is essentially equivalent to two consecutive procedure calls: *get_next*; *x_token*.

procedure *x_token*; { *get_x_token* without the initial *get_next* }
begin while *cur_cmd* > *max_command* **do**
begin *expand*; *get_next*;
end;
if *cur_cs* = 0 **then** *cur_tok* ← (*cur_cmd* ∗ ´400) + *cur_chr*
else *cur_tok* ← *cs_token_flag* + *cur_cs*;
end;

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

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

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

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

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

384. ⟨ 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*);

385. ⟨ 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;

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

⟨ Insert the appropriate mark text into the scanner 386 ⟩ ≡
begin if *cur_mark*[*cur_chr*] ≠ *null* **then** *begin_token_list*(*cur_mark*[*cur_chr*], *mark_text*);
end

This code is used in section 367.

387. 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` }

388. 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 }

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

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

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

This code is used in section 366.

390. 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 390⟩ ≡
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 389.

391. 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 391⟩ ≡
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 392⟩;
    { 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 389.

392. 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 392⟩ ≡
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* 394⟩;
  ⟨Contribute the recently matched tokens to the current parameter, and **goto** *continue* if a partial match is still in effect; but abort if $s = null$ 397⟩;
  if $cur\_tok = par\_token$ **then**
    if $long\_state \neq long\_call$ **then** ⟨Report a runaway argument and abort 396⟩;
  if $cur\_tok < right\_brace\_limit$ **then**
    if $cur\_tok < left\_brace\_limit$ **then** ⟨Contribute an entire group to the current parameter 399⟩
    else ⟨Report an extra right brace and **goto** *continue* 395⟩
  else ⟨Store the current token, but **goto** *continue* if it is a blank space that would become an undelimited parameter 393⟩;
  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 400⟩

This code is used in section 391.

393. ⟨Store the current token, but **goto** *continue* if it is a blank space that would become an undelimited parameter 393⟩ ≡
  begin if $cur\_tok = space\_token$ **then**
    if $info(r) \leq end\_match\_token$ **then**
      if $info(r) \geq match\_token$ **then goto** *continue*;
  store_new_token(*cur_tok*);
  end

This code is used in section 392.

394. 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* 394⟩ ≡
  begin $r \leftarrow link(r)$;
  if $(info(r) \geq match\_token) \land (info(r) \leq end\_match\_token)$ **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 392.

395. ⟨Report an extra right brace and **goto** *continue* 395⟩ ≡

```
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 392.

396. If *long_state* = *outer_call*, a runaway argument has already been reported.

⟨Report a runaway argument and abort 396⟩ ≡

```
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 392 and 399.

397. 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 $s = null$ 397 ⟩ ≡

```
if s ≠ r then
  if s = null then ⟨ Report an improper use of the macro and abort 398 ⟩
  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 392.

398. ⟨ Report an improper use of the macro and abort 398 ⟩ ≡

```
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 397.

399. ⟨ Contribute an entire group to the current parameter 399 ⟩ ≡

```
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 396 ⟩;
  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 392.

400. 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 400 ⟩ ≡
begin if $(m = 1) \land (info(p) < right\_brace\_limit)$ **then**
begin $link(rbrace\_ptr) \leftarrow null$; $free\_avail(p)$; $p \leftarrow link(temp\_head)$; $pstack[n] \leftarrow link(p)$; $free\_avail(p)$;
end
else $pstack[n] \leftarrow link(temp\_head)$;
$incr(n)$;
if $tracing\_macros > 0$ **then**
begin *begin_diagnostic*; $print\_nl(match\_chr)$; $print\_int(n)$; *print*("<-");
$show\_token\_list(pstack[n - 1], null, 1000)$; $end\_diagnostic(false)$;
end;
end

This code is used in section 392.

401. ⟨ Show the text of the macro being expanded 401 ⟩ ≡
begin *begin_diagnostic*; *print_ln*; $print\_cs(warning\_index)$; $token\_show(ref\_count)$;
$end\_diagnostic(false)$;
end

This code is used in section 389.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
procedure scan_int; forward; { scans an integer value }
⟨ Declare procedures that scan restricted classes of integers 433 ⟩
⟨ Declare procedures that scan font-related stuff 577 ⟩
```

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

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

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

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

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

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

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

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

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

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

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

414. ⟨ Fetch a character code from some table 414 ⟩ ≡
begin *scan_char_num*;
if *m* = *math_code_base* **then** *scanned_result*(*ho*(*math_code*(*cur_val*)))(*int_val*)
else if *m* < *math_code_base* **then** *scanned_result*(*equiv*(*m* + *cur_val*))(*int_val*)
else *scanned_result*(*eqtb*[*m* + *cur_val*].*int*)(*int_val*);
end

This code is used in section 413.

415. ⟨ Fetch a token list or font identifier, provided that *level* = *tok_val* 415 ⟩ ≡

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

This code is used in section 413.

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

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

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

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

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

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

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

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

This code is used in section 413.

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

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

This code is used in section 413.

420. ⟨ Fetch a box dimension 420 ⟩ ≡

```
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 413.

421. 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* 421 ⟩ ≡

```
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 413.

422. ⟨ Fetch the *prev_graf* 422 ⟩ ≡

```
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 413.

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

```
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 413.

424. 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 424 ⟩ ≡

```
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 413.

425. ⟨ Fetch a font dimension 425 ⟩ ≡

```
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 413.

426. ⟨ Fetch a font integer 426 ⟩ ≡

```
begin scan_font_ident;
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 413.

427. ⟨ Fetch a register 427 ⟩ ≡
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 413.

428. ⟨ Complain that `\the` can't do this; give zero result 428 ⟩ ≡
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 413.

429. 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 429 ⟩ ≡
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 413.

430. 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* 430 ⟩ ≡
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* 431 ⟩;
    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 413.

431. ⟨ Negate all three glue components of *cur_val* 431 ⟩ ≡
begin *negate*(*width*(*cur_val*)); *negate*(*stretch*(*cur_val*)); *negate*(*shrink*(*cur_val*));
end

This code is used in section 430.

432. 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*.

433. ⟨ Declare procedures that scan restricted classes of integers 433 ⟩ ≡

```
procedure scan_eight_bit_int;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > 255) then
    begin print_err("Bad␣register␣code");
    help2("A␣register␣number␣must␣be␣between␣0␣and␣255.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

See also sections 434, 435, 436, and 437.

This code is used in section 409.

434. ⟨ Declare procedures that scan restricted classes of integers 433 ⟩ +≡

```
procedure scan_char_num;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > 255) then
    begin print_err("Bad␣character␣code");
    help2("A␣character␣number␣must␣be␣between␣0␣and␣255.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

435. 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 433 ⟩ +≡

```
procedure scan_four_bit_int;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > 15) then
    begin print_err("Bad␣number");
    help2("Since␣I␣expected␣to␣read␣a␣number␣between␣0␣and␣15,")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

436. ⟨ Declare procedures that scan restricted classes of integers 433 ⟩ +≡

```
procedure scan_fifteen_bit_int;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > '77777) then
    begin print_err("Bad␣mathchar"); help2("A␣mathchar␣number␣must␣be␣between␣0␣and␣32767.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

437. ⟨ Declare procedures that scan restricted classes of integers 433 ⟩ +≡

```
procedure scan_twenty_seven_bit_int;
  begin scan_int;
  if (cur_val < 0) ∨ (cur_val > '777777777) then
    begin print_err("Bad␣delimiter␣code");
    help2("A␣numeric␣delimiter␣code␣must␣be␣between␣0␣and␣2^{27}-1.")
    ("I␣changed␣this␣one␣to␣zero."); int_error(cur_val); cur_val ← 0;
    end;
  end;
```

438. 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 }

439. 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*;

440. 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 441 ⟩;
  if *cur_tok* = *alpha_token* **then** ⟨ Scan an alphabetic character code into *cur_val* 442 ⟩
  else if (*cur_cmd* ≥ *min_internal*) ∧ (*cur_cmd* ≤ *max_internal*) **then**
      scan_something_internal(*int_val*, *false*)
    else ⟨ Scan a numeric constant 444 ⟩;
  if *negative* **then** *negate*(*cur_val*);
  end;

441. ⟨ Get the next non-blank non-sign token; set *negative* appropriately 441 ⟩ ≡
  negative ← *false*;
  repeat ⟨ Get the next non-blank non-call token 406 ⟩;
    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 440, 448, and 461.

442. 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* 442⟩ ≡
```
begin get_token; { suppress macro expansion }
if cur_tok < cs_token_flag then
  begin cur_val ← cur_chr;
  if cur_cmd ≤ right_brace then
    if cur_cmd = right_brace then incr(align_state)
    else decr(align_state);
  end
else if cur_tok < cs_token_flag + single_base then cur_val ← cur_tok − cs_token_flag − active_base
  else cur_val ← cur_tok − cs_token_flag − single_base;
if cur_val > 255 then
  begin print_err("Improper␣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 443⟩;
end
```
This code is used in section 440.

443. ⟨Scan an optional space 443⟩ ≡
```
begin get_x_token;
if cur_cmd ≠ spacer then back_input;
end
```
This code is used in sections 442, 448, 455, and 1200.

444. ⟨Scan a numeric constant 444⟩ ≡
```
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 445⟩;
if vacuous then ⟨Express astonishment that no number was here 446⟩
else if cur_cmd ≠ spacer then back_input;
end
```
This code is used in section 440.

445. **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 445 ⟩ ≡

```
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 444.

446. ⟨ Express astonishment that no number was here 446 ⟩ ≡

```
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 444.

447. 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* }

448. 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 450 ⟩
  begin *f* ← 0; *arith_error* ← *false*; *cur_order* ← *normal*; *negative* ← *false*;
  if ¬*shortcut* **then**
    begin ⟨ Get the next non-blank non-sign token; set *negative* appropriately 441 ⟩;
    if (*cur_cmd* ≥ *min_internal*) ∧ (*cur_cmd* ≤ *max_internal*) **then**
      ⟨ Fetch an internal dimension and **goto** *attach_sign*, or fetch an internal integer 449 ⟩
    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 452 ⟩;
      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 453 ⟩;
  ⟨ Scan an optional space 443 ⟩;
attach_sign: **if** *arith_error* ∨ (*abs*(*cur_val*) ≥ ´*10000000000*) **then**
    ⟨ Report that this dimension is out of range 460 ⟩;
  if *negative* **then** *negate*(*cur_val*);
  end;

449. ⟨ Fetch an internal dimension and **goto** *attach_sign*, or fetch an internal integer 449 ⟩ ≡
  if *mu* **then**
    begin *scan_something_internal*(*mu_val*, *false*); ⟨ Coerce glue to a dimension 451 ⟩;
    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 448.

450. ⟨ Local variables for dimension calculations 450 ⟩ ≡
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 448.

451. 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 451 ⟩ ≡
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 449 and 455.

452. 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 452 ⟩ ≡
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) ∨ (*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 448.

453. 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 453⟩ ≡
  if *inf* **then** ⟨Scan for `fil` units; **goto** *attach_fraction* if found 454⟩;
  ⟨Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 455⟩;
  if *mu* **then** ⟨Scan for `mu` units and **goto** *attach_fraction* 456⟩;
  if *scan_keyword*(`"true"`) **then** ⟨Adjust for the magnification ratio 457⟩;
  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 458⟩;
attach_fraction: **if** $cur\_val \ge {'40000}$ **then** *arith_error* ← *true*
  else *cur_val* ← *cur_val* ∗ *unity* + f;
done:

This code is used in section 448.

454. 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 454⟩ ≡
  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 453.

455. ⟨ Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 455 ⟩ ≡

```
save_cur_val ← cur_val; ⟨ Get the next non-blank non-call token 406 ⟩;
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 451 ⟩;
    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 558 ⟩)
else if scan_keyword("ex") then v ← (⟨ The x-height for cur_font 559 ⟩)
  else goto not_found;
⟨ Scan an optional space 443 ⟩;
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 453.

456. ⟨ Scan for mu units and **goto** *attach_fraction* 456 ⟩ ≡

```
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 453.

457. ⟨ Adjust for the magnification ratio 457 ⟩ ≡

```
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 453.

458. 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 458⟩ ≡
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* 459⟩;
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 453.

459. ⟨Complain about unknown unit and **goto** *done2* 459⟩ ≡
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 458.

460. ⟨Report that this dimension is out of range 460⟩ ≡
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 448.

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

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

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

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

462. ⟨Create a new glue specification whose width is *cur_val*; scan for its stretch and shrink components 462⟩ ≡

```
  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 461.

463. Here's a similar procedure that returns a pointer to a rule node. This routine is called just after TEX has seen `\hrule` or `\vrule`; therefore *cur_cmd* will be either *hrule* or *vrule*. The idea is to store the default rule dimensions in the node, then to override them if '`height`' or '`width`' or '`depth`' specifications are found (in any order).

```
  define default_rule = 26214   { 0.4 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;
```

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

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

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

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

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

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

```
function the_toks: pointer;
  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; scan_something_internal(tok_val, false);
  if cur_val_level ≥ ident_val then ⟨Copy the token list 466⟩
  else begin old_setting ← selector; selector ← new_string; b ← pool_ptr;
    case cur_val_level of
    int_val: print_int(cur_val);
    dimen_val: begin print_scaled(cur_val); print("pt");
      end;
    glue_val: begin print_spec(cur_val, "pt"); delete_glue_ref(cur_val);
      end;
    mu_val: begin print_spec(cur_val, "mu"); delete_glue_ref(cur_val);
      end;
    end; { there are no other cases }
    selector ← old_setting; the_toks ← str_toks(b);
    end;
  end;
```

466. ⟨ Copy the token list 466 ⟩ ≡

begin $p \leftarrow temp\_head$; $link(p) \leftarrow null$;
if $cur\_val\_level = ident\_val$ **then** $store\_new\_token(cs\_token\_flag + cur\_val)$
else if $cur\_val \neq null$ **then**
  begin $r \leftarrow link(cur\_val)$; { do not copy the reference count }
  while $r \neq null$ **do**
    begin $fast\_store\_new\_token(info(r))$; $r \leftarrow link(r)$;
    end;
  end;
the_toks $\leftarrow p$;
end

This code is used in section 465.

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

procedure *ins_the_toks*;
begin $link(garbage) \leftarrow the\_toks$; $ins\_list(link(temp\_head))$;
end;

468. 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 $job\_name\_code = 5$ { command code for `\jobname` }

⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
primitive("number", *convert*, *number_code*);
primitive("romannumeral", *convert*, *roman_numeral_code*);
primitive("string", *convert*, *string_code*);
primitive("meaning", *convert*, *meaning_code*);
primitive("fontname", *convert*, *font_name_code*);
primitive("jobname", *convert*, *job_name_code*);

469. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
convert: **case** *chr_code* **of**
number_code: *print_esc*("number");
roman_numeral_code: *print_esc*("romannumeral");
string_code: *print_esc*("string");
meaning_code: *print_esc*("meaning");
font_name_code: *print_esc*("fontname");
othercases *print_esc*("jobname")
endcases;

470. 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 }
  begin $c \leftarrow$ *cur_chr*; ⟨Scan the argument for command *c* 471⟩;
  old_setting ← *selector*; *selector* ← *new_string*; *b* ← *pool_ptr*; ⟨Print the result of command *c* 472⟩;
  selector ← *old_setting*; *link*(*garbage*) ← *str_toks*(*b*); *ins_list*(*link*(*temp_head*));
  end;

471. ⟨Scan the argument for command *c* 471⟩ ≡
  case *c* **of**
  number_code, *roman_numeral_code*: *scan_int*;
  string_code, *meaning_code*: **begin** *save_scanner_status* ← *scanner_status*; *scanner_status* ← *normal*;
    get_token; *scanner_status* ← *save_scanner_status*;
    end;
  font_name_code: *scan_font_ident*;
  job_name_code: **if** *job_name* = 0 **then** *open_log_file*;
  end { there are no other cases }

This code is used in section 470.

472. ⟨Print the result of command *c* 472⟩ ≡
  case *c* **of**
  number_code: *print_int*(*cur_val*);
  roman_numeral_code: *print_roman_int*(*cur_val*);
  string_code: **if** *cur_cs* ≠ 0 **then** *sprint_cs*(*cur_cs*)
    else *print_char*(*cur_chr*);
  meaning_code: *print_meaning*;
  font_name_code: **begin** *print*(*font_name*[*cur_val*]);
    if *font_size*[*cur_val*] ≠ *font_dsize*[*cur_val*] **then**
      begin *print*("␣at␣"); *print_scaled*(*font_size*[*cur_val*]); *print*("pt");
      end;
    end;
  job_name_code: *print*(*job_name*);
  end { there are no other cases }

This code is used in section 470.

473. Now we can't postpone the difficulties any longer; we must bravely tackle *scan_toks*. This function returns a pointer to the tail of a new token list, and it also makes *def_ref* point to the reference count at the head of that list.

There are two boolean parameters, *macro_def* and *xpand*. If *macro_def* is true, the goal is to create the token list for a macro definition; otherwise the goal is to create the token list for some other TEX primitive: `\mark`, `\output`, `\everypar`, `\lowercase`, `\uppercase`, `\message`, `\errmessage`, `\write`, or `\special`. In the latter cases a left brace must be scanned next; this left brace will not be part of the token list, nor will the matching right brace that comes at the end. If *xpand* is false, the token list will simply be copied from the input using *get_token*. Otherwise all expandable tokens will be expanded until unexpandable tokens are left, except that the results of expanding '`\the`' are not expanded further. If both *macro_def* and *xpand* are true, the expansion applies only to the macro body (i.e., to the material following the first *left_brace* character).

The value of *cur_cs* when *scan_toks* begins should be the *eqtb* address of the control sequence to display in "runaway" error messages.

function *scan_toks*(*macro_def*, *xpand* : *boolean*): *pointer*;
  label *found*, *continue*, *done*, *done1*, *done2*;
  var *t*: *halfword*; { token representing the highest parameter number }
    s: *halfword*; { saved token }
    p: *pointer*; { tail of the token list being built }
    q: *pointer*; { new node being added to the token list via *store_new_token* }
    unbalance: *halfword*; { number of unmatched left braces }
    hash_brace: *halfword*; { possible '`#{`' token }
  begin if *macro_def* **then** *scanner_status* ← *defining* **else** *scanner_status* ← *absorbing*;
  warning_index ← *cur_cs*; *def_ref* ← *get_avail*; *token_ref_count*(*def_ref*) ← *null*; *p* ← *def_ref*;
  hash_brace ← 0; *t* ← *zero_token*;
  if *macro_def* **then** ⟨Scan and build the parameter part of the macro definition 474⟩
  else *scan_left_brace*; { remove the compulsory left brace }
  ⟨Scan and build the body of the token list; **goto** *found* when finished 477⟩;
found: *scanner_status* ← *normal*;
  if *hash_brace* ≠ 0 **then** *store_new_token*(*hash_brace*);
  scan_toks ← *p*;
  end;

474. ⟨Scan and build the parameter part of the macro definition 474⟩ ≡
  begin loop
    begin *continue*: *get_token*; { set *cur_cmd*, *cur_chr*, *cur_tok* }
    if *cur_tok* < *right_brace_limit* **then goto** *done1*;
    if *cur_cmd* = *mac_param* **then** ⟨If the next character is a parameter number, make *cur_tok* a *match* token; but if it is a left brace, store '*left_brace*, *end_match*', set *hash_brace*, and **goto** *done* 476⟩;
    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* 475⟩;
done: **end**

This code is used in section 473.

475. ⟨Express shock at the missing left brace; **goto** *found* 475⟩ ≡
  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 474.

476. ⟨ 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* 476 ⟩ ≡

```
begin s ← match_token + cur_chr; get_token;
if cur_tok < left_brace_limit then
  begin hash_brace ← cur_tok; store_new_token(cur_tok); store_new_token(end_match_token);
  goto done;
  end;
if t = zero_token + 9 then
  begin print_err("You␣already␣have␣nine␣parameters");
  help2("I´m␣going␣to␣ignore␣the␣#␣sign␣you␣just␣used,")
  ("as␣well␣as␣the␣token␣that␣followed␣it."); error; goto continue;
  end
else begin incr(t);
  if cur_tok ≠ t then
    begin print_err("Parameters␣must␣be␣numbered␣consecutively");
    help2("I´ve␣inserted␣the␣digit␣you␣should␣have␣used␣after␣the␣#.")
    ("Type␣`1´␣to␣delete␣what␣you␣did␣use."); back_error;
    end;
  cur_tok ← s;
  end;
end
```

This code is used in section 474.

477. ⟨ Scan and build the body of the token list; **goto** *found* when finished 477 ⟩ ≡

```
unbalance ← 1;
loop begin if xpand then ⟨ Expand the next part of the input 478 ⟩
  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 ## 479 ⟩;
  store_new_token(cur_tok);
  end
```

This code is used in section 473.

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

⟨Expand the next part of the input 478⟩ ≡
  begin loop
    begin *get_next*;
    if *cur_cmd* $\le$ *max_command* **then goto** *done2*;
    if *cur_cmd* $\ne$ *the* **then** *expand*
    else begin $q \leftarrow$ *the_toks*;
      if *link*(*temp_head*) $\ne$ *null* **then**
        begin *link*(*p*) $\leftarrow$ *link*(*temp_head*); $p \leftarrow q$;
        end;
      end;
    end;
done2: *x_token*
  end

This code is used in section 477.

479. ⟨Look for parameter number or **##** 479⟩ ≡
  begin $s \leftarrow$ *cur_tok*;
  if *xpand* **then** *get_x_token*
  else *get_token*;
  if *cur_cmd* $\ne$ *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* $-$ "0" $+$ *cur_chr*;
  end

This code is used in section 477.

480. 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*] }

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

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

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

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

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

This code is used in section 482.

484. Here we input on-line into the *buffer* array, prompting the user explicitly if $n \ge 0$. 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 484⟩ ≡
  if interaction > nonstop_mode then
    if n < 0 then prompt_input("")
    else begin wake_up_terminal; print_ln; sprint_cs(r); prompt_input("="); n ← −1;
      end
  else fatal_error("*** (cannot \read from terminal in nonstop modes)")
```

This code is used in section 483.

485. 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*] 485 ⟩ ≡
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 483.

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

⟨ Input the next line of *read_file*[*m*] 486 ⟩ ≡
begin if ¬*input_ln*(*read_file*[*m*], *true*) **then**
begin *a_close*(*read_file*[*m*]); *read_open*[*m*] ← *closed*;
if *align_state* ≠ 1000000 **then**
begin *runaway*; *print_err*("File␣ended␣within␣"); *print_esc*("read");
help1("This␣\read␣has␣unbalanced␣braces."); *align_state* ← 1000000; *limit* ← 0; *error*;
end;
end;
end

This code is used in section 483.

487. 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*);

488. ⟨ 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;

489. 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 }

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

491. ⟨ 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*);

492. ⟨ 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");

493. 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 }

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

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

495. 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 495⟩ ≡
  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 498.

496. ⟨Pop the condition stack 496⟩ ≡

```
  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 498, 500, 509, and 510.

497. 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;
```

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

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

499. 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.

500. ⟨Skip to `\else` or `\fi`, then **goto** *common_ending* 500⟩ ≡

```
  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 496⟩;
    end
```

This code is used in section 498.

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

```
case this_if of
if_char_code, if_cat_code: ⟨ Test if two characters match 506 ⟩;
if_int_code, if_dim_code: ⟨ Test relation between integers or dimensions 503 ⟩;
if_odd_code: ⟨ Test if an integer is odd 504 ⟩;
if_vmode_code: b ← (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 505 ⟩;
ifx_code: ⟨ Test if two tokens match 507 ⟩;
if_eof_code: begin scan_four_bit_int; b ← (read_open[cur_val] = closed);
  end;
if_true_code: b ← true;
if_false_code: b ← false;
if_case_code: ⟨ Select the appropriate case and return or goto common_ending 509 ⟩;
end { there are no other cases }
```

This code is used in section 498.

502. ⟨ Display the value of *b* 502 ⟩ ≡

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

This code is used in section 498.

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

⟨ Test relation between integers or dimensions 503 ⟩ ≡

```
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 406 ⟩;
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 501.

504. ⟨ Test if an integer is odd 504 ⟩ ≡

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

This code is used in section 501.

505. ⟨Test box register status 505⟩ ≡

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

This code is used in section 501.

506. 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 506⟩ ≡

```
begin get_x_token_or_active_char;
if (cur_cmd > active_char) ∨ (cur_chr > 255) then  { not a character }
  begin m ← relax; n ← 256;
  end
else begin m ← cur_cmd; n ← cur_chr;
  end;
get_x_token_or_active_char;
if (cur_cmd > active_char) ∨ (cur_chr > 255) then
  begin cur_cmd ← relax; cur_chr ← 256;
  end;
if this_if = if_char_code then b ← (n = cur_chr) else b ← (m = cur_cmd);
end
```

This code is used in section 501.

507. 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 507⟩ ≡

```
begin save_scanner_status ← scanner_status; scanner_status ← normal; get_next; n ← cur_cs;
p ← cur_cmd; q ← cur_chr; get_next;
if cur_cmd ≠ p then b ← false
else if cur_cmd < call then b ← (cur_chr = q)
  else ⟨Test if two macro texts match 508⟩;
scanner_status ← save_scanner_status;
end
```

This code is used in section 501.

508. 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 508⟩ ≡
```
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 507.

509. ⟨Select the appropriate case and **return** or **goto** *common_ending* 509⟩ ≡
```
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 496⟩;
  end;
change_if_limit(or_code, save_cond_ptr); return; { wait for \or, \else, or \fi }
end
```

This code is used in section 501.

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

⟨Terminate the current conditional and skip to \fi 510⟩ ≡
```
if cur_chr > if_limit then
  if if_limit = if_code then insert_relax { condition not yet evaluated }
  else begin print_err("Extra␣"); print_cmd_chr(fi_or_else, cur_chr);
    help1("I´m␣ignoring␣this;␣it␣doesn´t␣match␣any␣\if."); error;
    end
else begin while cur_chr ≠ fi_code do pass_text; { skip to \fi }
  ⟨Pop the condition stack 496⟩;
  end
```

This code is used in section 367.

511. 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.

512. 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

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

These three procedures communicate with each other via global variables. Afterwards the file name will appear in the string pool as three strings called *cur_name*, *cur_area*, and *cur_ext*; the latter two are null (i.e., `""`), unless they were explicitly specified by the user.

Actually the situation is slightly more complicated, because TEX needs to know when the file name ends. The *more_name* routine is a function (with side effects) that returns *true* on the calls $more\_name(c_1)$, ..., $more\_name(c_{n-1})$. The final call $more\_name(c_n)$ returns *false*; or, it returns *true* and the token following c_n is something like '`\hbox`' (i.e., not a character). In other words, *more_name* is supposed to return *true* unless it is sure that the file name has been completely scanned; and *end_name* is supposed to be able to finish the assembly of *cur_name*, *cur_area*, and *cur_ext* regardless of whether $more\_name(c_n)$ returned *true* or *false*.

⟨ Global variables 13 ⟩ +≡
cur_name: *str_number*; { name of file just scanned }
cur_area: *str_number*; { file area just scanned, or `""` }
cur_ext: *str_number*; { file extension just scanned, or `""` }

513. The file names we shall deal with for illustrative purposes have the following structure: If the name contains '>' or ':', the file area consists of all characters up to and including the final such character; otherwise the file area is null. If the remaining file name contains '.', the file extension consists of all such characters from the first remaining '.' to the end, otherwise the file extension is null.

We can scan such file names easily by using two global variables that keep track of the occurrences of area and extension delimiters:

```
⟨ Global variables 13 ⟩ +≡
area_delimiter: pool_pointer;  { the most recent '>' or ':', if any }
ext_delimiter: pool_pointer;  { the relevant '.', if any }
```

514. Input files that can't be found in the user's area may appear in a standard system area called *TEX_area*. Font metric files whose areas are not given explicitly are assumed to appear in a standard system area called *TEX_font_area*. These system area names will, of course, vary from place to place.

```
define TEX_area ≡ "TeXinputs:"
define TEX_font_area ≡ "TeXfonts:"
```

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

```
procedure begin_name;
  begin area_delimiter ← 0; ext_delimiter ← 0;
  end;
```

516. And here's the second. The string pool might change as the file name is being scanned, since a new `\csname` might be entered; therefore we keep *area_delimiter* and *ext_delimiter* relative to the beginning of the current string, instead of assigning an absolute address like *pool_ptr* to them.

```
function more_name(c : ASCII_code): boolean;
  begin if c = "␣" then more_name ← false
  else begin str_room(1); append_char(c);  { contribute c to the current string }
    if (c = ">") ∨ (c = ":") then
      begin area_delimiter ← cur_length; ext_delimiter ← 0;
      end
    else if (c = ".") ∧ (ext_delimiter = 0) then ext_delimiter ← cur_length;
    more_name ← true;
    end;
  end;
```

517. The third.

```
procedure end_name;
  begin if str_ptr + 3 > max_strings then overflow("number␣of␣strings", max_strings − init_str_ptr);
  if area_delimiter = 0 then cur_area ← ""
  else begin cur_area ← str_ptr; str_start[str_ptr + 1] ← str_start[str_ptr] + area_delimiter; incr(str_ptr);
    end;
  if ext_delimiter = 0 then
    begin cur_ext ← ""; cur_name ← make_string;
    end
  else begin cur_name ← str_ptr;
    str_start[str_ptr + 1] ← str_start[str_ptr] + ext_delimiter − area_delimiter − 1; incr(str_ptr);
    cur_ext ← make_string;
    end;
  end;
```

518. Conversely, here is a routine that takes three strings and prints a file name that might have produced them. (The routine is system dependent, because some operating systems put the file area last instead of first.)

⟨ Basic printing procedures 57 ⟩ +≡
procedure *print_file_name*(n, a, e : *integer*);
  begin *slow_print*(a); *slow_print*(n); *slow_print*(e);
  end;

519. 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 $c \leftarrow$ #; *incr*(k);
      if $k \le$ *file_name_size* **then** *name_of_file*[k] ← *xchr*[c];
      end

procedure *pack_file_name*(n, a, e : *str_number*);
  var k: *integer*; { number of positions filled in *name_of_file* }
   c: *ASCII_code*; { character being packed }
   j: *pool_pointer*; { index into *str_pool* }
  begin $k \leftarrow 0$;
  for $j \leftarrow$ *str_start*[a] **to** *str_start*[$a + 1$] − 1 **do** *append_to_name*(*so*(*str_pool*[j]));
  for $j \leftarrow$ *str_start*[n] **to** *str_start*[$n + 1$] − 1 **do** *append_to_name*(*so*(*str_pool*[j]));
  for $j \leftarrow$ *str_start*[e] **to** *str_start*[$e + 1$] − 1 **do** *append_to_name*(*so*(*str_pool*[j]));
  if $k \le$ *file_name_size* **then** *name_length* ← k **else** *name_length* ← *file_name_size*;
  for $k \leftarrow$ *name_length* + 1 **to** *file_name_size* **do** *name_of_file*[k] ← ´␣´;
  end;

520. A messier routine is also needed, since format file names must be scanned before TEX's string mechanism has been initialized. We shall use the global variable *TEX_format_default* to supply the text for default system areas and extensions related to format files.

  define *format_default_length* = 20 { length of the *TEX_format_default* string }
  define *format_area_length* = 11 { length of its area part }
  define *format_ext_length* = 4 { length of its '`.fmt`' part }
  define *format_extension* = `".fmt"` { the extension, as a `WEB` constant }

⟨ Global variables 13 ⟩ +≡
TEX_format_default: **packed array** [1 .. *format_default_length*] **of** *char*;

521. ⟨ Set initial values of key variables 21 ⟩ +≡
  TEX_format_default ← ´`TeXformats:plain.fmt`´;

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

523. Here is the messy routine that was just mentioned. It sets *name_of_file* from the first n characters of *TEX_format_default*, followed by *buffer*$[a\ ..\ b]$, followed by the last *format_ext_length* characters of *TEX_format_default*.

We dare not give error messages here, since TEX calls this routine before the *error* routine is ready to roll. Instead, we simply drop excess characters, since the error will be detected in another way when a strange file name isn't found.

```
procedure pack_buffered_name(n : small_number; a, b : integer);
  var k: integer; { number of positions filled in name_of_file }
    c: ASCII_code; { character being packed }
    j: integer; { index into buffer or TEX_format_default }
  begin if n + b − a + 1 + format_ext_length > file_name_size then
    b ← a + file_name_size − n − 1 − format_ext_length;
  k ← 0;
  for j ← 1 to n do append_to_name(xord[TEX_format_default[j]]);
  for j ← a to b do append_to_name(buffer[j]);
  for j ← format_default_length − format_ext_length + 1 to format_default_length do
    append_to_name(xord[TEX_format_default[j]]);
  if k ≤ file_name_size then name_length ← k else name_length ← file_name_size;
  for k ← name_length + 1 to file_name_size do name_of_file[k] ← ´␣´;
  end;
```

524. 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*$[loc] \neq$ "␣".

```
⟨ Declare the function called open_fmt_file 524 ⟩ ≡
function open_fmt_file: boolean;
  label found, exit;
  var j: 0 .. buf_size; { the first space after the format file name }
  begin j ← loc;
  if buffer[loc] = "&" then
    begin incr(loc); j ← loc; buffer[last] ← "␣";
    while buffer[j] ≠ "␣" do incr(j);
    pack_buffered_name(0, loc, j − 1); { try first without the system file area }
    if w_open_in(fmt_file) then goto found;
    pack_buffered_name(format_area_length, loc, j − 1); { now try the system format file area }
    if w_open_in(fmt_file) then goto found;
    wake_up_terminal; wterm_ln(´Sorry,␣I␣can´´t␣find␣that␣format;´, ´␣will␣try␣PLAIN.´);
    update_terminal;
    end; { now pull out all the stops: try for the system plain file }
  pack_buffered_name(format_default_length − format_ext_length, 1, 0);
  if ¬w_open_in(fmt_file) then
    begin wake_up_terminal; wterm_ln(´I␣can´´t␣find␣the␣PLAIN␣format␣file!´);
    open_fmt_file ← false; return;
    end;
found: loc ← j; open_fmt_file ← true;
exit: end;
```

This code is used in section 1303.

525. Operating systems often make it possible to determine the exact name (and possible version number) of a file that has been opened. The following routine, which simply makes a TEX string from the value of *name_of_file*, should ideally be changed to deduce the full name of file f, which is the file most recently opened, if it is possible to do this in a Pascal program.

This routine might be called after string memory has overflowed, hence we dare not use '*str_room*'.

function *make_name_string*: *str_number*;
  var k: 1 .. *file_name_size*; { index into *name_of_file* }
  begin if (*pool_ptr* + *name_length* > *pool_size*) ∨ (*str_ptr* = *max_strings*) ∨ (*cur_length* > 0) **then**
    make_name_string ← "?"
  else begin for k ← 1 **to** *name_length* **do** *append_char*(*xord*[*name_of_file*[k]]);
    make_name_string ← *make_string*;
    end;
  end;
function *a_make_name_string*(**var** f : *alpha_file*): *str_number*;
  begin *a_make_name_string* ← *make_name_string*;
  end;
function *b_make_name_string*(**var** f : *byte_file*): *str_number*;
  begin *b_make_name_string* ← *make_name_string*;
  end;
function *w_make_name_string*(**var** f : *word_file*): *str_number*;
  begin *w_make_name_string* ← *make_name_string*;
  end;

526. 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 406 ⟩;
  loop begin if (*cur_cmd* > *other_char*) ∨ (*cur_chr* > 255) **then** { not a character }
      begin *back_input*; **goto** *done*;
      end;
    if ¬*more_name*(*cur_chr*) **then goto** *done*;
    get_x_token;
    end;
done: *end_name*; *name_in_progress* ← *false*;
  end;

527. 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? }

528. 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*;

529. Here is a routine that manufactures the output file names, assuming that *job_name* ≠ 0. It ignores and changes the current settings of *cur_area* and *cur_ext*.

define *pack_cur_name* ≡ *pack_file_name*(*cur_name*, *cur_area*, *cur_ext*)

```
procedure pack_job_name(s : str_number);  { s = ".log", ".dvi", or format_extension }
  begin cur_area ← ""; cur_ext ← s; cur_name ← job_name; pack_cur_name;
  end;
```

530. If some trouble arises when TeX tries to open a file, the following routine calls upon the user to supply another file name. Parameter *s* is used in the error message to identify the type of file; parameter *e* is the default extension if none is given. Upon exit from the routine, variables *cur_name*, *cur_area*, *cur_ext*, and *name_of_file* are ready for another attempt at file opening.

```
procedure prompt_file_name(s, e : str_number);
  label done;
  var k: 0 .. buf_size;  { index into buffer }
  begin if interaction = scroll_mode then wake_up_terminal;
  if s = "input␣file␣name" then print_err("I␣can´t␣find␣file␣`")
  else print_err("I␣can´t␣write␣on␣file␣`");
  print_file_name(cur_name, cur_area, cur_ext); print("´.");
  if e = ".tex" then show_context;
  print_nl("Please␣type␣another␣"); print(s);
  if interaction < scroll_mode then fatal_error("***␣(job␣aborted,␣file␣error␣in␣nonstop␣mode)");
  clear_terminal; prompt_input(":␣"); ⟨Scan file name in the buffer 531⟩;
  if cur_ext = "" then cur_ext ← e;
  pack_cur_name;
  end;
```

531. ⟨Scan file name in the buffer 531⟩ ≡

```
  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 530.

532. Here's an example of how these conventions are used. Whenever it is time to ship out a box of stuff, we shall use the macro *ensure_dvi_open*.

```
define ensure_dvi_open ≡
        if output_file_name = 0 then
          begin if job_name = 0 then open_log_file;
          pack_job_name(".dvi");
          while ¬b_open_out(dvi_file) do prompt_file_name("file␣name␣for␣output", ".dvi");
          output_file_name ← b_make_name_string(dvi_file);
          end
```

⟨ Global variables 13 ⟩ +≡
dvi_file: *byte_file*; { the device-independent output goes here }
output_file_name: *str_number*; { full name of the output file }
log_name: *str_number*; { full name of the log file }

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

534. The *open_log_file* routine is used to open the transcript file and to help it catch up to what has previously been printed on the terminal.

```
procedure open_log_file;
  var old_setting: 0 .. max_selector; { previous selector setting }
    k: 0 .. buf_size; { index into months and buffer }
    l: 0 .. buf_size; { end of first input line }
    months: packed array [1 .. 36] of char; { abbreviations of month names }
  begin old_setting ← selector;
  if job_name = 0 then job_name ← "texput";
  pack_job_name(".log");
  while ¬a_open_out(log_file) do ⟨ Try to get a different log file name 535 ⟩;
  log_name ← a_make_name_string(log_file); selector ← log_only; log_opened ← true;
  ⟨ Print the banner line, including the date and time 536 ⟩;
  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;
```

535. 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 535 ⟩ ≡

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

This code is used in section 534.

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

```
begin wlog(banner); slow_print(format_ident); print("␣␣"); print_int(sys_day); print_char("␣");
months ← ´JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC´;
for k ← 3 * sys_month − 2 to 3 * sys_month do wlog(months[k]);
print_char("␣"); print_int(sys_year); print_char("␣"); print_two(sys_time div 60); print_char(":");
print_two(sys_time mod 60);
end
```

This code is used in section 534.

537. Let's turn now to the procedure that is used to initiate file reading when an '\input' command is being processed. Beware: For historic reasons, this code foolishly conserves a tiny bit of string pool space; but that can confuse the interactive 'E' option.

```
procedure start_input; { TEX will \input something }
  label done;
  begin scan_file_name; { set cur_name to desired file name }
  if cur_ext = "" then cur_ext ← ".tex";
  pack_cur_name;
  loop begin begin_file_reading; { set up cur_file and new level of input }
    if a_open_in(cur_file) then goto done;
    if cur_area = "" then
      begin pack_file_name(cur_name, TEX_area, cur_ext);
      if a_open_in(cur_file) then goto done;
      end;
    end_file_reading; { remove the level that didn't work }
    prompt_file_name("input␣file␣name", ".tex");
    end;
done: name ← a_make_name_string(cur_file);
  if job_name = 0 then
    begin job_name ← cur_name; open_log_file;
    end; { open_log_file doesn't show_context, so limit and loc needn't be set to meaningful values yet }
  if term_offset + length(name) > max_print_line − 2 then print_ln
  else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
  print_char("("); incr(open_parens); slow_print(name); update_terminal; state ← new_line;
  if name = str_ptr − 1 then { conserve string pool space (but see note above) }
    begin flush_string; name ← cur_name;
    end;
  ⟨Read the first line of the new file 538⟩;
  end;
```

538. 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 538⟩ ≡

```
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 537.

539. 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*;

540. 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.

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

$$\begin{aligned} header &: \textbf{array}\ [0\mathrel{..}lh-1]\ \textbf{of}\ stuff\\ char\_info &: \textbf{array}\ [bc\mathrel{..}ec]\ \textbf{of}\ char\_info\_word\\ width &: \textbf{array}\ [0\mathrel{..}nw-1]\ \textbf{of}\ fix\_word\\ height &: \textbf{array}\ [0\mathrel{..}nh-1]\ \textbf{of}\ fix\_word\\ depth &: \textbf{array}\ [0\mathrel{..}nd-1]\ \textbf{of}\ fix\_word\\ italic &: \textbf{array}\ [0\mathrel{..}ni-1]\ \textbf{of}\ fix\_word\\ lig\_kern &: \textbf{array}\ [0\mathrel{..}nl-1]\ \textbf{of}\ lig\_kern\_command\\ kern &: \textbf{array}\ [0\mathrel{..}nk-1]\ \textbf{of}\ fix\_word\\ exten &: \textbf{array}\ [0\mathrel{..}ne-1]\ \textbf{of}\ extensible\_recipe\\ param &: \textbf{array}\ [1\mathrel{..}np]\ \textbf{of}\ fix\_word \end{aligned}$$

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 $+16$.

542. 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.

543. 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*.

544. 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 }

545. 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 boundary character of this font; the value of *next_char* need not lie between *bc* and *ec*. If the very last instruction of the *lig_kern* array has $skip\_byte = 255$, there is a special ligature/kerning program for a boundary character at the left, beginning at location $256 * op\_byte + remainder$. The interpretation is that TeX puts implicit boundary characters before and after each consecutive string of characters from the same font. These implicit characters do not appear in the output, but they can affect ligatures and kerning.

If the very first instruction of a character's *lig_kern* program has $skip\_byte > 128$, the program actually begins in location $256 * op\_byte + remainder$. This feature allows access to large *lig_kern* arrays, because the first instruction must otherwise appear in a location ≤ 255.

Any instruction with $skip\_byte > 128$ in the *lig_kern* array must satisfy the condition

$$256 * op\_byte + remainder < nl.$$

If such an instruction is encountered during normal program execution, it denotes an unconditional halt; no ligature or kerning command is performed.

define $stop\_flag \equiv qi(128)$ { 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 $rem\_byte(\#) \equiv \#.b3$

546. 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 }

547. 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

548. So that is what TFM files hold. Since TEX has to absorb such information about lots of fonts, it stores most of the data in a large array called *font_info*. Each item of *font_info* is a *memory_word*; the *fix_word* data gets converted into *scaled* entries, while everything else goes into words of type *four_quarters*.

When the user defines `\font\f`, say, TEX assigns an internal number to the user's font `\f`. Adding this number to *font_id_base* gives the *eqtb* location of a "frozen" control sequence that will always select the font.

⟨ Types in the outer block 18 ⟩ +≡
internal_font_number = *font_base* . . *font_max*; { *font* in a *char_node* }
font_index = 0 . . *font_mem_size*; { index into *font_info* }

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

define *non_char* ≡ *qi*(256) { a *halfword* code that can't match a real character }
define *non_address* = 0 { a spurious *bchar_label* }

⟨ Global variables 13 ⟩ +≡
font_info: **array** [*font_index*] **of** *memory_word*; { the big collection of font data }
fmem_ptr: *font_index*; { first unused word of *font_info* }
font_ptr: *internal_font_number*; { largest internal font number in use }
font_check: **array** [*internal_font_number*] **of** *four_quarters*; { check sum }
font_size: **array** [*internal_font_number*] **of** *scaled*; { "at" size }
font_dsize: **array** [*internal_font_number*] **of** *scaled*; { "design" size }
font_params: **array** [*internal_font_number*] **of** *font_index*; { how many font parameters are present }
font_name: **array** [*internal_font_number*] **of** *str_number*; { name of the font }
font_area: **array** [*internal_font_number*] **of** *str_number*; { area of the font }
font_bc: **array** [*internal_font_number*] **of** *eight_bits*; { beginning (smallest) character code }
font_ec: **array** [*internal_font_number*] **of** *eight_bits*; { ending (largest) character code }
font_glue: **array** [*internal_font_number*] **of** *pointer*;
{ glue specification for interword space, *null* if not allocated }
font_used: **array** [*internal_font_number*] **of** *boolean*;
{ has a character from this font actually appeared in the output? }
hyphen_char: **array** [*internal_font_number*] **of** *integer*; { current `\hyphenchar` values }
skew_char: **array** [*internal_font_number*] **of** *integer*; { current `\skewchar` values }
bchar_label: **array** [*internal_font_number*] **of** *font_index*;
{ start of *lig_kern* program for left boundary character, *non_address* if there is none }
font_bchar: **array** [*internal_font_number*] **of** *min_quarterword* .. *non_char*;
{ boundary character, *non_char* if there is none }
font_false_bchar: **array** [*internal_font_number*] **of** *min_quarterword* .. *non_char*;
{ *font_bchar* if it doesn't exist in the font, otherwise *non_char* }

550. Besides the arrays just enumerated, we have directory arrays that make it easy to get at the individual entries in *font_info*. For example, the *char_info* data for character c in font f will be in *font_info*[*char_base*[f] + c].*qqqq*; and if w is the *width_index* part of this word (the *b0* field), the width of the character is *font_info*[*width_base*[f] + w].*sc*. (These formulas assume that *min_quarterword* has already been added to c and to w, since TeX stores its quarterwords that way.)

⟨ Global variables 13 ⟩ +≡
char_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for *char_info* }
width_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for widths }
height_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for heights }
depth_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for depths }
italic_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for italic corrections }
lig_kern_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for ligature/kerning programs }
kern_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for kerns }
exten_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for extensible recipes }
param_base: **array** [*internal_font_number*] **of** *integer*; { base addresses for font parameters }

551. ⟨ Set initial values of key variables 21 ⟩ +≡
for k ← *font_base* **to** *font_max* **do** *font_used*[k] ← *false*;

552. 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 ⟩ +≡
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 *k* ← 0 **to** 6 **do** *font_info*[*k*].*sc* ← 0;

553. ⟨ 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*];

554. 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.

define *char_info_end*(#) ≡ #] .*qqqq*
define *char_info*(#) ≡ *font_info* [*char_base*[#] + *char_info_end*
define *char_width_end*(#) ≡ #.*b0*] .*sc*
define *char_width*(#) ≡ *font_info* [*width_base*[#] + *char_width_end*
define *char_exists*(#) ≡ (#.*b0* > *min_quarterword*)
define *char_italic_end*(#) ≡ (*qo*(#.*b2*)) **div** 4] .*sc*
define *char_italic*(#) ≡ *font_info* [*italic_base*[#] + *char_italic_end*
define *height_depth*(#) ≡ *qo*(#.*b1*)
define *char_height_end*(#) ≡ (#) **div** 16] .*sc*
define *char_height*(#) ≡ *font_info* [*height_base*[#] + *char_height_end*
define *char_depth_end*(#) ≡ (#) **mod** 16] .*sc*
define *char_depth*(#) ≡ *font_info* [*depth_base*[#] + *char_depth_end*
define *char_tag*(#) ≡ ((*qo*(#.*b2*)) **mod** 4)

555. 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 }

556. ⟨ 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*;

557. 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*$(i) \le$ *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*

558. 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* 558 ⟩ ≡
quad(*cur_font*)

This code is used in section 455.

559. ⟨ The x-height for *cur_font* 559 ⟩ ≡
x_height(*cur_font*)

This code is used in section 455.

560. 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 }
function read_font_info(u : pointer; nom, aire : str_number; s : scaled): internal_font_number;
      { input a TFM file }
  label done, bad_tfm, not_found;
  var k: font_index;  { index into font_info }
    file_opened: boolean;  { was tfm_file successfully opened? }
    lf, lh, bc, ec, nw, nh, nd, ni, nl, nk, ne, np: halfword;  { sizes of subfiles }
    f: internal_font_number;  { the new font's number }
    g: internal_font_number;  { the number to return }
    a, b, c, d: eight_bits;  { byte variables }
    qw: four_quarters; sw: scaled;  { accumulators }
    bch_label: integer;  { left boundary start location, or infinity }
    bchar: 0 .. 256;  { boundary character, or 256 }
    z: scaled;  { the design size or the "at" size }
    alpha: integer; beta: 1 .. 16;  { auxiliary quantities used in fixed-point multiplication }
  begin g ← null_font;
  ⟨Read and check the font data; abort if the TFM file is malformed; if there's no room for this font, say so
      and goto done; otherwise incr(font_ptr) and goto done 562⟩;
bad_tfm: ⟨Report that the font won't be loaded 561⟩;
done: if file_opened then b_close(tfm_file);
  read_font_info ← g;
  end;
```

561. 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 $s \geq 0$ **then**
      begin *print*("␣at␣"); *print_scaled*(*s*); *print*("pt");
      end
    else if $s \neq -1000$ **then**
        begin *print*("␣scaled␣"); *print_int*(−*s*);
        end

⟨Report that the font won't be loaded 561⟩ ≡
  start_font_error_message;
  if *file_opened* **then** *print*("␣not␣loadable:␣Bad␣metric␣(TFM)␣file")
  else *print*("␣not␣loadable:␣Metric␣(TFM)␣file␣not␣found");
  help5("I␣wasn´t␣able␣to␣read␣the␣size␣data␣for␣this␣font,")
  ("so␣I␣will␣ignore␣the␣font␣specification.")
  ("[Wizards␣can␣fix␣TFM␣files␣using␣TFtoPL/PLtoTF.]")
  ("You␣might␣try␣inserting␣a␣different␣font␣spec;")
  ("e.g.,␣type␣`I\font<same␣font␣id>=<substitute␣font␣name>´."); *error*

This code is used in section 560.

562. ⟨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* 562⟩ ≡
  ⟨Open *tfm_file* for input 563⟩;
  ⟨Read the TFM size fields 565⟩;
  ⟨Use size fields to allocate font information 566⟩;
  ⟨Read the TFM header 568⟩;
  ⟨Read character data 569⟩;
  ⟨Read box dimensions 571⟩;
  ⟨Read ligature/kern program 573⟩;
  ⟨Read extensible character recipes 574⟩;
  ⟨Read font parameters 575⟩;
  ⟨Make final adjustments and **goto** *done* 576⟩

This code is used in section 560.

563. ⟨Open *tfm_file* for input 563⟩ ≡
  file_opened ← *false*;
  if *aire* = "" **then** *pack_file_name*(*nom*, *TEX_font_area*, ".tfm")
  else *pack_file_name*(*nom*, *aire*, ".tfm");
  if ¬*b_open_in*(*tfm_file*) **then** *abort*;
  file_opened ← *true*

This code is used in section 562.

564. Note: A malformed TFM file might be shorter than it claims to be; thus *eof* (*tfm_file*) might be true when *read_font_info* refers to *tfm_file*↑ or when it says *get*(*tfm_file*). If such circumstances cause system error messages, you will have to defeat them somehow, for example by defining *fget* to be '**begin** *get*(*tfm_file*); **if** *eof* (*tfm_file*) **then** *abort*; **end**'.

```
define fget ≡ get(tfm_file)
define fbyte ≡ tfm_file↑
define read_sixteen(#) ≡
        begin # ← fbyte;
        if # > 127 then abort;
        fget; # ← # * ´400 + fbyte;
        end
define store_four_quarters(#) ≡
        begin fget; a ← fbyte; qw.b0 ← qi(a); fget; b ← fbyte; qw.b1 ← qi(b); fget; c ← fbyte;
        qw.b2 ← qi(c); fget; d ← fbyte; qw.b3 ← qi(d); # ← qw;
        end
```

565. ⟨ Read the TFM size fields 565 ⟩ ≡

```
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 562.

566. 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 566 ⟩ ≡

```
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 567 ⟩;
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 562.

567. ⟨ Apologize for not loading the font, **goto** *done* 567 ⟩ ≡

```
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 566.

568. Only the first two words of the header are needed by TEX82.

⟨Read the TFM header 568⟩ ≡
```
begin if lh < 2 then abort;
store_four_quarters(font_check[f]); fget; read_sixteen(z); {this rejects a negative design size}
fget; z ← z * ´400 + fbyte; fget; z ← (z * ´20) + (fbyte div ´20);
if z < unity then abort;
while lh > 2 do
  begin fget; fget; fget; fget; decr(lh); {ignore the rest of the header}
  end;
font_dsize[f] ← z;
if s ≠ −1000 then
  if s ≥ 0 then z ← s
  else z ← xn_over_d(z, −s, 1000);
font_size[f] ← z;
end
```
This code is used in section 562.

569. ⟨Read character data 569⟩ ≡
```
for k ← fmem_ptr to width_base[f] − 1 do
  begin store_four_quarters(font_info[k].qqqq);
  if (a ≥ nw) ∨ (b div ´20 ≥ nh) ∨ (b mod ´20 ≥ nd) ∨ (c div 4 ≥ ni) then abort;
  case c mod 4 of
  lig_tag: if d ≥ nl then abort;
  ext_tag: if d ≥ ne then abort;
  list_tag: ⟨Check for charlist cycle 570⟩;
  othercases do_nothing {no_tag}
  endcases;
  end
```
This code is used in section 562.

570. 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) ∨ (# > ec) then abort
    end
define current_character_being_worked_on ≡ k + bc − fmem_ptr
```

⟨Check for charlist cycle 570⟩ ≡
```
begin check_byte_range(d);
while d < current_character_being_worked_on do
  begin qw ← char_info(f)(d); {N.B.: not qi(d), since char_base[f] hasn't been adjusted yet}
  if char_tag(qw) ≠ list_tag then goto not_found;
  d ← qo(rem_byte(qw)); {next character on the list}
  end;
if d = current_character_being_worked_on then abort; {yes, there's a cycle}
not_found: end
```
This code is used in section 569.

571. 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 571 ⟩ ≡
begin ⟨ Replace *z* by *z*′ and compute α, β 572 ⟩;
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 562.

572. ⟨ Replace *z* by *z*′ and compute α, β 572 ⟩ ≡
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 section 571.

573. **define** *check_existence*(#) ≡
begin *check_byte_range*(#); *qw* ← *char_info*(*f*)(#); { N.B.: not *qi*(#) }
if ¬*char_exists*(*qw*) **then** *abort*;
end

⟨ Read ligature/kern program 573 ⟩ ≡

```
bch_label ← ´77777; bchar ← 256;
if nl > 0 then
  begin for k ← 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 ≥ nl then abort;
      if a = 255 then
        if k = lig_kern_base[f] then bchar ← b;
      end
    else begin if b ≠ bchar then check_existence(b);
      if c < 128 then check_existence(d)   { check ligature }
      else if 256 ∗ (c − 128) + d ≥ nk then abort;   { check kern }
      if a < 128 then
        if k − lig_kern_base[f] + a + 1 ≥ nl then abort;
      end;
    end;
  if a = 255 then bch_label ← 256 ∗ c + d;
  end;
for k ← kern_base[f] + kern_base_offset to exten_base[f] − 1 do store_scaled(font_info[k].sc);
```

This code is used in section 562.

574. ⟨ Read extensible character recipes 574 ⟩ ≡

```
for k ← exten_base[f] to param_base[f] − 1 do
  begin store_four_quarters(font_info[k].qqqq);
  if a ≠ 0 then check_existence(a);
  if b ≠ 0 then check_existence(b);
  if c ≠ 0 then check_existence(c);
  check_existence(d);
  end
```

This code is used in section 562.

575. 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 575 ⟩ ≡

```
begin for k ← 1 to np do
  if k = 1 then   { the slant parameter is a pure number }
    begin fget; sw ← fbyte;
    if sw > 127 then sw ← sw − 256;
    fget; sw ← sw ∗ ´400 + fbyte; fget; sw ← sw ∗ ´400 + fbyte; fget;
    font_info[param_base[f]].sc ← (sw ∗ ´20) + (fbyte div ´20);
    end
  else store_scaled(font_info[param_base[f] + k − 1].sc);
if eof(tfm_file) then abort;
for k ← np + 1 to 7 do font_info[param_base[f] + k − 1].sc ← 0;
end
```

This code is used in section 562.

576. 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* 576 ⟩ ≡
  if *np* ≥ 7 **then** *font_params*[*f*] ← *np* **else** *font_params*[*f*] ← 7;
  hyphen_char[*f*] ← *default_hyphen_char*; *skew_char*[*f*] ← *default_skew_char*;
  if *bch_label* < *nl* **then** *bchar_label*[*f*] ← *bch_label* + *lig_kern_base*[*f*]
  else *bchar_label*[*f*] ← *non_address*;
  font_bchar[*f*] ← *qi*(*bchar*); *font_false_bchar*[*f*] ← *qi*(*bchar*);
  if *bchar* ≤ *ec* **then**
    if *bchar* ≥ *bc* **then**
      begin *qw* ← *char_info*(*f*)(*bchar*); { N.B.: not *qi*(*bchar*) }
      if *char_exists*(*qw*) **then** *font_false_bchar*[*f*] ← *non_char*;
      end;
  font_name[*f*] ← *nom*; *font_area*[*f*] ← *aire*; *font_bc*[*f*] ← *bc*; *font_ec*[*f*] ← *ec*; *font_glue*[*f*] ← *null*;
  adjust(*char_base*); *adjust*(*width_base*); *adjust*(*lig_kern_base*); *adjust*(*kern_base*); *adjust*(*exten_base*);
  decr(*param_base*[*f*]); *fmem_ptr* ← *fmem_ptr* + *lf*; *font_ptr* ← *f*; *g* ← *f*; **goto** *done*

This code is used in section 562.

577. Before we forget about the format of these tables, let's deal with two of TeX's basic scanning routines related to font information.

⟨ Declare procedures that scan font-related stuff 577 ⟩ ≡
procedure *scan_font_ident*;
  var *f*: *internal_font_number*; *m*: *halfword*;
  begin ⟨ Get the next non-blank non-call token 406 ⟩;
  if *cur_cmd* = *def_font* **then** *f* ← *cur_font*
  else if *cur_cmd* = *set_font* **then** *f* ← *cur_chr*
    else if *cur_cmd* = *def_family* **then**
        begin *m* ← *cur_chr*; *scan_four_bit_int*; *f* ← *equiv*(*m* + *cur_val*);
        end
      else begin *print_err*("`Missing␣font␣identifier`");
        help2("`I␣was␣looking␣for␣a␣control␣sequence␣whose`")
        ("`current␣meaning␣has␣been␣defined␣by␣\font.`"); *back_error*; *f* ← *null_font*;
        end;
  cur_val ← *f*;
  end;

See also section 578.

This code is used in section 409.

578. 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 577 ⟩ +≡
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 580 ⟩
    else $cur\_val \leftarrow n + param\_base[f]$;
    end;
  ⟨ Issue an error message if $cur\_val = fmem\_ptr$ 579 ⟩;
  end;

579. ⟨ Issue an error message if $cur\_val = fmem\_ptr$ 579 ⟩ ≡
  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 578.

580. ⟨ Increase the number of parameters in the last font 580 ⟩ ≡
  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 578.

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

procedure *char_warning*(*f* : *internal_font_number*; *c* : *eight_bits*);
  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;

582. Here is a function that returns a pointer to a character node for a given character in a given font. If that character doesn't exist, *null* is returned instead.

function *new_character*(*f* : *internal_font_number*; *c* : *eight_bits*): *pointer*;
  label *exit*;
  var *p*: *pointer*; { newly allocated node }
  begin if *font_bc*[*f*] $\le$ *c* **then**
    if *font_ec*[*f*] $\ge$ *c* **then**
      if *char_exists*(*char_info*(*f*)(*qi*(*c*))) **then**
        begin *p* $\leftarrow$ *get_avail*; *font*(*p*) $\leftarrow$ *f*; *character*(*p*) $\leftarrow$ *qi*(*c*); *new_character* $\leftarrow$ *p*; **return**;
        end;
  char_warning(*f*, *c*); *new_character* $\leftarrow$ *null*;
exit: **end**;

583. 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.)

584. 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.

585. 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 $0 \le c < 65536$. TeX82 never uses this command, but it should come in handy for extensions of TeX that deal with oriental languages.

set3 130 $c[3]$. Same as *set1*, except that c is three bytes long, so it can be as large as $2^{24} - 1$. Not even the Chinese language has this many characters, but this command might prove useful in some yet unforeseen extension.

set4 131 $c[4]$. Same as *set1*, except that c is four bytes long. Imagine that.

set_rule 132 $a[4]$ $b[4]$. Typeset a solid black rectangle of height a and width b, with its bottom left corner at (h, v). Then set $h \leftarrow h + b$. If either $a \le 0$ 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 $c_0[4]$ $c_1[4]$ ... $c_9[4]$ $p[4]$. 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 $f \leftarrow 1, \ldots, f \leftarrow 63$, 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 $i[1]$ $num[4]$ $den[4]$ $mag[4]$ $k[1]$ $x[k]$. Beginning of the preamble; this must come at the very beginning of the file. Parameters i, num, den, mag, k, and x are explained below.

post 248. Beginning of the postamble, see below.

post_post 249. Ending of the postamble, see below.

Commands 250–255 are undefined at the present time.

586. **define** *set_char_0* = 0 { typeset character 0 and move right }
define *set1* = 128 { typeset a character and move right }
define *set_rule* = 132 { typeset a rule and move right }
define *put_rule* = 137 { typeset a rule }
define *nop* = 138 { no operation }
define *bop* = 139 { beginning of page }
define *eop* = 140 { ending of page }
define *push* = 141 { save the current positions }
define *pop* = 142 { restore previous positions }
define *right1* = 143 { move right }
define *w0* = 147 { move right by w }
define *w1* = 148 { move right and set w }
define *x0* = 152 { move right by x }
define *x1* = 153 { move right and set x }
define *down1* = 157 { move down }
define *y0* = 161 { move down by y }
define *y1* = 162 { move down and set y }
define *z0* = 166 { move down by z }
define *z1* = 167 { move down and set z }
define *fnt_num_0* = 171 { set current font to 0 }
define *fnt1* = 235 { set current font }
define *xxx1* = 239 { extension to `DVI` primitives }
define *xxx4* = 242 { potentially long extension to `DVI` primitives }
define *fnt_def1* = 243 { define the meaning of a font number }
define *pre* = 247 { preamble }
define *post* = 248 { postamble beginning }
define *post_post* = 249 { postamble ending }

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

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

The i byte identifies `DVI` format; currently this byte is always set to 2. (The value $i = 3$ is currently used for an extended format that allows a mixture of right-to-left and left-to-right typesetting. Some day we will set $i = 4$, when `DVI` format makes another incompatible change—perhaps in the year 2048.)

The next two parameters, *num* and *den*, are positive integers that define the units of measurement; they are the numerator and denominator of a fraction by which all dimensions in the `DVI` file could be multiplied in order to get lengths in units of 10^{-7} meters. Since 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* $= 2$ { identifies the kind of `DVI` files described here }

588. 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*.

589. 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.

590. 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

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

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.

591. 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.)

592. Shipping pages out. After considering TEX's eyes and stomach, we come now to the bowels.

The *ship_out* procedure is given a pointer to a box; its mission is to describe that box in `DVI` form, outputting a "page" to *dvi_file*. The `DVI` coordinates $(h, v) = (0, 0)$ should correspond to the upper left corner of the box being shipped.

Since boxes can be inside of boxes inside of boxes, the main work of *ship_out* is done by two mutually recursive routines, *hlist_out* and *vlist_out*, which traverse the hlists and vlists inside of horizontal and vertical boxes.

As individual pages are being processed, we need to accumulate information about the entire set of pages, since such statistics must be reported in the postamble. The global variables *total_pages*, *max_v*, *max_h*, *max_push*, and *last_bop* are used to record this information.

The variable *doing_leaders* is *true* while leaders are being output. The variable *dead_cycles* contains the number of times an output routine has been initiated since the last *ship_out*.

A few additional global variables are also defined here for use in *vlist_out* and *hlist_out*. They could have been local variables, but that would waste stack space when boxes are deeply nested, since the values of these variables are not needed during recursive calls.

⟨Global variables 13⟩ +≡
total_pages: *integer*; { the number of pages that have been shipped out }
max_v: *scaled*; { maximum height-plus-depth of pages shipped so far }
max_h: *scaled*; { maximum width of pages shipped so far }
max_push: *integer*; { deepest nesting of *push* commands encountered so far }
last_bop: *integer*; { location of previous *bop* in the `DVI` output }
dead_cycles: *integer*; { recent outputs that didn't ship anything out }
doing_leaders: *boolean*; { are we inside a leader box? }

c, *f*: *quarterword*; { character and font in current *char_node* }
rule_ht, *rule_dp*, *rule_wd*: *scaled*; { size of current rule being output }
g: *pointer*; { current glue specification }
lq, *lr*: *integer*; { quantities used in calculations for leaders }

593. ⟨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;

594. 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 ≥ *dvi_gone*.

⟨Types in the outer block 18⟩ +≡
dvi_index = 0 .. *dvi_buf_size*; { an index into the output buffer }

595. Some systems may find it more efficient to make *dvi_buf* a **packed** array, since output of four bytes at once may be facilitated.

⟨ Global variables 13 ⟩ +≡
dvi_buf: **array** [*dvi_index*] **of** *eight_bits*; { buffer for DVI output }
half_buf: *dvi_index*; { half of *dvi_buf_size* }
dvi_limit: *dvi_index*; { end of the current half buffer }
dvi_ptr: *dvi_index*; { the next available buffer address }
dvi_offset: *integer*; { *dvi_buf_size* times the number of times the output buffer has been fully emptied }
dvi_gone: *integer*; { the number of bytes already output to *dvi_file* }

596. 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;

597. The actual output of *dvi_buf* [*a* . . *b*] to *dvi_file* is performed by calling *write_dvi*(*a*, *b*). For best results, this procedure should be optimized to run as fast as possible on each particular system, since it is part of TEX's inner loop. It is safe to assume that a and $b+1$ will both be multiples of 4 when *write_dvi*(*a*, *b*) is called; therefore it is possible on many machines to use efficient methods to pack four bytes per word and to output an array of words with one system call.

```
procedure write_dvi(a, b : dvi_index);
  var k: dvi_index;
  begin for k ← a to b do write(dvi_file, dvi_buf[k]);
  end;
```

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

```
  define dvi_out(#) ≡ begin dvi_buf[dvi_ptr] ← #; incr(dvi_ptr);
        if dvi_ptr = dvi_limit then dvi_swap;
        end
procedure dvi_swap; { outputs half of the buffer }
  begin if dvi_limit = dvi_buf_size then
    begin write_dvi(0, half_buf − 1); dvi_limit ← half_buf; dvi_offset ← dvi_offset + dvi_buf_size;
    dvi_ptr ← 0;
    end
  else begin write_dvi(half_buf, dvi_buf_size − 1); dvi_limit ← dvi_buf_size;
    end;
  dvi_gone ← dvi_gone + half_buf;
  end;
```

599. 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* 599 ⟩ ≡
```
  if dvi_limit = half_buf then write_dvi(half_buf, dvi_buf_size − 1);
  if dvi_ptr > 0 then write_dvi(0, dvi_ptr − 1)
```
This code is used in section 642.

600. 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;

601. A mild optimization of the output is performed by the *dvi_pop* routine, which issues a *pop* unless it is possible to cancel a '*push pop*' pair. The parameter to *dvi_pop* is the byte address following the old *push* that matches the new *pop*.

procedure *dvi_pop*(*l* : *integer*);
  begin if $(l = dvi\_offset + dvi\_ptr) \land (dvi\_ptr > 0)$ **then** *decr*(*dvi_ptr*)
  else *dvi_out*(*pop*);
  end;

602. Here's a procedure that outputs a font definition. Since TEX82 uses at most 256 different fonts per job, *fnt_def1* is always used as the command code.

procedure *dvi_font_def*(*f* : *internal_font_number*);
  var *k*: *pool_pointer*; { index into *str_pool* }
  begin *dvi_out*(*fnt_def1*); *dvi_out*($f - font\_base - 1$);
  dvi_out(*qo*(*font_check*[*f*].*b0*)); *dvi_out*(*qo*(*font_check*[*f*].*b1*)); *dvi_out*(*qo*(*font_check*[*f*].*b2*));
  dvi_out(*qo*(*font_check*[*f*].*b3*));
  dvi_four(*font_size*[*f*]); *dvi_four*(*font_dsize*[*f*]);
  dvi_out(*length*(*font_area*[*f*])); *dvi_out*(*length*(*font_name*[*f*]));
  ⟨Output the font name whose internal number is *f* 603⟩;
  end;

603. ⟨Output the font name whose internal number is *f* 603⟩ ≡
  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 602.

604. 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.)

605. 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 }

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

607. 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" 611 ⟩;
  ⟨ Generate a *down* or *right* command for w and **return** 610 ⟩;
found: ⟨ Generate a *y0* or *z0* command in order to reuse a previous appearance of w 609 ⟩;
exit: **end**;

608. 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 }

609. 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* 609 ⟩ ≡

```
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 607.

610. ⟨ Generate a *down* or *right* command for *w* and **return** 610 ⟩ ≡

```
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 607.

611. 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" 611 ⟩ ≡
p ← *link*(*q*); *mstate* ← *none_seen*;
while *p* ≠ *null* **do**
  begin if *width*(*p*) = *w* **then** ⟨ Consider a node with matching width; **goto** *found* if it's a hit 612 ⟩
  else case *mstate* + *info*(*p*) **of**
    none_seen + *y_here*: *mstate* ← *y_seen*;
    none_seen + *z_here*: *mstate* ← *z_seen*;
    y_seen + *z_here*, *z_seen* + *y_here*: **goto** *not_found*;
    othercases *do_nothing*
    endcases;
  p ← *link*(*p*);
  end;
not_found:

This code is used in section 607.

612. 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 612 ⟩ ≡
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* 613 ⟩;
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* 614 ⟩;
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 611.

613. ⟨ Change buffered instruction to *y* or *w* and **goto** *found* 613 ⟩ ≡
begin *k* ← *location*(*p*) − *dvi_offset*;
if *k* < 0 **then** *k* ← *k* + *dvi_buf_size*;
dvi_buf[*k*] ← *dvi_buf*[*k*] + *y1* − *down1*; *info*(*p*) ← *y_here*; **goto** *found*;
end

This code is used in section 612.

614. ⟨ Change buffered instruction to *z* or *x* and **goto** *found* 614 ⟩ ≡
begin *k* ← *location*(*p*) − *dvi_offset*;
if *k* < 0 **then** *k* ← *k* + *dvi_buf_size*;
dvi_buf[*k*] ← *dvi_buf*[*k*] + *z1* − *down1*; *info*(*p*) ← *z_here*; **goto** *found*;
end

This code is used in section 612.

615. 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 ≥ l }
  label done, exit;
  var p: pointer; { node being deleted }
  begin while down_ptr ≠ null do
    begin if location(down_ptr) < l then goto done;
    p ← down_ptr; down_ptr ← link(p); free_node(p, movement_node_size);
    end;
done: while right_ptr ≠ null do
    begin if location(right_ptr) < l then return;
    p ← right_ptr; right_ptr ← link(p); free_node(p, movement_node_size);
    end;
exit: end;
```

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

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

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

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

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

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

617. ⟨ Initialize variables as *ship_out* begins 617 ⟩ ≡
dvi_h ← 0; *dvi_v* ← 0; *cur_h* ← *h_offset*; *dvi_f* ← *null_font*; *ensure_dvi_open*;
if *total_pages* = 0 **then**
begin *dvi_out*(*pre*); *dvi_out*(*id_byte*); { output the preamble }
dvi_four(25400000); *dvi_four*(473628672); { conversion ratio for sp }
prepare_mag; *dvi_four*(*mag*); { magnification factor is frozen }
old_setting ← *selector*; *selector* ← *new_string*; *print*("␣TeX␣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

This code is used in section 640.

618. 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 }

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

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

⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1368 ⟩
procedure *hlist_out*; { output an *hlist_node* box }
label *reswitch*, *move_past*, *fin_rule*, *next_p*;
var *base_line*: *scaled*; { the baseline coordinate for this box }
left_edge: *scaled*; { the left coordinate for this box }
save_h, *save_v*: *scaled*; { what *dvi_h* and *dvi_v* should pop to }
this_box: *pointer*; { pointer to containing box }
g_order: *glue_ord*; { applicable order of infinity for glue }
g_sign: *normal* .. *shrinking*; { selects type of glue }
p: *pointer*; { current position in the hlist }
save_loc: *integer*; { DVI byte location upon entry }
leader_box: *pointer*; { the leader box being replicated }
leader_wd: *scaled*; { width of leader box being replicated }
lx: *scaled*; { extra space between leader boxes }
outer_doing_leaders: *boolean*; { were we doing leaders? }
edge: *scaled*; { 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* 620 ⟩;
prune_movements(*save_loc*);
if *cur_s* > 0 **then** *dvi_pop*(*save_loc*);
decr(*cur_s*);
end;

620. We ought to give special care to the efficiency of one part of *hlist_out*, since it belongs to TEX's inner loop. When a *char_node* is encountered, we save a little time by processing several nodes in succession until reaching a non-*char_node*. The program uses the fact that $set\_char\_0 = 0$.

⟨ Output node p for *hlist_out* and move to the next node, maintaining the condition $cur\_v = base\_line$ 620 ⟩ ≡

```
reswitch: if is_char_node(p) then
    begin synch_h; synch_v;
    repeat f ← font(p); c ← character(p);
      if f ≠ dvi_f then ⟨Change font dvi_f to f 621⟩;
      if c ≥ qi(128) then dvi_out(set1);
      dvi_out(qo(c));
      cur_h ← cur_h + char_width(f)(char_info(f)(c)); p ← link(p);
    until ¬is_char_node(p);
    dvi_h ← cur_h;
    end
  else ⟨Output the non-char_node p for hlist_out and move to the next node 622⟩
```

This code is used in section 619.

621. ⟨ Change font *dvi_f* to f 621 ⟩ ≡

```
  begin if ¬font_used[f] then
    begin dvi_font_def(f); font_used[f] ← true;
    end;
  if f ≤ 64 + font_base then dvi_out(f − font_base − 1 + fnt_num_0)
  else begin dvi_out(fnt1); dvi_out(f − font_base − 1);
    end;
  dvi_f ← f;
  end
```

This code is used in section 620.

622. ⟨ Output the non-*char_node* p for *hlist_out* and move to the next node 622 ⟩ ≡

```
  begin case type(p) of
  hlist_node, vlist_node: ⟨Output a box in an hlist 623⟩;
  rule_node: begin rule_ht ← height(p); rule_dp ← depth(p); rule_wd ← width(p); goto fin_rule;
    end;
  whatsit_node: ⟨Output the whatsit node p in an hlist 1367⟩;
  glue_node: ⟨Move right or output leaders 625⟩;
  kern_node, math_node: cur_h ← cur_h + width(p);
  ligature_node: ⟨Make node p look like a char_node and goto reswitch 652⟩;
  othercases do_nothing
  endcases;
  goto next_p;
fin_rule: ⟨Output a rule in an hlist 624⟩;
move_past: cur_h ← cur_h + rule_wd;
next_p: p ← link(p);
  end
```

This code is used in section 620.

623. ⟨ Output a box in an hlist 623 ⟩ ≡

```
if list_ptr(p) = null then cur_h ← cur_h + width(p)
else begin save_h ← dvi_h; save_v ← dvi_v; cur_v ← base_line + shift_amount(p);
      { shift the box down }
  temp_ptr ← p; edge ← cur_h;
  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 622.

624. ⟨ Output a rule in an hlist 624 ⟩ ≡

```
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 622.

625. **define** *billion* ≡ *float_constant*(1000000000)

```
define vet_glue(#) ≡ glue_temp ← #;
      if glue_temp > billion then glue_temp ← billion
      else if glue_temp < −billion then glue_temp ← −billion
```

⟨ Move right or output leaders 625 ⟩ ≡

```
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 626 ⟩;
goto move_past;
end
```

This code is used in section 622.

626. ⟨ Output leaders in an hlist, **goto** *fin_rule* if a rule or to *next_p* if done 626 ⟩ ≡

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

This code is used in section 625.

627. 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 627 ⟩ ≡

```
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 626.

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

⟨ Output a leader box at *cur_h*, then advance *cur_h* by *leader_wd* + *lx* 628 ⟩ ≡

```
begin cur_v ← base_line + shift_amount(leader_box); synch_v; save_v ← dvi_v;
synch_h; save_h ← dvi_h; temp_ptr ← leader_box; outer_doing_leaders ← doing_leaders;
doing_leaders ← true;
if type(leader_box) = vlist_node then vlist_out else hlist_out;
doing_leaders ← outer_doing_leaders; dvi_v ← save_v; dvi_h ← save_h; cur_v ← base_line;
cur_h ← save_h + leader_wd + lx;
end
```

This code is used in section 626.

629. The *vlist_out* routine is similar to *hlist_out*, but a bit simpler.

procedure *vlist_out*; { output a *vlist_node* box }
label *move_past*, *fin_rule*, *next_p*;
var *left_edge*: *scaled*; { the left coordinate for this box }
top_edge: *scaled*; { the top coordinate for this box }
save_h, *save_v*: *scaled*; { what *dvi_h* and *dvi_v* should pop to }
this_box: *pointer*; { pointer to containing box }
g_order: *glue_ord*; { applicable order of infinity for glue }
g_sign: *normal* .. *shrinking*; { selects type of glue }
p: *pointer*; { current position in the vlist }
save_loc: *integer*; { DVI byte location upon entry }
leader_box: *pointer*; { the leader box being replicated }
leader_ht: *scaled*; { height of leader box being replicated }
lx: *scaled*; { extra space between leader boxes }
outer_doing_leaders: *boolean*; { were we doing leaders? }
edge: *scaled*; { bottom boundary of leader space }
glue_temp: *real*; { glue value before rounding }
cur_glue: *real*; { glue seen so far }
cur_g: *scaled*; { rounded equivalent of *cur_glue* times the glue ratio }
begin *cur_g* ← 0; *cur_glue* ← *float_constant*(0); *this_box* ← *temp_ptr*; *g_order* ← *glue_order*(*this_box*);
g_sign ← *glue_sign*(*this_box*); *p* ← *list_ptr*(*this_box*); *incr*(*cur_s*);
if *cur_s* > 0 **then** *dvi_out*(*push*);
if *cur_s* > *max_push* **then** *max_push* ← *cur_s*;
save_loc ← *dvi_offset* + *dvi_ptr*; *left_edge* ← *cur_h*; *cur_v* ← *cur_v* − *height*(*this_box*); *top_edge* ← *cur_v*;
while *p* ≠ *null* **do** ⟨ Output node *p* for *vlist_out* and move to the next node, maintaining the condition *cur_h* = *left_edge* 630 ⟩;
prune_movements(*save_loc*);
if *cur_s* > 0 **then** *dvi_pop*(*save_loc*);
decr(*cur_s*);
end;

630. ⟨ Output node *p* for *vlist_out* and move to the next node, maintaining the condition *cur_h* = *left_edge* 630 ⟩ ≡
begin if *is_char_node*(*p*) **then** *confusion*("vlistout")
else ⟨ Output the non-*char_node* *p* for *vlist_out* 631 ⟩;
next_p: *p* ← *link*(*p*);
end

This code is used in section 629.

631. ⟨ Output the non-*char_node* *p* for *vlist_out* 631 ⟩ ≡
begin case *type*(*p*) **of**
hlist_node, *vlist_node*: ⟨ Output a box in a vlist 632 ⟩;
rule_node: **begin** *rule_ht* ← *height*(*p*); *rule_dp* ← *depth*(*p*); *rule_wd* ← *width*(*p*); **goto** *fin_rule*;
end;
whatsit_node: ⟨ Output the whatsit node *p* in a vlist 1366 ⟩;
glue_node: ⟨ Move down or output leaders 634 ⟩;
kern_node: *cur_v* ← *cur_v* + *width*(*p*);
othercases *do_nothing*
endcases;
goto *next_p*;
fin_rule: ⟨ Output a rule in a vlist, **goto** *next_p* 633 ⟩;
move_past: *cur_v* ← *cur_v* + *rule_ht*;
end

This code is used in section 630.

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

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

This code is used in section 631.

633. ⟨ Output a rule in a vlist, **goto** *next_p* 633 ⟩ ≡
if *is_running*(*rule_wd*) **then** *rule_wd* ← *width*(*this_box*);
rule_ht ← *rule_ht* + *rule_dp*; { this is the rule thickness }
cur_v ← *cur_v* + *rule_ht*;
if (*rule_ht* > 0) ∧ (*rule_wd* > 0) **then** { we don't output empty rules }
begin *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 631.

634. ⟨ Move down or output leaders 634 ⟩ ≡

begin $g \leftarrow glue\_ptr(p)$; $rule\_ht \leftarrow width(g) - cur\_g$;
if $g\_sign \neq normal$ **then**
  begin if $g\_sign = stretching$ **then**
    begin if $stretch\_order(g) = g\_order$ **then**
      begin $cur\_glue \leftarrow cur\_glue + stretch(g)$; $vet\_glue(float(glue\_set(this\_box)) * cur\_glue)$;
      $cur\_g \leftarrow round(glue\_temp)$;
      end;
    end
  else if $shrink\_order(g) = g\_order$ **then**
      begin $cur\_glue \leftarrow cur\_glue - shrink(g)$; $vet\_glue(float(glue\_set(this\_box)) * cur\_glue)$;
      $cur\_g \leftarrow round(glue\_temp)$;
      end;
  end;
$rule\_ht \leftarrow rule\_ht + cur\_g$;
if $subtype(p) \geq a\_leaders$ **then**
  ⟨ Output leaders in a vlist, **goto** *fin_rule* if a rule or to *next_p* if done 635 ⟩;
goto *move_past*;
end

This code is used in section 631.

635. ⟨ Output leaders in a vlist, **goto** *fin_rule* if a rule or to *next_p* if done 635 ⟩ ≡

begin $leader\_box \leftarrow leader\_ptr(p)$;
if $type(leader\_box) = rule\_node$ **then**
  begin $rule\_wd \leftarrow width(leader\_box)$; $rule\_dp \leftarrow 0$; **goto** *fin_rule*;
  end;
$leader\_ht \leftarrow height(leader\_box) + depth(leader\_box)$;
if $(leader\_ht > 0) \land (rule\_ht > 0)$ **then**
  begin $rule\_ht \leftarrow rule\_ht + 10$; { compensate for floating-point rounding }
  $edge \leftarrow cur\_v + rule\_ht$; $lx \leftarrow 0$; ⟨ Let *cur_v* be the position of the first box, and set $leader\_ht + lx$ to the spacing between corresponding parts of boxes 636 ⟩;
  while $cur\_v + leader\_ht \leq edge$ **do**
    ⟨ Output a leader box at *cur_v*, then advance *cur_v* by $leader\_ht + lx$ 637 ⟩;
  $cur\_v \leftarrow edge - 10$; **goto** *next_p*;
  end;
end

This code is used in section 634.

636. ⟨ Let *cur_v* be the position of the first box, and set $leader\_ht + lx$ to the spacing between corresponding parts of boxes 636 ⟩ ≡

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

This code is used in section 635.

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

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

This code is used in section 635.

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

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

639. ⟨ Flush the box from memory, showing statistics if requested 639 ⟩ ≡

```
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 638.

640. ⟨ Ship box p out 640 ⟩ ≡

```
⟨ Update the values of max_h and max_v; but if the page is too large, goto done 641 ⟩;
⟨ Initialize variables as ship_out begins 617 ⟩;
page_loc ← dvi_offset + dvi_ptr; dvi_out(bop);
for k ← 0 to 9 do dvi_four(count(k));
dvi_four(last_bop); last_bop ← page_loc; cur_v ← height(p) + v_offset; temp_ptr ← p;
if type(p) = vlist_node then vlist_out else hlist_out;
dvi_out(eop); incr(total_pages); cur_s ← −1;
done:
```

This code is used in section 638.

641. 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* 641 ⟩ ≡

```
if (height(p) > max_dimen) ∨ (depth(p) > max_dimen) ∨
      (height(p) + depth(p) + v_offset > max_dimen) ∨ (width(p) + h_offset > max_dimen) then
  begin print_err("Huge␣page␣cannot␣be␣shipped␣out");
  help2("The␣page␣just␣created␣is␣more␣than␣18␣feet␣tall␣or")
  ("more␣than␣18␣feet␣wide,␣so␣I␣suspect␣something␣went␣wrong."); error;
  if tracing_output ≤ 0 then
    begin begin_diagnostic; print_nl("The␣following␣box␣has␣been␣deleted:"); show_box(p);
    end_diagnostic(true);
    end;
  goto done;
  end;
if height(p) + depth(p) + v_offset > max_v then max_v ← height(p) + depth(p) + v_offset;
if width(p) + h_offset > max_h then max_h ← width(p) + h_offset
```

This code is used in section 640.

642. 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 642⟩ ≡
  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 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 643⟩;
    dvi_out(post_post); dvi_four(last_bop); dvi_out(id_byte);
    k ← 4 + ((dvi_buf_size − dvi_ptr) mod 4); { the number of 223's }
    while k > 0 do
      begin dvi_out(223); decr(k);
      end;
    ⟨Empty the last bytes out of dvi_buf 599⟩;
    print_nl("Output␣written␣on␣"); slow_print(output_file_name); print("␣("); print_int(total_pages);
    print("␣page");
    if total_pages ≠ 1 then print_char("s");
    print(",␣"); print_int(dvi_offset + dvi_ptr); print("␣bytes)."); b_close(dvi_file);
    end
```

This code is used in section 1333.

643. ⟨Output the font definitions for all fonts that were used 643⟩ ≡

```
  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 642.

644. 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* }

645. 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:

$$\mathit{save\_ptr} \leftarrow \mathit{save\_ptr} - 2;$$
$$\mathit{hpack}(p, \mathit{saved}(1), \mathit{saved}(0)).$$

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* ← *exactly*
  else if *scan_keyword*("spread") **then** *spec_code* ← *additional*
    else begin *spec_code* ← *additional*; *cur_val* ← 0; **goto** *found*;
      end;
  scan_normal_dimen;
found: **if** *three_codes* **then**
    begin *saved*(0) ← *s*; *incr*(*save_ptr*);
    end;
  saved(0) ← *spec_code*; *saved*(1) ← *cur_val*; *save_ptr* ← *save_ptr* + 2; *new_save_level*(*c*); *scan_left_brace*;
  end;

646. 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 }

647. 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 }

648. ⟨Set initial values of key variables 21⟩ +≡
adjust_tail ← *null*; *last_badness* ← 0;

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

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

650. ⟨ Clear dimensions to zero 650 ⟩ ≡
$d \leftarrow 0$; $x \leftarrow 0$; $total\_stretch[normal] \leftarrow 0$; $total\_shrink[normal] \leftarrow 0$; $total\_stretch[fil] \leftarrow 0$;
$total\_shrink[fil] \leftarrow 0$; $total\_stretch[fill] \leftarrow 0$; $total\_shrink[fill] \leftarrow 0$; $total\_stretch[filll] \leftarrow 0$;
$total\_shrink[filll] \leftarrow 0$

This code is used in sections 649 and 668.

651. ⟨ Examine node p in the hlist, taking account of its effect on the dimensions of the new box, or moving it to the adjustment list; then advance p to the next node 651 ⟩ ≡
begin *reswitch*: **while** $is\_char\_node(p)$ **do** ⟨ Incorporate character dimensions into the dimensions of the hbox that will contain it, then move to the next node 654 ⟩;
if $p \neq null$ **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 653 ⟩;
  ins_node, *mark_node*, *adjust_node*: **if** $adjust\_tail \neq null$ **then**
    ⟨ Transfer node p to the adjustment list 655 ⟩;
  whatsit_node: ⟨ Incorporate a whatsit node into an hbox 1360 ⟩;
  glue_node: ⟨ Incorporate glue into the horizontal totals 656 ⟩;
  kern_node, *math_node*: $x \leftarrow x + width(p)$;
  ligature_node: ⟨ Make node p look like a *char_node* and **goto** *reswitch* 652 ⟩;
  othercases *do_nothing*
  endcases;
  $p \leftarrow link(p)$;
  end;
end

This code is used in section 649.

652. ⟨ Make node p look like a *char_node* and **goto** *reswitch* 652 ⟩ ≡
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 622, 651, and 1147.

653. 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 653 ⟩ ≡
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 651.

654. 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 654 ⟩ ≡
begin $f \leftarrow font(p)$; $i \leftarrow char\_info(f)(character(p))$; $hd \leftarrow height\_depth(i)$; $x \leftarrow x + char\_width(f)(i)$;
$s \leftarrow char\_height(f)(hd)$; **if** $s > h$ **then** $h \leftarrow s$;
$s \leftarrow char\_depth(f)(hd)$; **if** $s > d$ **then** $d \leftarrow s$;
$p \leftarrow link(p)$;
end

This code is used in section 651.

655. 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 655 ⟩ ≡
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 651.

656. ⟨ Incorporate glue into the horizontal totals 656 ⟩ ≡
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 651.

657. 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* 657 ⟩ ≡
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* 658 ⟩
  else ⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 664 ⟩

This code is used in section 649.

658. ⟨ Determine horizontal glue stretch setting, then **return** or **goto** *common_ending* 658 ⟩ ≡
begin ⟨ Determine the stretch order 659 ⟩;
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 660 ⟩;
return;
end

This code is used in section 657.

659. ⟨ Determine the stretch order 659 ⟩ ≡
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 658, 673, and 796.

660. ⟨ Report an underfull hbox and **goto** *common_ending*, if this box is sufficiently bad 660 ⟩ ≡
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 658.

661. 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 }

662. ⟨ Set initial values of key variables 21 ⟩ +≡
pack_begin_line ← 0;

663. ⟨ Finish issuing a diagnostic message for an overfull or underfull hbox 663 ⟩ ≡
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*; *short_display*(*list_ptr*(*r*)); *print_ln*;
begin_diagnostic; *show_box*(*r*); *end_diagnostic*(*true*)

This code is used in section 649.

664. ⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 664 ⟩ ≡
begin ⟨ Determine the shrink order 665 ⟩;
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 666 ⟩;
 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 667 ⟩;
return;
end

This code is used in section 657.

665. ⟨ Determine the shrink order 665 ⟩ ≡
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 664, 676, and 796.

666. ⟨ Report an overfull hbox and **goto** *common_ending*, if this box is sufficiently bad 666 ⟩ ≡
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 664.

667. ⟨Report a tight hbox and **goto** *common_ending*, if this box is sufficiently bad 667⟩ ≡

```
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 664.

668. 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 650⟩;
  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 669⟩;
  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 672⟩;
common_ending: ⟨Finish issuing a diagnostic message for an overfull or underfull vbox 675⟩;
exit: vpackage ← r;
  end;
```

669. ⟨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 669⟩ ≡

```
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 670⟩;
  whatsit_node: ⟨Incorporate a whatsit node into a vbox 1359⟩;
  glue_node: ⟨Incorporate glue into the vertical totals 671⟩;
  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 668.

670. ⟨Incorporate box dimensions into the dimensions of the vbox that will contain it 670⟩ ≡

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 669.

671. ⟨Incorporate glue into the vertical totals 671⟩ ≡

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 669.

672. 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* 672⟩ ≡

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* 673⟩
  else ⟨Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 676⟩

This code is used in section 668.

673. ⟨Determine vertical glue stretch setting, then **return** or **goto** *common_ending* 673⟩ ≡

begin ⟨Determine the stretch order 659⟩;
$glue\_order(r) \leftarrow o$; $glue\_sign(r) \leftarrow stretching$;
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 674⟩;
return;
end

This code is used in section 672.

674. ⟨ Report an underfull vbox and **goto** *common_ending*, if this box is sufficiently bad 674 ⟩ ≡

```
begin last_badness ← badness(x, total_stretch[normal]);
if last_badness > vbadness then
  begin print_ln;
  if last_badness > 100 then print_nl("Underfull") else print_nl("Loose");
  print("␣\vbox␣(badness␣"); print_int(last_badness); goto common_ending;
  end;
end
```

This code is used in section 673.

675. ⟨ Finish issuing a diagnostic message for an overfull or underfull vbox 675 ⟩ ≡

```
if output_active then print(")␣has␣occurred␣while␣\output␣is␣active")
else begin if pack_begin_line ≠ 0 then  { it's actually negative }
    begin print(")␣in␣alignment␣at␣lines␣"); print_int(abs(pack_begin_line)); print("--");
    end
  else print(")␣detected␣at␣line␣");
  print_int(line); print_ln;
  end;
begin_diagnostic; show_box(r); end_diagnostic(true)
```

This code is used in section 668.

676. ⟨ Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 676 ⟩ ≡

```
begin ⟨ Determine the shrink order 665 ⟩;
glue_order(r) ← o; glue_sign(r) ← shrinking;
if total_shrink[o] ≠ 0 then glue_set(r) ← unfloat((−x)/total_shrink[o])
else begin glue_sign(r) ← normal; set_glue_ratio_zero(glue_set(r));  { there's nothing to shrink }
  end;
if (total_shrink[o] < −x) ∧ (o = normal) ∧ (list_ptr(r) ≠ null) then
  begin last_badness ← 1000000; set_glue_ratio_one(glue_set(r));  { use the maximum shrinkage }
  ⟨ Report an overfull vbox and goto common_ending, if this box is sufficiently bad 677 ⟩;
  end
else if o = normal then
    if list_ptr(r) ≠ null then
      ⟨ Report a tight vbox and goto common_ending, if this box is sufficiently bad 678 ⟩;
return;
end
```

This code is used in section 672.

677. ⟨ Report an overfull vbox and **goto** *common_ending*, if this box is sufficiently bad 677 ⟩ ≡

```
if (−x − total_shrink[normal] > vfuzz) ∨ (vbadness < 100) then
  begin print_ln; print_nl("Overfull␣\vbox␣("); print_scaled(−x − total_shrink[normal]);
  print("pt␣too␣high"); goto common_ending;
  end
```

This code is used in section 676.

678. ⟨ Report a tight vbox and **goto** *common_ending*, if this box is sufficiently bad 678 ⟩ ≡

```
begin last_badness ← badness(−x, total_shrink[normal]);
if last_badness > vbadness then
  begin print_ln; print_nl("Tight␣\vbox␣(badness␣"); print_int(last_badness); goto common_ending;
  end;
end
```

This code is used in section 676.

679. 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* ← *width*(*baseline_skip*) − *prev_depth* − *height*(*b*);
    if *d* < *line_skip_limit* **then** *p* ← *new_param_glue*(*line_skip_code*)
    else begin *p* ← *new_skip_param*(*baseline_skip_code*); *width*(*temp_ptr*) ← *d*;
        { *temp_ptr* = *glue_ptr*(*p*) }
      end;
    link(*tail*) ← *p*; *tail* ← *p*;
    end;
  link(*tail*) ← *b*; *tail* ← *b*; *prev_depth* ← *depth*(*b*);
  end;

680. 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.

681. 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 }

682. Each portion of a formula is classified as Ord, Op, Bin, Rel, Open, Close, Punct, or Inner, for purposes of spacing and line breaking. An *ord_noad*, *op_noad*, *bin_noad*, *rel_noad*, *open_noad*, *close_noad*, *punct_noad*, or *inner_noad* is used to represent portions of the various types. For example, an '`=`' sign in a formula leads to the creation of a *rel_noad* whose *nucleus* field is a representation of an equals sign (usually *fam* = 0, *character* = ´75). A formula preceded by `\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 Open }
define *close_noad* = *ord_noad* + 5 { *type* of a noad classified Close }
define *punct_noad* = *ord_noad* + 6 { *type* of a noad classified Punct }
define *inner_noad* = *ord_noad* + 7 { *type* of a noad classified Inner }
define *limits* = 1 { *subtype* of *op_noad* whose scripts are to be above, below }
define *no_limits* = 2 { *subtype* of *op_noad* whose scripts are to be normal }

683. 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 }

684. 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*;

685. ⟨ 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*;

686. 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;

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

And finally, we have *left_noad* and *right_noad* types, to implement TEX's `\left` and `\right`. 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*)

688. 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;

689. 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;

690. 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 690 ⟩ ≡
style_node: *print_style*(*subtype*(*p*));
choice_node: ⟨ Display choice node *p* 695 ⟩;
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* 696 ⟩;
fraction_noad: ⟨ Display fraction noad *p* 697 ⟩;

This code is used in section 183.

691. Here are some simple routines used in the display of noads.

⟨ Declare procedures needed for displaying the elements of mlists 691 ⟩ ≡
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 692 and 694.

This code is used in section 179.

692. 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 691⟩ +≡
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;
```

693. 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;
```

694.
```
⟨Declare procedures needed for displaying the elements of mlists 691⟩ +≡
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;
```

695. ⟨ Display choice node *p* 695 ⟩ ≡
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 690.

696. ⟨ Display normal noad *p* 696 ⟩ ≡
begin case *type*(*p*) **of**
ord_noad: *print_esc*("mathord");
op_noad: *print_esc*("mathop");
bin_noad: *print_esc*("mathbin");
rel_noad: *print_esc*("mathrel");
open_noad: *print_esc*("mathopen");
close_noad: *print_esc*("mathclose");
punct_noad: *print_esc*("mathpunct");
inner_noad: *print_esc*("mathinner");
over_noad: *print_esc*("overline");
under_noad: *print_esc*("underline");
vcenter_noad: *print_esc*("vcenter");
radical_noad: **begin** *print_esc*("radical"); *print_delimiter*(*left_delimiter*(*p*));
end;
accent_noad: **begin** *print_esc*("accent"); *print_fam_and_char*(*accent_chr*(*p*));
end;
left_noad: **begin** *print_esc*("left"); *print_delimiter*(*delimiter*(*p*));
end;
right_noad: **begin** *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 690.

697. ⟨ Display fraction noad *p* 697 ⟩ ≡

```
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 690.

698. That which can be displayed can also be destroyed.

⟨ Cases of *flush_node_list* that arise in mlists only 698 ⟩ ≡

```
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.

699. 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;

700. 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

701. 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

702. We also need to compute the change in style between mlists and their subsidiaries. The following macros define the subsidiary style for an overlined nucleus (*cramped_style*), for a subscript or a superscript (*sub_style* or *sup_style*), or for a numerator or denominator (*num_style* or *denom_style*).

define *cramped_style*(#) ≡ 2 ∗ (# **div** 2) + *cramped* { cramp the style }
define *sub_style*(#) ≡ 2 ∗ (# **div** 4) + *script_style* + *cramped* { smaller and cramped }
define *sup_style*(#) ≡ 2 ∗ (# **div** 4) + *script_style* + (# **mod** 2) { smaller }
define *num_style*(#) ≡ # + 2 − 2 ∗ (# **div** 6) { smaller unless already script-script }
define *denom_style*(#) ≡ 2 ∗ (# **div** 2) + *cramped* + 2 − 2 ∗ (# **div** 6) { smaller, cramped }

703. 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* 703 ⟩ ≡
begin if *cur_style* < *script_style* **then** *cur_size* ← *text_size*
else *cur_size* ← 16 ∗ ((*cur_style* − *text_style*) **div** 2);
cur_mu ← *x_over_n*(*math_quad*(*cur_size*), 18);
end

This code is used in sections 720, 726, 730, 754, 760, and 763.

704. Here is a function that returns a pointer to a rule node having a given thickness *t*. The rule will extend horizontally to the boundary of the vlist that eventually contains it.

function *fraction_rule*(*t* : *scaled*): *pointer*; { construct the bar for a fraction }
var *p*: *pointer*; { the new node }
begin *p* ← *new_rule*; *height*(*p*) ← *t*; *depth*(*p*) ← 0; *fraction_rule* ← *p*;
end;

705. The *overbar* function returns a pointer to a vlist box that consists of a given box *b*, above which has been placed a kern of height *k* under a fraction rule of thickness *t* under additional space of height *t*.

function *overbar*(*b* : *pointer*; *k*, *t* : *scaled*): *pointer*;
var *p*, *q*: *pointer*; { nodes being constructed }
begin *p* ← *new_kern*(*k*); *link*(*p*) ← *b*; *q* ← *fraction_rule*(*t*); *link*(*q*) ← *p*; *p* ← *new_kern*(*t*); *link*(*p*) ← *q*;
overbar ← *vpack*(*p*, *natural*);
end;

706. 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* 709 ⟩
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 $f \leftarrow$ *null_font*; $w \leftarrow 0$; *large_attempt* $\leftarrow$ *false*; $z \leftarrow$ *small_fam*(d); $x \leftarrow$ *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 707 ⟩;
    if *large_attempt* **then goto** *found*; { there were none large enough }
    large_attempt $\leftarrow$ *true*; $z \leftarrow$ *large_fam*(d); $x \leftarrow$ *large_char*(d);
    end;
found: **if** $f \neq$ *null_font* **then** ⟨ Make variable b point to a box for (f, c) 710 ⟩
  else begin $b \leftarrow$ *new_null_box*; *width*(b) $\leftarrow$ *null_delimiter_space*;
      { use this width if no delimiter was found }
    end;
  shift_amount(b) $\leftarrow$ *half*(*height*(b) $-$ *depth*(b)) $-$ *axis_height*(s); *var_delimiter* $\leftarrow b$;
  end;

707. 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 707 ⟩ ≡
  if $(z \neq 0) \lor (x \neq$ *min_quarterword*$)$ **then**
    begin $z \leftarrow z + s + 16$;
    repeat $z \leftarrow z - 16$; $g \leftarrow$ *fam_fnt*(z);
      if $g \neq$ *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 708 ⟩;
    until $z < 16$;
    end

This code is used in section 706.

708. ⟨ 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 708 ⟩ ≡

```
begin y ← x;
if (qo(y) ≥ font_bc[g]) ∧ (qo(y) ≤ font_ec[g]) then
  begin continue: q ← 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 707.

709. 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* 709 ⟩ ≡

```
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 711 and 712.

This code is used in section 706.

710. 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) 710 ⟩ ≡

```
  if char_tag(q) = ext_tag then
    ⟨ Construct an extensible character in a new box b, using recipe rem_byte(q) and font f 713 ⟩
  else b ← char_box(f, c)
```

This code is used in section 706.

711. 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* 709 ⟩ +≡
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;

712. Another handy subroutine computes the height plus depth of a given character:

⟨ Declare subprocedures for *var_delimiter* 709 ⟩ +≡
function *height_plus_depth*(*f* : *internal_font_number*; *c* : *quarterword*): *scaled*;
  var *q*: *four_quarters*; *hd*: *eight_bits*; { *height_depth* byte }
  begin *q* ← *char_info*(*f*)(*c*); *hd* ← *height_depth*(*q*);
  height_plus_depth ← *char_height*(*f*)(*hd*) + *char_depth*(*f*)(*hd*);
  end;

713. ⟨ Construct an extensible character in a new box *b*, using recipe *rem_byte*(*q*) and font *f* 713 ⟩ ≡
  begin *b* ← *new_null_box*; *type*(*b*) ← *vlist_node*; *r* ← *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*) 714 ⟩;
  c ← *ext_bot*(*r*);
  if $c \neq$ *min_quarterword* **then** *stack_into_box*(*b*, *f*, *c*);
  c ← *ext_rep*(*r*);
  for *m* ← 1 **to** *n* **do** *stack_into_box*(*b*, *f*, *c*);
  c ← *ext_mid*(*r*);
  if $c \neq$ *min_quarterword* **then**
    begin *stack_into_box*(*b*, *f*, *c*); *c* ← *ext_rep*(*r*);
    for *m* ← 1 **to** *n* **do** *stack_into_box*(*b*, *f*, *c*);
    end;
  c ← *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 710.

714. 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) 714 ⟩ ≡

$c \leftarrow ext\_rep(r)$; $u \leftarrow height\_plus\_depth(f, c)$; $w \leftarrow 0$; $q \leftarrow char\_info(f)(c)$;
$width(b) \leftarrow char\_width(f)(q) + char\_italic(f)(q)$;
$c \leftarrow ext\_bot(r)$; **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 713.

715. 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 $(width(b) \neq w) \wedge (list\_ptr(b) \neq null)$ **then**
    begin if $type(b) = vlist\_node$ **then** $b \leftarrow hpack(b, natural)$;
    $p \leftarrow list\_ptr(b)$;
    if $(is\_char\_node(p)) \wedge (link(p) = null)$ **then**
      begin $f \leftarrow font(p)$; $v \leftarrow char\_width(f)(char\_info(f)(character(p)))$;
      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;

716. 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;
```

717. 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;
```

718. 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;
```

719. 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? }
```

720. The recursion in *mlist_to_hlist* is due primarily to a subroutine called *clean_box* that puts a given noad field into a box using a given math style; *mlist_to_hlist* can call *clean_box*, which can call *mlist_to_hlist*.

The box returned by *clean_box* is "clean" in the sense that its *shift_amount* is zero.

```
procedure mlist_to_hlist; forward;
function clean_box(p : pointer; s : small_number): pointer;
  label found;
  var q: pointer;  { beginning of a list to be boxed }
    save_style: small_number;  { cur_style to be restored }
    x: pointer;  { box to be returned }
    r: pointer;  { temporary pointer }
  begin case math_type(p) of
  math_char: begin cur_mlist ← new_noad; mem[nucleus(cur_mlist)] ← mem[p];
    end;
  sub_box: begin q ← info(p); goto found;
    end;
  sub_mlist: cur_mlist ← info(p);
  othercases begin q ← new_null_box; goto found;
    end
  endcases;
  save_style ← cur_style; cur_style ← s; mlist_penalties ← false;
  mlist_to_hlist; q ← link(temp_head);  { recursive call }
  cur_style ← save_style;  { restore the style }
  ⟨Set up the values of cur_size and cur_mu, based on cur_style 703⟩;
found: if is_char_node(q) ∨ (q = null) then x ← hpack(q, natural)
  else if (link(q) = null) ∧ (type(q) ≤ vlist_node) ∧ (shift_amount(q) = 0) then x ← q
        { it's already clean }
    else x ← hpack(q, natural);
  ⟨Simplify a trivial box 721⟩;
  clean_box ← x;
  end;
```

721. Here we save memory space in a common case.

⟨Simplify a trivial box 721⟩ ≡
  q ← *list_ptr*(*x*);
  if *is_char_node*(*q*) **then**
    begin *r* ← *link*(*q*);
    if $r \neq$ *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 720.

722. 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 723⟩
  else begin if (*qo*(*cur_c*) ≥ *font_bc*[*cur_f*]) ∧ (*qo*(*cur_c*) ≤ *font_ec*[*cur_f*]) **then**
      cur_i ← *char_info*(*cur_f*)(*cur_c*)
    else *cur_i* ← *null_character*;
    if ¬(*char_exists*(*cur_i*)) **then**
      begin *char_warning*(*cur_f*, *qo*(*cur_c*)); *math_type*(*a*) ← *empty*; *cur_i* ← *null_character*;
      end;
    end;
  end;

723. ⟨Complain about an undefined family and set *cur_i* null 723⟩ ≡
  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 722.

724. 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 }

725. 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 }

726. 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 734 ⟩
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* 703 ⟩;
while *q* ≠ *null* **do** ⟨ Process node-or-noad *q* as much as possible in preparation for the second pass of *mlist_to_hlist*, then move to the next item in the mlist 727 ⟩;
⟨ Convert a final *bin_noad* to an *ord_noad* 729 ⟩;
⟨ Make a second pass over the mlist, removing all noads and inserting the proper spacing and penalties 760 ⟩;
end;

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

⟨ Process node-or-noad *q* as much as possible in preparation for the second pass of *mlist_to_hlist*, then move to the next item in the mlist 727 ⟩ ≡

begin ⟨ Do first-pass processing based on *type*(*q*); **goto** *done_with_noad* if a noad has been fully processed, **goto** *check_dimensions* if it has been translated into *new_hlist*(*q*), or **goto** *done_with_node* if a node has been fully processed 728 ⟩;
check_dimensions: $z \leftarrow$ *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* ← *q*; *r_type* ← *type*(*r*);
done_with_node: *q* ← *link*(*q*);
end

This code is used in section 726.

728. 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 728 ⟩ ≡

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* 729 ⟩;
if *type*(*q*) = *right_noad* **then goto** *done_with_noad*;
end;
⟨ Cases for noads that can follow a *bin_noad* 733 ⟩
⟨ Cases for nodes that can appear in an mlist, after which we **goto** *done_with_node* 730 ⟩
othercases *confusion*("mlist1")
endcases;
⟨ Convert *nucleus*(*q*) to an hlist and attach the sub/superscripts 754 ⟩

This code is used in section 727.

729. ⟨ Convert a final *bin_noad* to an *ord_noad* 729 ⟩ ≡
if *r_type* = *bin_noad* **then** *type*(*r*) ← *ord_noad*

This code is used in sections 726 and 728.

730. ⟨ Cases for nodes that can appear in an mlist, after which we **goto** *done_with_node* 730 ⟩ ≡
style_node: **begin** *cur_style* ← *subtype*(*q*);
⟨ Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 703 ⟩;
goto *done_with_node*;
end;
choice_node: ⟨ Change this node to a style node followed by the correct choice, then **goto** *done_with_node* 731 ⟩;
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 732 ⟩;
goto *done_with_node*;
end;
kern_node: **begin** *math_kern*(*q*, *cur_mu*); **goto** *done_with_node*;
end;

This code is used in section 728.

731. **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* 731 ⟩ ≡
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 730.

732. 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 732 ⟩ ≡
  if *subtype*(*q*) = *mu_glue* **then**
    begin *x* ← *glue_ptr*(*q*); *y* ← *math_glue*(*x*, *cur_mu*); *delete_glue_ref*(*x*); *glue_ptr*(*q*) ← *y*;
    subtype(*q*) ← *normal*;
    end
  else if (*cur_size* ≠ *text_size*) ∧ (*subtype*(*q*) = *cond_math_glue*) **then**
      begin *p* ← *link*(*q*);
      if *p* ≠ *null* **then**
        if (*type*(*p*) = *glue_node*) ∨ (*type*(*p*) = *kern_node*) **then**
          begin *link*(*q*) ← *link*(*p*); *link*(*p*) ← *null*; *flush_node_list*(*p*);
          end;
      end

This code is used in section 730.

733. ⟨ Cases for noads that can follow a *bin_noad* 733 ⟩ ≡
left_noad: **goto** *done_with_noad*;
fraction_noad: **begin** *make_fraction*(*q*); **goto** *check_dimensions*;
  end;
op_noad: **begin** *delta* ← *make_op*(*q*);
  if *subtype*(*q*) = *limits* **then goto** *check_dimensions*;
  end;
ord_noad: *make_ord*(*q*);
open_noad, *inner_noad*: *do_nothing*;
radical_noad: *make_radical*(*q*);
over_noad: *make_over*(*q*);
under_noad: *make_under*(*q*);
accent_noad: *make_math_accent*(*q*);
vcenter_noad: *make_vcenter*(*q*);

This code is used in section 728.

734. 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 734 ⟩ ≡
procedure *make_over*(*q* : *pointer*);
  begin *info*(*nucleus*(*q*)) ← *overbar*(*clean_box*(*nucleus*(*q*), *cramped_style*(*cur_style*)),
      3 ∗ *default_rule_thickness*, *default_rule_thickness*); *math_type*(*nucleus*(*q*)) ← *sub_box*;
  end;

See also sections 735, 736, 737, 738, 743, 749, 752, 756, and 762.

This code is used in section 726.

735. ⟨ Declare math construction procedures 734 ⟩ +≡
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;

736. ⟨ Declare math construction procedures 734 ⟩ +≡
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;

737. 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 734 ⟩ +≡
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;

738. 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 734 ⟩ +≡
procedure *make_math_accent*(*q* : *pointer*);
  label *done*, *done1*;
  var *p*, *x*, *y*: *pointer*; { temporary registers for box construction }
    a: *integer*; { address of lig/kern instruction }
    c: *quarterword*; { accent character }
    f: *internal_font_number*; { its font }
    i: *four_quarters*; { its *char_info* }
    s: *scaled*; { amount to skew the accent to the right }
    h: *scaled*; { height of character being accented }
    delta: *scaled*; { space to remove between accent and accentee }
    w: *scaled*; { width of the accentee, not including sub/superscripts }
  begin *fetch*(*accent_chr*(*q*));
  if *char_exists*(*cur_i*) **then**
    begin *i* ← *cur_i*; *c* ← *cur_c*; *f* ← *cur_f*;
    ⟨ Compute the amount of skew 741 ⟩;
    x ← *clean_box*(*nucleus*(*q*), *cramped_style*(*cur_style*)); *w* ← *width*(*x*); *h* ← *height*(*x*);
    ⟨ Switch to a larger accent if available and appropriate 740 ⟩;
    if *h* < *x_height*(*f*) **then** *delta* ← *h* **else** *delta* ← *x_height*(*f*);
    if (*math_type*(*supscr*(*q*)) ≠ *empty*) ∨ (*math_type*(*subscr*(*q*)) ≠ *empty*) **then**
      if *math_type*(*nucleus*(*q*)) = *math_char* **then** ⟨ Swap the subscript and superscript into box *x* 742 ⟩;
    y ← *char_box*(*f*, *c*); *shift_amount*(*y*) ← *s* + *half*(*w* − *width*(*y*)); *width*(*y*) ← 0; *p* ← *new_kern*(−*delta*);
    link(*p*) ← *x*; *link*(*y*) ← *p*; *y* ← *vpack*(*y*, *natural*); *width*(*y*) ← *width*(*x*);
    if *height*(*y*) < *h* **then** ⟨ Make the height of box *y* equal to *h* 739 ⟩;
    info(*nucleus*(*q*)) ← *y*; *math_type*(*nucleus*(*q*)) ← *sub_box*;
    end;
  end;

739. ⟨ Make the height of box *y* equal to *h* 739 ⟩ ≡
  begin *p* ← *new_kern*(*h* − *height*(*y*)); *link*(*p*) ← *list_ptr*(*y*); *list_ptr*(*y*) ← *p*; *height*(*y*) ← *h*;
  end

This code is used in section 738.

740. ⟨ Switch to a larger accent if available and appropriate 740 ⟩ ≡
  loop begin if *char_tag*(*i*) ≠ *list_tag* **then goto** *done*;
    y ← *rem_byte*(*i*); *i* ← *char_info*(*f*)(*y*);
    if ¬*char_exists*(*i*) **then goto** *done*;
    if *char_width*(*f*)(*i*) > *w* **then goto** *done*;
    c ← *y*;
    end;
done:

This code is used in section 738.

741. ⟨Compute the amount of skew 741⟩ ≡

```
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 738.

742. ⟨Swap the subscript and superscript into box x 742⟩ ≡

```
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 738.

743. 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 734⟩ +≡

```
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 744⟩;
  if thickness(q) = 0 then ⟨Adjust shift_up and shift_down for the case of no fraction line 745⟩
  else ⟨Adjust shift_up and shift_down for the case of a fraction line 746⟩;
  ⟨Construct a vlist box for the fraction, according to shift_up and shift_down 747⟩;
  ⟨Put the fraction into a box with its delimiters, and make new_hlist(q) point to it 748⟩;
  end;
```

744. ⟨ 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 744 ⟩ ≡

$x \leftarrow$ *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* ← *num1*(*cur_size*); *shift_down* ← *denom1*(*cur_size*);
  end
else begin *shift_down* ← *denom2*(*cur_size*);
  if *thickness*(q) ≠ 0 **then** *shift_up* ← *num2*(*cur_size*)
  else *shift_up* ← *num3*(*cur_size*);
  end

This code is used in section 743.

745. The numerator and denominator must be separated by a certain minimum clearance, called *clr* in the following program. The difference between *clr* and the actual clearance is twice *delta*.

⟨ Adjust *shift_up* and *shift_down* for the case of no fraction line 745 ⟩ ≡
begin if *cur_style* < *text_style* **then** *clr* ← 7 ∗ *default_rule_thickness*
else *clr* ← 3 ∗ *default_rule_thickness*;
delta ← *half*(*clr* − ((*shift_up* − *depth*(x)) − (*height*(z) − *shift_down*)));
if *delta* > 0 **then**
  begin *shift_up* ← *shift_up* + *delta*; *shift_down* ← *shift_down* + *delta*;
  end;
end

This code is used in section 743.

746. 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 746 ⟩ ≡
begin if *cur_style* < *text_style* **then** *clr* ← 3 ∗ *thickness*(q)
else *clr* ← *thickness*(q);
delta ← *half*(*thickness*(q)); *delta1* ← *clr* − ((*shift_up* − *depth*(x)) − (*axis_height*(*cur_size*) + *delta*));
delta2 ← *clr* − ((*axis_height*(*cur_size*) − *delta*) − (*height*(z) − *shift_down*));
if *delta1* > 0 **then** *shift_up* ← *shift_up* + *delta1*;
if *delta2* > 0 **then** *shift_down* ← *shift_down* + *delta2*;
end

This code is used in section 743.

747. ⟨ Construct a vlist box for the fraction, according to *shift_up* and *shift_down* 747 ⟩ ≡
$v \leftarrow$ *new_null_box*; *type*(v) ← *vlist_node*; *height*(v) ← *shift_up* + *height*(x);
depth(v) ← *depth*(z) + *shift_down*; *width*(v) ← *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) ← 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) ← p; *link*(p) ← z;
  $p \leftarrow$ *new_kern*((*shift_up* − *depth*(x)) − (*axis_height*(*cur_size*) + *delta*)); *link*(p) ← y;
  end;
link(x) ← p; *list_ptr*(v) ← x

This code is used in section 743.

748. ⟨Put the fraction into a box with its delimiters, and make *new_hlist*(*q*) point to it 748⟩ ≡

```
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 743.

749. 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 734⟩ +≡

```
function make_op(q : pointer): scaled;
  var delta: scaled; { offset between subscript and superscript }
    p, v, x, y, z: pointer; { temporary registers for box construction }
    c: quarterword; i: four_quarters; { registers for character examination }
    shift_up, shift_down: scaled; { dimensions for box calculation }
  begin if (subtype(q) = normal) ∧ (cur_style < text_style) then subtype(q) ← limits;
  if math_type(nucleus(q)) = math_char then
    begin fetch(nucleus(q));
    if (cur_style < text_style) ∧ (char_tag(cur_i) = list_tag) then { make it larger }
      begin c ← rem_byte(cur_i); i ← char_info(cur_f)(c);
      if char_exists(i) then
        begin cur_c ← c; cur_i ← i; character(nucleus(q)) ← c;
        end;
      end;
    delta ← char_italic(cur_f)(cur_i); x ← clean_box(nucleus(q), cur_style);
    if (math_type(subscr(q)) ≠ empty) ∧ (subtype(q) ≠ limits) then width(x) ← width(x) − delta;
          { remove italic correction }
    shift_amount(x) ← half(height(x) − depth(x)) − axis_height(cur_size); { center vertically }
    math_type(nucleus(q)) ← sub_box; info(nucleus(q)) ← x;
    end
  else delta ← 0;
  if subtype(q) = limits then ⟨Construct a box with limits above and below it, skewed by delta 750⟩;
  make_op ← delta;
  end;
```

750. 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* 750⟩ ≡
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) ← *vlist_node*;
width(v) ← *width*(y);
if *width*(x) > *width*(v) **then** *width*(v) ← *width*(x);
if *width*(z) > *width*(v) **then** *width*(v) ← *width*(z);
$x \leftarrow$ *rebox*(x, *width*(v)); $y \leftarrow$ *rebox*(y, *width*(v)); $z \leftarrow$ *rebox*(z, *width*(v));
shift_amount(x) ← *half*(*delta*); *shift_amount*(z) ← −*shift_amount*(x); *height*(v) ← *height*(y);
depth(v) ← *depth*(y);
⟨Attach the limits to y and adjust *height*(v), *depth*(v) to account for their presence 751⟩;
new_hlist(q) ← v;
end

This code is used in section 749.

751. 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 751⟩ ≡
if *math_type*(*supscr*(q)) = *empty* **then**
  begin *free_node*(x, *box_node_size*); *list_ptr*(v) ← y;
  end
else begin *shift_up* ← *big_op_spacing3* − *depth*(x);
  if *shift_up* < *big_op_spacing1* **then** *shift_up* ← *big_op_spacing1*;
  $p \leftarrow$ *new_kern*(*shift_up*); *link*(p) ← y; *link*(x) ← p;
  $p \leftarrow$ *new_kern*(*big_op_spacing5*); *link*(p) ← x; *list_ptr*(v) ← p;
  height(v) ← *height*(v) + *big_op_spacing5* + *height*(x) + *depth*(x) + *shift_up*;
  end;
if *math_type*(*subscr*(q)) = *empty* **then** *free_node*(z, *box_node_size*)
else begin *shift_down* ← *big_op_spacing4* − *height*(z);
  if *shift_down* < *big_op_spacing2* **then** *shift_down* ← *big_op_spacing2*;
  $p \leftarrow$ *new_kern*(*shift_down*); *link*(y) ← p; *link*(p) ← z;
  $p \leftarrow$ *new_kern*(*big_op_spacing5*); *link*(z) ← p;
  depth(v) ← *depth*(v) + *big_op_spacing5* + *height*(z) + *depth*(z) + *shift_down*;
  end

This code is used in section 750.

752. 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 734⟩ +≡
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 753⟩;
                    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;
```

753. 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* 753⟩ ≡

```
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 752.

754. 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 754 ⟩ ≡
  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 755 ⟩;
  empty: $p \leftarrow$ *null*;
  sub_box: $p \leftarrow$ *info*(*nucleus*(q));
  sub_mlist: **begin** *cur_mlist* ← *info*(*nucleus*(q)); *save_style* ← *cur_style*; *mlist_penalties* ← *false*;
    mlist_to_hlist; { recursive call }
    cur_style ← *save_style*; ⟨ Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 703 ⟩;
    $p \leftarrow$ *hpack*(*link*(*temp_head*), *natural*);
    end;
  othercases *confusion*("mlist2")
  endcases;
  new_hlist(q) ← p;
  if (*math_type*(*subscr*(q)) = *empty*) ∧ (*math_type*(*supscr*(q)) = *empty*) **then goto** *check_dimensions*;
  make_scripts(q, *delta*)

This code is used in section 728.

755. ⟨ 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 755 ⟩ ≡
  begin *fetch*(*nucleus*(q));
  if *char_exists*(*cur_i*) **then**
    begin *delta* ← *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*) ≠ 0) **then** *delta* ← 0;
        { no italic correction in mid-word of text font }
    if (*math_type*(*subscr*(q)) = *empty*) ∧ (*delta* ≠ 0) **then**
      begin *link*(p) ← *new_kern*(*delta*); *delta* ← 0;
      end;
    end
  else $p \leftarrow$ *null*;
  end

This code is used in section 754.

756. 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 the subscript and superscript aren't both empty. The superscript will appear to the right of the subscript by a given distance *delta*.

We set *shift_down* and *shift_up* to the minimum amounts to shift the baseline of subscripts and superscripts based on the given nucleus.

```
⟨Declare math construction procedures 734⟩ +≡
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 757⟩
  else begin ⟨Construct a superscript box x 758⟩;
    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 759⟩;
    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;
```

757. 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 757⟩ ≡
  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 756.

758. The bottom of a superscript should never descend below the baseline plus one-fourth of the x-height.

⟨ Construct a superscript box *x* 758 ⟩ ≡
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 756.

759. 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* 759 ⟩ ≡
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 756.

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

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

This code is used in section 726.

761. 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 761⟩ ≡

```
t ← ord_noad; s ← noad_size; pen ← inf_penalty;
case type(q) of
op_noad, open_noad, close_noad, punct_noad, inner_noad: t ← type(q);
bin_noad: begin t ← bin_noad; pen ← bin_op_penalty;
  end;
rel_noad: begin t ← rel_noad; pen ← rel_penalty;
  end;
ord_noad, vcenter_noad, over_noad, under_noad: do_nothing;
radical_noad: s ← radical_noad_size;
accent_noad: s ← accent_noad_size;
fraction_noad: s ← fraction_noad_size;
left_noad, right_noad: t ← make_left_right(q, style, max_d, max_h);
style_node: ⟨Change the current style and goto delete_q 763⟩;
whatsit_node, penalty_node, rule_node, disc_node, adjust_node, ins_node, mark_node, glue_node, kern_node:
  begin link(p) ← q; p ← q; q ← link(q); link(p) ← null; goto done;
  end;
othercases confusion("mlist3")
endcases
```

This code is used in section 760.

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

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

⟨Declare math construction procedures 734⟩ +≡

```
function make_left_right(q : pointer; style : small_number; max_d, max_h : scaled): small_number;
  var delta, delta1, delta2: scaled; { dimensions used in the calculation }
  begin if style < script_style then cur_size ← text_size
  else cur_size ← 16 ∗ ((style − text_style) div 2);
  delta2 ← max_d + axis_height(cur_size); delta1 ← max_h + max_d − delta2;
  if delta2 > delta1 then delta1 ← delta2; { delta1 is max distance from axis }
  delta ← (delta1 div 500) ∗ delimiter_factor; delta2 ← delta1 + delta1 − delimiter_shortfall;
  if delta < delta2 then delta ← delta2;
  new_hlist(q) ← var_delimiter(delimiter(q), cur_size, delta);
  make_left_right ← type(q) − (left_noad − open_noad); { open_noad or close_noad }
  end;
```

763. ⟨Change the current style and **goto** *delete_q* 763⟩ ≡

```
begin cur_style ← subtype(q); s ← style_node_size;
⟨Set up the values of cur_size and cur_mu, based on cur_style 703⟩;
goto delete_q;
end
```

This code is used in section 761.

764. The inter-element spacing in math formulas depends on an 8×8 table that TEX preloads as a 64-digit string. The elements of this string have the following significance:

`0` means no space;
`1` means a conditional thin space (`\nonscript\mskip\thinmuskip`);
`2` means a thin space (`\mskip\thinmuskip`);
`3` means a conditional medium space (`\nonscript\mskip\medmuskip`);
`4` means a conditional thick space (`\nonscript\mskip\thickmuskip`);
`*` means an impossible case.

This is all pretty cryptic, but *The TEXbook* explains what is supposed to happen, and the string makes it happen.

A global variable *magic_offset* is computed so that if a and b are in the range *ord_noad* .. *inner_noad*, then *str_pool*$[a * 8 + b + magic\_offset]$ is the digit for spacing between noad types a and b.

If Pascal had provided a good way to preload constant arrays, this part of the program would not have been so strange.

define *math_spacing* =
`"0234000122*4000133**3**344*0400400*000000234000111*1111112341011"`

⟨ Global variables 13 ⟩ +≡
magic_offset: *integer*; { used to find inter-element spacing }

765. ⟨ Compute the magic offset 765 ⟩ ≡
magic_offset ← *str_start*[*math_spacing*] − 9 ∗ *ord_noad*

This code is used in section 1337.

766. ⟨ Append inter-element spacing based on *r_type* and *t* 766 ⟩ ≡
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 760.

767. 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 767 ⟩ ≡
```
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 760.

768. 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.

769. 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 }

770. Alignments can occur within alignments, so a small stack is used to access the alignrecord information. At each level we have a *preamble* pointer, indicating the beginning of the preamble list; a *cur_align* pointer, indicating the current position in the preamble list; a *cur_span* pointer, indicating the value of *cur_align* at the beginning of a sequence of spanned columns; a *cur_loop* pointer, indicating the tabskip glue before an alignrecord that should be copied next if the current list is extended; and the *align_state* variable, which indicates the nesting of braces so that `\cr` and `\span` and tab marks are properly intercepted. There also are pointers *cur_head* and *cur_tail* to the head and tail of a list of adjustments being moved out from horizontal mode to vertical mode.

The current values of these seven quantities appear in global variables; when they have to be pushed down, they are stored in 5-word nodes, and *align_ptr* points to the topmost such node.

define *preamble* ≡ *link*(*align_head*) { the current preamble list }
define *align_stack_node_size* = 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 }

771. 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*;

772. 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;

773. 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*.

774. 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* 782 ⟩
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 776 ⟩;
  push_nest; { enter a new semantic level }
  ⟨ Change current mode to −*vmode* for `\halign`, −*hmode* for `\valign` 775 ⟩;
  scan_spec(*align_group*, *false*);
  ⟨ Scan the preamble and record it in the *preamble* list 777 ⟩;
  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;

775. 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` 775 ⟩ ≡
  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 774.

776. When `\halign` is used as a displayed formula, there should be no other pieces of mlists present.

⟨ Check for improper alignment in displayed math 776 ⟩ ≡
  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 774.

777. ⟨ Scan the preamble and record it in the *preamble* list 777 ⟩ ≡

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 778 ⟩;
  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 779 ⟩;
  end;
done: *scanner_status* ← *normal*

This code is used in section 774.

778. ⟨ Append the current tabskip glue to the preamble list 778 ⟩ ≡

link(*cur_align*) ← *new_param_glue*(*tab_skip_code*); *cur_align* ← *link*(*cur_align*)

This code is used in section 777.

779. ⟨ 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 779 ⟩ ≡

⟨ Scan the template ⟨u_j⟩, putting the resulting token list in *hold_head* 783 ⟩;
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 ⟨v_j⟩, putting the resulting token list in *hold_head* 784 ⟩;
v_part(*cur_align*) ← *link*(*hold_head*)

This code is used in section 777.

780. 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*;

781. ⟨ 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`");

782. 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 782⟩ ≡
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 774.

783. Spaces are eliminated from the beginning of a template.

```
⟨Scan the template ⟨u_j⟩, putting the resulting token list in hold_head 783⟩ ≡
  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 779.

784. ⟨ Scan the template ⟨v_j⟩, putting the resulting token list in *hold_head* 784 ⟩ ≡

```
p ← hold_head; link(p) ← null;
loop begin continue: get_preamble_token;
  if (cur_cmd ≤ car_ret) ∧ (cur_cmd ≥ tab_mark) ∧ (align_state = −1000000) then goto done2;
  if cur_cmd = mac_param then
    begin print_err("Only␣one␣#␣is␣allowed␣per␣tab");
    help3("There␣should␣be␣exactly␣one␣#␣between␣&´s,␣when␣an")
    ("\halign␣or␣\valign␣is␣being␣set␣up.␣In␣this␣case␣you␣had")
    ("more␣than␣one,␣so␣I´m␣ignoring␣all␣but␣the␣first."); error; goto continue;
    end;
  link(p) ← get_avail; p ← link(p); info(p) ← cur_tok;
  end;
done2: link(p) ← get_avail; p ← link(p); info(p) ← end_template_token  { put \endtemplate at the end }
```

This code is used in section 779.

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

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

⟨ Declare the procedure called *align_peek* 785 ⟩ ≡

```
procedure align_peek;
  label restart;
  begin restart: align_state ← 1000000; ⟨ Get the next non-blank non-call token 406 ⟩;
  if cur_cmd = no_align then
    begin scan_left_brace; new_save_level(no_align_group);
    if mode = −vmode then normal_paragraph;
    end
  else if cur_cmd = right_brace then fin_align
    else if (cur_cmd = car_ret) ∧ (cur_chr = cr_cr_code) then goto restart  { ignore \crcr }
      else begin init_row;  { start a new row }
        init_col;  { start a new column and replace what we peeked at }
        end;
  end;
```

This code is used in section 800.

786. 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* 787 ⟩

```
procedure init_row;
  begin push_nest; mode ← (−hmode − vmode) − mode;
  if mode = −hmode then space_factor ← 0 else prev_depth ← 0;
  tail_append(new_glue(glue_ptr(preamble))); subtype(tail) ← tab_skip_code + 1;
  cur_align ← link(preamble); cur_tail ← cur_head; init_span(cur_align);
  end;
```

787. 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* 787 ⟩ ≡
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 786.

788. 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;

789. 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* 789 ⟩ ≡
  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.

790. 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 790 ⟩ ≡
  info(*omit_template*) ← *end_template_token*; { *link*(*omit_template*) = *null* }

See also sections 797, 820, 981, and 988.

This code is used in section 164.

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

function *fin_col*: *boolean*;
  label *exit*;
  var *p*: *pointer*; { the alignrecord after the current one }
    q, *r*: *pointer*; { temporary pointers for list manipulation }
    s: *pointer*; { a new span node }
    u: *pointer*; { a new unset box }
    w: *scaled*; { natural width }
    o: *glue_ord*; { order of infinity }
    n: *halfword*; { span counter }
  begin if *cur_align* = *null* **then** *confusion*("endv");
  q ← *link*(*cur_align*); **if** *q* = *null* **then** *confusion*("endv");
  if *align_state* < 500000 **then** *fatal_error*("(interwoven␣alignment␣preambles␣are␣not␣allowed)");
  p ← *link*(*q*); ⟨If the preamble list has been traversed, check that the row has ended 792⟩;
  if *extra_info*(*cur_align*) ≠ *span_code* **then**
    begin *unsave*; *new_save_level*(*align_group*);
    ⟨Package an unset box for the current column and record its width 796⟩;
    ⟨Copy the tabskip glue between columns 795⟩;
    if *extra_info*(*cur_align*) ≥ *cr_code* **then**
      begin *fin_col* ← *true*; **return**;
      end;
    init_span(*p*);
    end;
  align_state ← 1000000; ⟨Get the next non-blank non-call token 406⟩;
  cur_align ← *p*; *init_col*; *fin_col* ← *false*;
exit: **end**;

792. ⟨If the preamble list has been traversed, check that the row has ended 792⟩ ≡
  if (*p* = *null*) ∧ (*extra_info*(*cur_align*) < *cr_code*) **then**
    if *cur_loop* ≠ *null* **then** ⟨Lengthen the preamble periodically 793⟩
    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 791.

793. ⟨Lengthen the preamble periodically 793⟩ ≡
  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* 794⟩;
  cur_loop ← *link*(*cur_loop*); *link*(*p*) ← *new_glue*(*glue_ptr*(*cur_loop*)); *subtype*(*link*(*p*)) ← *tab_skip_code* + 1;
  end

This code is used in section 792.

794. ⟨ Copy the templates from node *cur_loop* into node p 794 ⟩ ≡

$q \leftarrow$ *hold_head*; $r \leftarrow$ *u_part*(*cur_loop*);
while $r \neq$ *null* **do**
  begin *link*(q) ← *get_avail*; $q \leftarrow$ *link*(q); *info*(q) ← *info*(r); $r \leftarrow$ *link*(r);
  end;
link(q) ← *null*; *u_part*(p) ← *link*(*hold_head*); $q \leftarrow$ *hold_head*; $r \leftarrow$ *v_part*(*cur_loop*);
while $r \neq$ *null* **do**
  begin *link*(q) ← *get_avail*; $q \leftarrow$ *link*(q); *info*(q) ← *info*(r); $r \leftarrow$ *link*(r);
  end;
link(q) ← *null*; *v_part*(p) ← *link*(*hold_head*)

This code is used in section 793.

795. ⟨ Copy the tabskip glue between columns 795 ⟩ ≡

tail_append(*new_glue*(*glue_ptr*(*link*(*cur_align*)))); *subtype*(*tail*) ← *tab_skip_code* + 1

This code is used in section 791.

796. ⟨ Package an unset box for the current column and record its width 796 ⟩ ≡

begin if *mode* = −*hmode* **then**
  begin *adjust_tail* ← *cur_tail*; $u \leftarrow$ *hpack*(*link*(*head*), *natural*); $w \leftarrow$ *width*(u); *cur_tail* ← *adjust_tail*;
  adjust_tail ← *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* ≠ *cur_align* **then** ⟨ Update width entry for spanned columns 798 ⟩
else if $w >$ *width*(*cur_align*) **then** *width*(*cur_align*) ← w;
type(u) ← *unset_node*; *span_count*(u) ← n;
⟨ Determine the stretch order 659 ⟩;
glue_order(u) ← o; *glue_stretch*(u) ← *total_stretch*[o];
⟨ Determine the shrink order 665 ⟩;
glue_sign(u) ← o; *glue_shrink*(u) ← *total_shrink*[o];
pop_nest; *link*(*tail*) ← u; *tail* ← u;
end

This code is used in section 791.

797. 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 790 ⟩ +≡
link(*end_span*) ← *max_quarterword* + 1; *info*(*end_span*) ← *null*;

798. ⟨Update width entry for spanned columns 798⟩ ≡
begin $q \gets cur\_span$;
repeat $incr(n)$; $q \gets link(link(q))$;
until $q = cur\_align$;
if $n > max\_quarterword$ **then** *confusion*("256␣spans"); { this can happen, but won't }
$q \gets cur\_span$;
while $link(info(q)) < n$ **do** $q \gets info(q)$;
if $link(info(q)) > n$ **then**
begin $s \gets get\_node(span\_node\_size)$; $info(s) \gets info(q)$; $link(s) \gets n$; $info(q) \gets s$; $width(s) \gets w$;
end
else if $width(info(q)) < w$ **then** $width(info(q)) \gets w$;
end

This code is used in section 796.

799. 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 $p \gets hpack(link(head), natural)$; *pop_nest*; *append_to_vlist*(*p*);
if $cur\_head \neq cur\_tail$ **then**
begin $link(tail) \gets link(cur\_head)$; $tail \gets cur\_tail$;
end;
end
else begin $p \gets vpack(link(head), natural)$; *pop_nest*; $link(tail) \gets p$; $tail \gets p$; $space\_factor \gets 1000$;
end;
$type(p) \gets unset\_node$; $glue\_stretch(p) \gets 0$;
if $every\_cr \neq null$ **then** *begin_token_list*(*every_cr*, *every_cr_text*);
align_peek;
end; { note that $glue\_shrink(p) = 0$ since $glue\_shrink \equiv shift\_amount$ }

800. Finally, we will reach the end of the alignment, and we can breathe a sigh of relief that memory hasn't overflowed. All the unset boxes will now be set so that the columns line up, taking due account of spanned columns.

procedure *do_assignments*; *forward*;
procedure *resume_after_display*; *forward*;
procedure *build_page*; *forward*;
procedure *fin_align*;
  var *p*, *q*, *r*, *s*, *u*, *v*: *pointer*; { registers for the list operations }
    t, *w*: *scaled*; { width of column }
    o: *scaled*; { shift offset for unset boxes }
    n: *halfword*; { matching span amount }
    rule_save: *scaled*; { temporary storage for *overfull_rule* }
    aux_save: *memory_word*; { temporary storage for *aux* }
  begin if *cur_group* ≠ *align_group* **then** *confusion*("align1");
  unsave; { that *align_group* was for individual entries }
  if *cur_group* ≠ *align_group* **then** *confusion*("align0");
  unsave; { that *align_group* was for the whole alignment }
  if *nest*[*nest_ptr* − 1].*mode_field* = *mmode* **then** *o* ← *display_indent*
  else *o* ← 0;
  ⟨ Go through the preamble list, determining the column widths and changing the alignrecords to dummy unset boxes 801 ⟩;
  ⟨ Package the preamble list, to determine the actual tabskip glue amounts, and let *p* point to this prototype box 804 ⟩;
  ⟨ Set the glue in all the unset boxes of the current list 805 ⟩;
  flush_node_list(*p*); *pop_alignment*; ⟨ Insert the current list into its environment 812 ⟩;
  end;
⟨ Declare the procedure called *align_peek* 785 ⟩

801. 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 801 ⟩ ≡

```
q ← link(preamble);
repeat flush_list(u_part(q)); flush_list(v_part(q)); p ← link(link(q));
  if width(q) = null_flag then ⟨ Nullify width(q) and the tabskip glue following this column 802 ⟩;
  if info(q) ≠ end_span then
    ⟨ Merge the widths in the span nodes of q with those of p, destroying the span nodes of q 803 ⟩;
  type(q) ← unset_node; span_count(q) ← min_quarterword; height(q) ← 0; depth(q) ← 0;
  glue_order(q) ← normal; glue_sign(q) ← normal; glue_stretch(q) ← 0; glue_shrink(q) ← 0; q ← p;
until q = null
```

This code is used in section 800.

802. ⟨ Nullify *width*(*q*) and the tabskip glue following this column 802 ⟩ ≡

```
begin width(q) ← 0; r ← link(q); s ← glue_ptr(r);
if s ≠ zero_glue then
  begin add_glue_ref(zero_glue); delete_glue_ref(s); glue_ptr(r) ← zero_glue;
  end;
end
```

This code is used in section 801.

803. 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 803⟩ ≡
  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 801.

804. 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 804⟩ ≡
  $save\_ptr \leftarrow save\_ptr - 2$; $pack\_begin\_line \leftarrow -mode\_line$;
  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 800.

805. ⟨ Set the glue in all the unset boxes of the current list 805 ⟩ ≡

$q \leftarrow link(head)$; $s \leftarrow head$;
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 807 ⟩
    else if $type(q) = rule\_node$ **then**
      ⟨ Make the running dimensions in rule q extend to the boundaries of the alignment 806 ⟩;
  $s \leftarrow q$; $q \leftarrow link(q)$;
  end

This code is used in section 800.

806. ⟨ Make the running dimensions in rule q extend to the boundaries of the alignment 806 ⟩ ≡

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 805.

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

⟨ Set the unset box q and the unset boxes in it 807 ⟩ ≡

begin if $mode = -vmode$ **then**
  begin $type(q) \leftarrow hlist\_node$; $width(q) \leftarrow width(p)$;
  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 808 ⟩;
  $r \leftarrow link(link(r))$; $s \leftarrow link(link(s))$;
until $r = null$;
end

This code is used in section 805.

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

⟨Set the glue in node r and change it from an unset node 808⟩ ≡

```
n ← span_count(r); t ← width(s); w ← t; u ← 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 809⟩;
  end;
if mode = −vmode then
  ⟨Make the unset node r into an hlist_node of width w, setting the glue as if the width were t 810⟩
else ⟨Make the unset node r into a vlist_node of height w, setting the glue as if the height were t 811⟩;
shift_amount(r) ← 0;
if u ≠ hold_head then  { append blank boxes to account for spanned nodes }
  begin link(u) ← link(r); link(r) ← link(hold_head); r ← u;
  end
```

This code is used in section 807.

809. ⟨Append tabskip glue and an empty box to list u, and update s and t as the prototype nodes are passed 809⟩ ≡

```
s ← link(s); v ← glue_ptr(s); link(u) ← new_glue(v); u ← link(u); subtype(u) ← tab_skip_code + 1;
t ← t + width(v);
if glue_sign(p) = stretching then
  begin if stretch_order(v) = glue_order(p) then t ← t + round(float(glue_set(p)) ∗ stretch(v));
  end
else if glue_sign(p) = shrinking then
    begin if shrink_order(v) = glue_order(p) then t ← t − round(float(glue_set(p)) ∗ shrink(v));
    end;
s ← link(s); link(u) ← new_null_box; u ← link(u); t ← t + width(s);
if mode = −vmode then width(u) ← width(s) else begin type(u) ← vlist_node; height(u) ← width(s);
  end
```

This code is used in section 808.

810. ⟨Make the unset node r into an *hlist_node* of width w, setting the glue as if the width were t 810⟩ ≡

```
begin height(r) ← height(q); depth(r) ← depth(q);
if t = width(r) then
  begin glue_sign(r) ← normal; glue_order(r) ← normal; set_glue_ratio_zero(glue_set(r));
  end
else if t > width(r) then
    begin glue_sign(r) ← stretching;
    if glue_stretch(r) = 0 then set_glue_ratio_zero(glue_set(r))
    else glue_set(r) ← unfloat((t − width(r))/glue_stretch(r));
    end
  else begin glue_order(r) ← glue_sign(r); glue_sign(r) ← shrinking;
    if glue_shrink(r) = 0 then set_glue_ratio_zero(glue_set(r))
    else if (glue_order(r) = normal) ∧ (width(r) − t > glue_shrink(r)) then
        set_glue_ratio_one(glue_set(r))
      else glue_set(r) ← unfloat((width(r) − t)/glue_shrink(r));
    end;
width(r) ← w; type(r) ← hlist_node;
end
```

This code is used in section 808.

811. ⟨Make the unset node r into a *vlist_node* of height w, setting the glue as if the height were t 811⟩ ≡

```
begin width(r) ← width(q);
if t = height(r) then
  begin glue_sign(r) ← normal; glue_order(r) ← normal; set_glue_ratio_zero(glue_set(r));
  end
else if t > height(r) then
    begin glue_sign(r) ← stretching;
    if glue_stretch(r) = 0 then set_glue_ratio_zero(glue_set(r))
    else glue_set(r) ← unfloat((t − height(r))/glue_stretch(r));
    end
  else begin glue_order(r) ← glue_sign(r); glue_sign(r) ← shrinking;
    if glue_shrink(r) = 0 then set_glue_ratio_zero(glue_set(r))
    else if (glue_order(r) = normal) ∧ (height(r) − t > glue_shrink(r)) then
        set_glue_ratio_one(glue_set(r))
      else glue_set(r) ← unfloat((height(r) − t)/glue_shrink(r));
    end;
height(r) ← w; type(r) ← vlist_node;
end
```

This code is used in section 808.

812. 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 812⟩ ≡

```
aux_save ← aux; p ← link(head); q ← tail; pop_nest;
if mode = mmode then ⟨Finish an alignment in a display 1206⟩
else begin aux ← aux_save; link(tail) ← p;
  if p ≠ null then tail ← q;
  if mode = vmode then build_page;
  end
```

This code is used in section 800.

813. Breaking paragraphs into lines. We come now to what is probably the most interesting algorithm of TEX: the mechanism for choosing the "best possible" breakpoints that yield the individual lines of a paragraph. TEX's line-breaking algorithm takes a given horizontal list and converts it to a sequence of boxes that are appended to the current vertical list. In the course of doing this, it creates a special data structure containing three kinds of records that are not used elsewhere in TEX. Such nodes are created while a paragraph is being processed, and they are destroyed afterwards; thus, the other parts of TEX do not need to know anything about how line-breaking is done.

The method used here is based on an approach devised by Michael F. Plass and the author in 1977, subsequently generalized and improved by the same two people in 1980. A detailed discussion appears in *Software—Practice and Experience* **11** (1981), 1119–1184, where it is shown that the line-breaking problem can be regarded as a special case of the problem of computing the shortest path in an acyclic network. The cited paper includes numerous examples and describes the history of line breaking as it has been practiced by printers through the ages. The present implementation adds two new ideas to the algorithm of 1980: Memory space requirements are considerably reduced by using smaller records for inactive nodes than for active ones, and arithmetic overflow is avoided by using "delta distances" instead of keeping track of the total distance from the beginning of the paragraph to the current point.

814. The *line_break* procedure should be invoked only in horizontal mode; it leaves that mode and places its output into the current vlist of the enclosing vertical mode (or internal vertical mode). There is one explicit parameter: *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` is in use; *prev_graf* is zero unless this paragraph is being continued after a displayed formula. Other implicit parameters, such as the *par_shape_ptr* and various penalties to use for hyphenation, etc., appear in *eqtb*.

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

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

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

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

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

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

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

See also sections 827, 834, and 848.
This code is used in section 815.

817. 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 }

818. 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.

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

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

break_node points to the passive node associated with this breakpoint.

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

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

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

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

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

define *active_node_size* = 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 }

820. ⟨ Initialize the special list heads and constant nodes 790 ⟩ +≡
type(*last_active*) ← *hyphenated*; *line_number*(*last_active*) ← *max_halfword*; *subtype*(*last_active*) ← 0;
{ the *subtype* is never examined by the algorithm }

821. 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 }

822. 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 $mem[q+2 \mathinner{..} q+5].sc$; 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 }

823. 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*(#) ≡ #(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 }

824. 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 $x_1y_1 \ldots x_ny_n$. 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.

825. 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*(#) $\equiv$
  if (*shrink_order*(#) $\neq$ *normal*) $\wedge$ (*shrink*(#) $\neq 0$) **then**
    begin # $\leftarrow$ *finite_shrink*(#);
    end

⟨ Global variables 13 ⟩ +≡
no_shrink_error_yet: *boolean*; { have we complained about infinite shrinkage? }

826. ⟨ Declare subprocedures for *line_break* 826 ⟩ ≡

```
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;
    stat if tracing_paragraphs > 0 then end_diagnostic(true);
    tats print_err("Infinite␣glue␣shrinkage␣found␣in␣a␣paragraph");
    help5("The␣paragraph␣just␣ended␣includes␣some␣glue␣that␣has")
    ("infinite␣shrinkability,␣e.g.,␣`\hskip␣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;
    stat if tracing_paragraphs > 0 then begin_diagnostic;
    tats
    end;
  q ← new_spec(p); shrink_order(q) ← normal; delete_glue_ref(p); finite_shrink ← q;
  end;
```

See also sections 829, 877, 895, and 942.

This code is used in section 815.

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

```
  no_shrink_error_yet ← true;
  check_shrinkage(left_skip); check_shrinkage(right_skip);
  q ← left_skip; r ← right_skip; background[1] ← width(q) + width(r);
  background[2] ← 0; background[3] ← 0; background[4] ← 0; background[5] ← 0;
  background[2 + stretch_order(q)] ← stretch(q);
  background[2 + stretch_order(r)] ← background[2 + stretch_order(r)] + stretch(r);
  background[6] ← shrink(q) + shrink(r);
```

828. 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 }
```

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

The parameter *pi* to *try_break* is the penalty associated with a break at *cur_p*; we have *pi* = *eject_penalty* if the break is forced, and *pi* = *inf_penalty* if the break is illegal.

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

define *copy_to_cur_active*(#) ≡ *cur_active_width*[#] ← *active_width*[#]
define *deactivate* = 60 { go here when node *r* should be deactivated }

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

830. ⟨ Other local variables for *try_break* 830 ⟩ ≡
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 829.

831. ⟨Make sure that *pi* is in the proper range 831⟩ ≡
  if *abs*(*pi*) ≥ *inf_penalty* **then**
    if *pi* > 0 **then return** { this breakpoint is inhibited by infinite penalty }
    else *pi* ← *eject_penalty* { this breakpoint will be forced }

This code is used in section 829.

832. 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* 832⟩ ≡
  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 829.

833. 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 }

834. ⟨Get ready to start line breaking 816⟩ +≡
  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*;

835. 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* 835⟩ ≡
  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 836⟩;
    if *r* = *last_active* **then return**;
    ⟨Compute the new line width 850⟩;
    end;
  end

This code is used in section 829.

836. 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 836⟩ ≡
  begin if *no_break_yet* **then** ⟨Compute the values of *break_width* 837⟩;
  ⟨Insert a delta node to prepare for breaks at *cur_p* 843⟩;
  if *abs*(*adj_demerits*) ≥ *awful_bad* − *minimum_demerits* **then** *minimum_demerits* ← *awful_bad* − 1
  else *minimum_demerits* ← *minimum_demerits* + *abs*(*adj_demerits*);
  for *fit_class* ← *very_loose_fit* **to** *tight_fit* **do**
    begin if *minimal_demerits*[*fit_class*] ≤ *minimum_demerits* **then**
      ⟨Insert a new active node from *best_place*[*fit_class*] to *cur_p* 845⟩;
    minimal_demerits[*fit_class*] ← *awful_bad*;
    end;
  minimum_demerits ← *awful_bad*; ⟨Insert a delta node to prepare for the next active node 844⟩;
  end

This code is used in section 835.

837. 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* 837⟩ ≡
  begin *no_break_yet* ← *false*; *do_all_six*(*set_break_width_to_background*); *s* ← *cur_p*;
  if *break_type* > *unhyphenated* **then**
    if *cur_p* ≠ *null* **then** ⟨Compute the discretionary *break_width* values 840⟩;
  while *s* ≠ *null* **do**
    begin if *is_char_node*(*s*) **then goto** *done*;
    case *type*(*s*) **of**
    glue_node: ⟨Subtract glue from *break_width* 838⟩;
    penalty_node: *do_nothing*;
    math_node: *break_width*[1] ← *break_width*[1] − *width*(*s*);
    kern_node: **if** *subtype*(*s*) ≠ *explicit* **then goto** *done*
      else *break_width*[1] ← *break_width*[1] − *width*(*s*);
    othercases goto *done*
    endcases;
    s ← *link*(*s*);
    end;
done: **end**

This code is used in section 836.

838. ⟨Subtract glue from *break_width* 838⟩ ≡

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 837.

839. 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 }

840. ⟨Compute the discretionary *break_width* values 840⟩ ≡

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* 841⟩;
  end;
while $s \neq null$ **do**
  begin ⟨Add the width of node s to *break_width* 842⟩;
  $s \leftarrow link(s)$;
  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 837.

841. 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* 841⟩ ≡

if $is\_char\_node(v)$ **then**
  begin $f \leftarrow font(v)$; $break\_width[1] \leftarrow break\_width[1] - char\_width(f)(char\_info(f)(character(v)))$;
  end
else case $type(v)$ **of**
  ligature_node: **begin** $f \leftarrow font(lig\_char(v))$;
    $break\_width[1] \leftarrow break\_width[1] - char\_width(f)(char\_info(f)(character(lig\_char(v))))$;
    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 840.

842. ⟨Add the width of node *s* to *break_width* 842⟩ ≡
if *is_char_node*(*s*) **then**
  begin $f \leftarrow font(s)$; $break\_width[1] \leftarrow break\_width[1] + char\_width(f)(char\_info(f)(character(s)))$;
  end
else case *type*(*s*) **of**
  ligature_node: **begin** $f \leftarrow font(lig\_char(s))$;
    $break\_width[1] \leftarrow break\_width[1] + char\_width(f)(char\_info(f)(character(lig\_char(s))))$;
    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 840.

843. 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* 843⟩ ≡
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 836.

844. 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 844⟩ ≡
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 836.

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

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

This code is used in section 836.

846. ⟨Print a symbolic description of the new break node 846⟩ ≡
begin *print_nl*("@@"); *print_int*(*serial*(*passive*)); *print*(":␣line␣"); *print_int*(*line_number*(q) − 1);
print_char("."); *print_int*(*fit_class*);
if *break_type* = *hyphenated* **then** *print_char*("-");
print("␣t="); *print_int*(*total_demerits*(q)); *print*("␣->␣@@");
if *prev_break*(*passive*) = *null* **then** *print_char*("0")
else *print_int*(*serial*(*prev_break*(*passive*)));
end

This code is used in section 845.

847. 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 $n \ge 0$, 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* }

848. 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 816⟩ +≡
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 849⟩
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*

849. ⟨Set line length parameters in preparation for hanging indentation 849⟩ ≡
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 848.

850. When we come to the following code, we have just encountered the first active node *r* whose *line_number* field contains *l*. Thus we want to compute the length of the *l*th line of the current paragraph. Furthermore, we want to set *old_l* to the last number in the class of line numbers equivalent to *l*.

⟨Compute the new line width 850⟩ ≡
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 835.

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

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

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

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

This code is used in section 829.

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

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

⟨Set the value of b to the badness for stretching the line, and compute the corresponding *fit_class* 852⟩ ≡

```
if (cur_active_width[3] ≠ 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 851.

853. 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* 853⟩ ≡
  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 851.

854. 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* 854⟩ ≡
  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 851.

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

⟨Record a new feasible break 855⟩ ≡
  if *artificial_demerits* **then** $d \leftarrow 0$
  else ⟨Compute the demerits, d, from r to *cur_p* 859⟩;
  stat if *tracing_paragraphs* > 0 **then** ⟨Print a symbolic description of this feasible break 856⟩;
  tats
  $d \leftarrow d +$ *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 851.

856. ⟨Print a symbolic description of this feasible break 856⟩ ≡

```
begin if printed_node ≠ cur_p then
  ⟨Print the list between printed_node and cur_p, then set printed_node ← cur_p 857⟩;
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 855.

857. ⟨Print the list between printed_node and cur_p, then set printed_node ← cur_p 857⟩ ≡

```
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 856.

858. 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 858⟩ ≡

```
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 829.

859. ⟨Compute the demerits, d, from r to *cur_p* 859⟩ ≡

```
begin d ← line_penalty + b;
if abs(d) ≥ 10000 then d ← 100000000 else d ← d * d;
if pi ≠ 0 then
  if pi > 0 then d ← d + pi * pi
  else if pi > eject_penalty then d ← d − pi * pi;
if (break_type = hyphenated) ∧ (type(r) = hyphenated) then
  if cur_p ≠ null then d ← d + double_hyphen_demerits
  else d ← d + final_hyphen_demerits;
if abs(fit_class − fitness(r)) > 1 then d ← d + adj_demerits;
end
```

This code is used in section 855.

860. 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 860⟩ ≡

```
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 861⟩
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 851.

861. 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 861⟩ ≡

```
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 860.

862. 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 862⟩ ≡
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 893.

This code is used in section 815.

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

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

This code is used in section 815.

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

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

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

This code is used in section 863.

865. ⟨Clean up the memory by removing the break nodes 865⟩ ≡

```
q ← link(active);
while q ≠ last_active do
  begin cur_p ← link(q);
  if type(q) = delta_node then free_node(q, delta_node_size)
  else free_node(q, active_node_size);
  q ← cur_p;
  end;
q ← passive;
while q ≠ null do
  begin cur_p ← link(q); free_node(q, passive_node_size); q ← cur_p;
  end
```

This code is used in sections 815 and 863.

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

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

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

```
begin if is_char_node(cur_p) then
  ⟨Advance cur_p to the node following the present string of characters 867⟩;
case type(cur_p) of
hlist_node, vlist_node, rule_node: act_width ← act_width + width(cur_p);
whatsit_node: ⟨Advance past a whatsit node in the line_break loop 1362⟩;
glue_node: begin ⟨If node cur_p is a legal breakpoint, call try_break; then update the active widths by
      including the glue in glue_ptr(cur_p) 868⟩;
  if second_pass ∧ auto_breaking then ⟨Try to hyphenate the following word 894⟩;
  end;
kern_node: if subtype(cur_p) = explicit then kern_break
  else act_width ← act_width + width(cur_p);
ligature_node: begin f ← font(lig_char(cur_p));
  act_width ← act_width + char_width(f)(char_info(f)(character(lig_char(cur_p))));
  end;
disc_node: ⟨Try to break after a discretionary fragment, then goto done5 869⟩;
math_node: begin 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 863.

867. 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 867⟩ ≡
  begin *prev_p* ← *cur_p*;
  repeat *f* ← *font*(*cur_p*); *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 866.

868. 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*) 868⟩ ≡
  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 866.

869. 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* 869⟩ ≡
  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* 870⟩;
      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* 871⟩;
    decr(*r*); *s* ← *link*(*s*);
    end;
  prev_p ← *cur_p*; *cur_p* ← *s*; **goto** *done5*;
  end

This code is used in section 866.

870. ⟨ Add the width of node *s* to *disc_width* 870 ⟩ ≡

```
if is_char_node(s) then
  begin f ← font(s); 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));
    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 869.

871. ⟨ Add the width of node *s* to *act_width* 871 ⟩ ≡

```
if is_char_node(s) then
  begin f ← font(s); 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));
    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 869.

872. 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 }
```

873. ⟨ Try the final line break at the end of the paragraph, and **goto** *done* if the desired breakpoints have been found 873 ⟩ ≡

```
begin try_break(eject_penalty, hyphenated);
if link(active) ≠ last_active then
  begin ⟨ Find an active node with fewest demerits 874 ⟩;
  if looseness = 0 then goto done;
  ⟨ Find the best active node for the desired looseness 875 ⟩;
  if (actual_looseness = looseness) ∨ final_pass then goto done;
  end;
end
```

This code is used in section 863.

874. ⟨Find an active node with fewest demerits 874⟩ ≡

$r \gets link(active)$; $fewest\_demerits \gets awful\_bad$;
repeat if $type(r) \neq delta\_node$ **then**
  if $total\_demerits(r) < fewest\_demerits$ **then**
    begin $fewest\_demerits \gets total\_demerits(r)$; $best\_bet \gets r$;
    end;
 $r \gets link(r)$;
until $r = last\_active$;
$best\_line \gets line\_number(best\_bet)$

This code is used in section 873.

875. 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 875⟩ ≡

begin $r \gets link(active)$; $actual\_looseness \gets 0$;
repeat if $type(r) \neq delta\_node$ **then**
  begin $line\_diff \gets line\_number(r) - best\_line$;
  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 873.

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

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

$post\_line\_break(final\_widow\_penalty)$

This code is used in section 815.

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

define *next_break* ≡ *prev_break* { new name for *prev_break* after links are reversed }

⟨ Declare subprocedures for *line_break* 826 ⟩ +≡
procedure *post_line_break*(*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 878 ⟩;
cur_line ← *prev_graf* + 1;
repeat ⟨ Justify the line ending at breakpoint *cur_p*, and append it to the current vertical list, together with associated penalties and other insertions 880 ⟩;
incr(*cur_line*); *cur_p* ← *next_break*(*cur_p*);
if *cur_p* ≠ *null* **then**
if ¬*post_disc_break* **then** ⟨ Prune unwanted nodes at the beginning of the next line 879 ⟩;
until *cur_p* = *null*;
if (*cur_line* ≠ *best_line*) ∨ (*link*(*temp_head*) ≠ *null*) **then** *confusion*("line␣breaking");
prev_graf ← *best_line* − 1;
end;

878. 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 878 ⟩ ≡
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 877.

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

⟨Prune unwanted nodes at the beginning of the next line 879⟩ ≡

```
  begin r ← temp_head;
  loop begin q ← link(r);
    if q = cur_break(cur_p) then goto done1;  { cur_break(cur_p) is the next breakpoint }
        { now q cannot be null }
    if is_char_node(q) then goto done1;
    if non_discardable(q) then goto done1;
    if type(q) = kern_node then
      if subtype(q) ≠ explicit then goto done1;
    r ← q;  { now type(q) = glue_node, kern_node, math_node, or penalty_node }
    end;
done1: if r ≠ temp_head then
    begin link(r) ← null; flush_node_list(link(temp_head)); link(temp_head) ← q;
    end;
  end
```

This code is used in section 877.

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

⟨Justify the line ending at breakpoint *cur_p*, and append it to the current vertical list, together with associated penalties and other insertions 880⟩ ≡

⟨Modify the end of the line to reflect the nature of the break and to include `\rightskip`; also set the proper value of *disc_break* 881⟩;
⟨Put the `\leftskip` glue at the left and detach this line 887⟩;
⟨Call the packaging subroutine, setting *just_box* to the justified box 889⟩;
⟨Append the new box to the current vertical list, followed by the list of special nodes taken out of the box by the packager 888⟩;
⟨Append a penalty node, if a nonzero penalty is appropriate 890⟩

This code is used in section 877.

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

⟨Modify the end of the line to reflect the nature of the break and to include `\rightskip`; also set the proper value of *disc_break* 881⟩ ≡

```
q ← cur_break(cur_p); disc_break ← false; post_disc_break ← false;
if q ≠ null then  { q cannot be a char_node }
  if type(q) = glue_node then
    begin delete_glue_ref(glue_ptr(q)); glue_ptr(q) ← right_skip; subtype(q) ← right_skip_code + 1;
    add_glue_ref(right_skip); goto done;
    end
  else begin if type(q) = disc_node then
      ⟨Change discretionary to compulsory and set disc_break ← true 882⟩
    else if (type(q) = math_node) ∨ (type(q) = kern_node) then width(q) ← 0;
    end
else begin q ← temp_head;
  while link(q) ≠ null do q ← link(q);
  end;
⟨Put the \rightskip glue after node q 886⟩;
done:
```

This code is used in section 880.

882. ⟨Change discretionary to compulsory and set *disc_break* ← *true* 882⟩ ≡

```
begin t ← replace_count(q);
⟨Destroy the t nodes following q, and make r point to the following node 883⟩;
if post_break(q) ≠ null then ⟨Transplant the post-break list 884⟩;
if pre_break(q) ≠ null then ⟨Transplant the pre-break list 885⟩;
link(q) ← r; disc_break ← true;
end
```

This code is used in section 881.

883. ⟨Destroy the t nodes following q, and make r point to the following node 883⟩ ≡

```
if t = 0 then r ← link(q)
else begin r ← q;
  while t > 1 do
    begin r ← link(r); decr(t);
    end;
  s ← link(r); r ← link(s); link(s) ← null; flush_node_list(link(q)); replace_count(q) ← 0;
  end
```

This code is used in section 882.

884. 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 884⟩ ≡

```
begin s ← post_break(q);
while link(s) ≠ null do s ← link(s);
link(s) ← r; r ← post_break(q); post_break(q) ← null; post_disc_break ← true;
end
```

This code is used in section 882.

885. 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 885 ⟩ ≡
 begin $s \gets pre\_break(q)$; $link(q) \gets s$;
 while $link(s) \neq null$ **do** $s \gets link(s)$;
 $pre\_break(q) \gets null$; $q \gets s$;
 end

This code is used in section 882.

886. ⟨ Put the `\rightskip` glue after node q 886 ⟩ ≡
 $r \gets new\_param\_glue(right\_skip\_code)$; $link(r) \gets link(q)$; $link(q) \gets r$; $q \gets r$

This code is used in section 881.

887. 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 887 ⟩ ≡
 $r \gets link(q)$; $link(q) \gets null$; $q \gets link(temp\_head)$; $link(temp\_head) \gets r$;
 if $left\_skip \neq zero\_glue$ **then**
  begin $r \gets new\_param\_glue(left\_skip\_code)$; $link(r) \gets q$; $q \gets r$;
  end

This code is used in section 880.

888. ⟨ Append the new box to the current vertical list, followed by the list of special nodes taken out of the box by the packager 888 ⟩ ≡
 $append\_to\_vlist(just\_box)$;
 if $adjust\_head \neq adjust\_tail$ **then**
  begin $link(tail) \gets link(adjust\_head)$; $tail \gets adjust\_tail$;
  end;
 $adjust\_tail \gets null$

This code is used in section 880.

889. 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 889 ⟩ ≡
 if $cur\_line > last\_special\_line$ **then**
  begin $cur\_width \gets second\_width$; $cur\_indent \gets second\_indent$;
  end
 else if $par\_shape\_ptr = null$ **then**
   begin $cur\_width \gets first\_width$; $cur\_indent \gets first\_indent$;
   end
  else begin $cur\_width \gets mem[par\_shape\_ptr + 2 * cur\_line].sc$;
   $cur\_indent \gets mem[par\_shape\_ptr + 2 * cur\_line - 1].sc$;
   end;
 $adjust\_tail \gets adjust\_head$; $just\_box \gets hpack(q, cur\_width, exactly)$; $shift\_amount(just\_box) \gets cur\_indent$

This code is used in section 880.

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

⟨ Append a penalty node, if a nonzero penalty is appropriate 890 ⟩ ≡

if *cur_line* + 1 ≠ *best_line* **then**
  begin *pen* ← *inter_line_penalty*;
  if *cur_line* = *prev_graf* + 1 **then** *pen* ← *pen* + *club_penalty*;
  if *cur_line* + 2 = *best_line* **then** *pen* ← *pen* + *final_widow_penalty*;
  if *disc_break* **then** *pen* ← *pen* + *broken_penalty*;
  if *pen* ≠ 0 **then**
    begin *r* ← *new_penalty*(*pen*); *link*(*tail*) ← *r*; *tail* ← *r*;
    end;
  end

This code is used in section 880.

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

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

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

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

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

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

This code is used in section 863.

892. 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: 0 . . 64; { the number of positions occupied in *hc*; not always a *small_number* }
ha, *hb*: *pointer*; { nodes *ha* . . *hb* should be replaced by the hyphenated result }
hf: *internal_font_number*; { font number of the letters in *hc* }
hu: **array** [0 . . 63] **of** 0 . . 256; { like *hc*, before conversion to lowercase }
hyf_char: *integer*; { hyphen character of the relevant font }
cur_lang, *init_cur_lang*: *ASCII_code*; { current hyphenation table of interest }
l_hyf, *r_hyf*, *init_l_hyf*, *init_r_hyf*: *integer*; { limits on fragment sizes }
hyf_bchar: *halfword*; { boundary character after c_n }

893. Hyphenation routines need a few more local variables.

⟨Local variables for line breaking 862⟩ +≡
j: *small_number*; { an index into *hc* or *hu* }
c: 0 . . 255; { character being considered for hyphenation }

894. 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 894⟩ ≡
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 896⟩;
if *l_hyf* + *r_hyf* > 63 **then goto** *done1*;
⟨Skip to node *hb*, putting letters into *hu* and *hc* 897⟩;
⟨Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto** *done1* 899⟩;
hyphenate;
end;
done1: **end**

This code is used in section 866.

895. ⟨Declare subprocedures for *line_break* 826⟩ +≡
⟨Declare the function called *reconstitute* 906⟩
procedure *hyphenate*;
label *common_ending*, *done*, *found*, *found1*, *found2*, *not_found*, *exit*;
var ⟨Local variables for hyphenation 901⟩
begin ⟨Find hyphen locations for the word in *hc*, or **return** 923⟩;
⟨If no hyphens were found, **return** 902⟩;
⟨Replace nodes *ha* . . *hb* by a sequence of nodes that includes the discretionary hyphens 903⟩;
exit: **end**;

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

```
⟨Skip to node ha, or goto done1 if no hyphenation should be attempted 896⟩ ≡
  loop begin if is_char_node(s) then
        begin c ← qo(character(s)); hf ← font(s);
        end
    else if type(s) = ligature_node then
          if lig_ptr(s) = null then goto continue
          else begin q ← lig_ptr(s); c ← qo(character(q)); hf ← font(q);
            end
        else if (type(s) = kern_node) ∧ (subtype(s) = normal) then goto continue
          else if type(s) = whatsit_node then
              begin ⟨Advance past a whatsit node in the pre-hyphenation loop 1363⟩;
              goto continue;
              end
            else goto done1;
    if lc_code(c) ≠ 0 then
        if (lc_code(c) = c) ∨ (uc_hyph > 0) then goto done2
        else goto done1;
  continue: prev_s ← s; s ← link(prev_s);
    end;
done2: hyf_char ← hyphen_char[hf];
  if hyf_char < 0 then goto done1;
  if hyf_char > 255 then goto done1;
  ha ← prev_s
```

This code is used in section 894.

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

```
⟨Skip to node hb, putting letters into hu and hc 897⟩ ≡
  hn ← 0;
  loop begin if is_char_node(s) then
        begin if font(s) ≠ hf then goto done3;
        hyf_bchar ← character(s); c ← qo(hyf_bchar);
        if lc_code(c) = 0 then goto done3;
        if hn = 63 then goto done3;
        hb ← s; incr(hn); hu[hn] ← c; hc[hn] ← lc_code(c); hyf_bchar ← non_char;
        end
    else if type(s) = ligature_node then ⟨Move the characters of a ligature node to hu and hc; but goto
              done3 if they are not all letters 898⟩
        else if (type(s) = kern_node) ∧ (subtype(s) = normal) then
            begin hb ← s; hyf_bchar ← font_bchar[hf];
            end
          else goto done3;
    s ← link(s);
    end;
done3:
```

This code is used in section 894.

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

⟨Move the characters of a ligature node to *hu* and *hc*; but **goto** *done3* if they are not all letters 898⟩ ≡

```
begin if font(lig_char(s)) ≠ 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 897.

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

```
if hn < l_hyf + r_hyf then goto done1;   { l_hyf and r_hyf are ≥ 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 894.

900. 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? }

901. ⟨ Local variables for hyphenation 901 ⟩ ≡
i, *j*, *l*: 0 . . 65; { indices into *hc* or *hu* }
q, *r*, *s*: *pointer*; { temporary registers for list manipulation }
bchar: *halfword*; { boundary character of hyphenated word, or *non_char* }

See also sections 912, 922, and 929.

This code is used in section 895.

902. 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** 902 ⟩ ≡
  for *j* ← *l_hyf* **to** *hn* − *r_hyf* **do**
    if *odd*(*hyf*[*j*]) **then goto** *found1*;
  return;
found1:

This code is used in section 895.

903. 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 903⟩ ≡
  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 913⟩;
  flush_list(init_list)
```

This code is used in section 895.

904. 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 $p_{a-1} = \texttt{x}$ and $p_a = \texttt{y}$, 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`.

905. 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 }

906. ⟨ Declare the function called *reconstitute* 906 ⟩ ≡
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* 908 ⟩;
continue: ⟨ If there's a ligature or kern at the cursor position, update the data structures, possibly advancing *j*; continue until the cursor moves 909 ⟩;
  ⟨ Append a ligature and/or kern to the translation; **goto** *continue* if the stack of inserted ligatures is nonempty 910 ⟩;
  reconstitute ← *j*;
  end;

This code is used in section 895.

907. 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? }

908. **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* 908 ⟩ ≡
  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 906.

909. 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 909⟩ ≡

```
if cur_l = non_char then
  begin k ← bchar_label[hf];
  if k = non_address then goto done else q ← font_info[k].qqqq;
  end
else begin q ← char_info(hf)(cur_l);
  if char_tag(q) ≠ lig_tag then goto done;
  k ← lig_kern_start(hf)(q); q ← font_info[k].qqqq;
  if skip_byte(q) > stop_flag then
    begin k ← lig_kern_restart(hf)(q); q ← font_info[k].qqqq;
    end;
  end; { now k is the starting address of the lig/kern program }
if cur_rh < non_char then test_char ← cur_rh else test_char ← cur_r;
loop begin if next_char(q) = test_char then
    if skip_byte(q) ≤ stop_flag then
      if cur_rh < non_char then
        begin hyphen_passed ← j; hchar ← non_char; cur_rh ← non_char; goto continue;
        end
      else begin if hchar < non_char then
          if odd(hyf[j]) then
            begin hyphen_passed ← j; hchar ← non_char;
            end;
        if op_byte(q) < kern_flag then
          ⟨Carry out a ligature replacement, updating the cursor structure and possibly advancing j;
              goto continue if the cursor doesn't advance, otherwise goto done 911⟩;
        w ← 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 ← k + qo(skip_byte(q)) + 1; q ← font_info[k].qqqq;
  end;
done:
```

This code is used in section 906.

910. **define** *wrap_lig*(#) ≡
    if *ligature_present* **then**
      begin $p \leftarrow$ *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; $t \leftarrow p$; *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 $hu[j+1]$ }
      $t \leftarrow$ *link*(t); *incr*(j);
      end;
    $p \leftarrow$ *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 910 ⟩ ≡
  wrap_lig(*rt_hit*);
  if $w \neq 0$ **then**
    begin *link*(t) ← *new_kern*(w); $t \leftarrow$ *link*(t); $w \leftarrow 0$;
    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 906.

911. ⟨ Carry out a ligature replacement, updating the cursor structure and possibly advancing j; **goto** *continue* if the cursor doesn't advance, otherwise **goto** *done* 911 ⟩ ≡

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 909.

912. 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 901 ⟩ +≡
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 }

913. When the following code is performed, $hyf[0]$ and $hyf[hn]$ will be zero.

⟨Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 913⟩ ≡

```
repeat l ← j; j ← 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 914⟩;
until j > hn;
link(s) ← q
```

This code is used in section 903.

914. 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 914⟩ ≡

```
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) 915⟩;
  ⟨Put the characters hu[i + 1 . .] into post_break(r), appending to this list and to major_tail until
      synchronization has been achieved 916⟩;
  ⟨Move pointer s to the end of the current list, and set replace_count(r) appropriately 918⟩;
  hyphen_passed ← j − 1; link(hold_head) ← null;
until ¬odd(hyf[j − 1])
```

This code is used in section 913.

915. 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*) 915⟩ ≡

```
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 914.

916. 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 916⟩ ≡

```
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 917⟩;
  end
```

This code is used in section 914.

917. ⟨Append characters of *hu*[*j* ..] to *major_tail*, advancing *j* 917⟩ ≡

```
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 916.

918. 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 918 ⟩ ≡
  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 914.

919. Hyphenation. When a word $hc[1\mathinner{\ldotp\ldotp}hn]$ 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 $hc[0\mathinner{\ldotp\ldotp}(hn+1)]$. 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 $hc[(j+1)\mathinner{\ldotp\ldotp}(j+k)]$, 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.

920. 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 $p_1 \ldots p_k$.

Comparatively few different number sequences $n_0 \ldots n_k$ actually occur, since most of the n's are generally zero. Therefore the number sequences are encoded in such a way that $trie\_op(z_k)$ is only one byte long. If $trie\_op(z_k) \ne min\_quarterword$, when $p_1 \ldots p_k$ has matched the letters in $hc[(l-k+1)\mathinner{\ldotp\ldotp}l\,]$ of language t, we perform all of the required operations for this pattern by carrying out the following little program: Set $v \leftarrow trie\_op(z_k)$. Then set $v \leftarrow v + op\_start[t]$, $hyf[l - hyf\_distance[v]] \leftarrow \max(hyf[l - hyf\_distance[v]], hyf\_num[v])$, and $v \leftarrow hyf\_next[v]$; repeat, if necessary, until $v = min\_quarterword$.

⟨ Types in the outer block 18 ⟩ +≡
trie_pointer = 0 .. *trie_size*; { an index into *trie* }

921. define *trie_link*(#) ≡ *trie*[#].*rh* { "downward" link in a trie }
define *trie_char*(#) ≡ *trie*[#].*b1* { character matched at this trie location }
define *trie_op*(#) ≡ *trie*[#].*b0* { program for hyphenation at this trie location }

⟨ Global variables 13 ⟩ +≡
trie: **array** [*trie_pointer*] **of** *two_halves*; { *trie_link*, *trie_char*, *trie_op* }
hyf_distance: **array** [1 .. *trie_op_size*] **of** *small_number*; { position $k - j$ of n_j }
hyf_num: **array** [1 .. *trie_op_size*] **of** *small_number*; { value of n_j }
hyf_next: **array** [1 .. *trie_op_size*] **of** *quarterword*; { continuation code }
op_start: **array** [*ASCII_code*] **of** 0 .. *trie_op_size*; { offset for current language }

922. ⟨ Local variables for hyphenation 901 ⟩ +≡
z: *trie_pointer*; { an index into *trie* }
v: *integer*; { an index into *hyf_distance*, etc. }

923. 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** 923 ⟩ ≡
  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 930 ⟩;
  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 $trie\_op(z) \neq min\_quarterword$ **then** ⟨ Store maximum values in the *hyf* table 924 ⟩;
      $incr(l)$; $z \leftarrow trie\_link(z) + hc[l]$;
      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 895.

924. ⟨ Store maximum values in the *hyf* table 924 ⟩ ≡
  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 $v = min\_quarterword$;
  end

This code is used in section 923.

925. 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.

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 ⟩ +≡
  $hyph\_pointer = 0\,..\,hyph\_size$; { an index into the ordered hash table }

926. ⟨ Global variables 13 ⟩ +≡
hyph_word: **array** [*hyph_pointer*] **of** *str_number*; { exception words }
hyph_list: **array** [*hyph_pointer*] **of** *pointer*; { lists of hyphen positions }
hyph_count: *hyph_pointer*; { the number of words in the exception dictionary }

927. ⟨ Local variables for initialization 19 ⟩ +≡
z: *hyph_pointer*; { runs through the exception dictionary }

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

for $z \leftarrow 0$ **to** *hyph_size* **do**
  begin *hyph_word*[z] ← 0; *hyph_list*[z] ← *null*;
  end;
hyph_count ← 0;

929. 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 901 ⟩ +≡
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* }

930. 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 930 ⟩ ≡
  $h \leftarrow hc[1]$; *incr*(*hn*); *hc*[*hn*] ← *cur_lang*;
  for $j \leftarrow 2$ **to** *hn* **do** $h \leftarrow (h + h + hc[j])$ **mod** *hyph_size*;
  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* 931 ⟩;
    if $h > 0$ **then** *decr*(h) **else** $h \leftarrow$ *hyph_size*;
    end;
not_found: *decr*(*hn*)

This code is used in section 923.

931. ⟨ 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* 931 ⟩ ≡
  $k \leftarrow$ *hyph_word*[h];
  if $k = 0$ **then goto** *not_found*;
  if *length*(k) < *hn* **then goto** *not_found*;
  if *length*(k) = *hn* **then**
    begin $j \leftarrow 1$; $u \leftarrow$ *str_start*[k];
    repeat if *so*(*str_pool*[u]) < *hc*[j] **then goto** *not_found*;
      if *so*(*str_pool*[u]) > *hc*[j] **then goto** *done*;
      incr(j); *incr*(u);
    until $j > hn$;
    ⟨ Insert hyphens as specified in *hyph_list*[h] 932 ⟩;
    decr(*hn*); **goto** *found*;
    end;
done:

This code is used in section 930.

932. ⟨ Insert hyphens as specified in *hyph_list*[h] 932 ⟩ ≡
  $s \leftarrow$ *hyph_list*[h];
  while $s \neq$ *null* **do**
    begin *hyf*[*info*(s)] ← 1; $s \leftarrow$ *link*(s);
    end

This code is used in section 931.

933. ⟨Search *hyph_list* for pointers to *p* 933⟩ ≡

```
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.

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

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

```
define set_cur_lang ≡
    if language ≤ 0 then cur_lang ← 0
    else if language > 255 then cur_lang ← 0
      else cur_lang ← language

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

935. ⟨Enter as many hyphenation exceptions as are listed, until coming to a right brace; then **return** 935⟩ ≡

```
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 937⟩;
  char_num: begin scan_char_num; cur_chr ← cur_val; cur_cmd ← char_given; goto reswitch;
    end;
  spacer, right_brace: begin if n > 1 then ⟨Enter a hyphenation exception 939⟩;
    if cur_cmd = right_brace then return;
    n ← 0; p ← null;
    end;
  othercases ⟨Give improper \hyphenation error 936⟩
  endcases;
  end
```

This code is used in section 934.

936. ⟨Give improper `\hyphenation` error 936⟩ ≡
```
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 935.

937. ⟨Append a new letter or hyphen 937⟩ ≡
```
if cur_chr = "-" then ⟨Append the value n to list p 938⟩
else begin 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] ← lc_code(cur_chr);
      end;
  end
```
This code is used in section 935.

938. ⟨Append the value n to list p 938⟩ ≡
```
begin if n < 63 then
  begin q ← get_avail; link(q) ← p; info(q) ← n; p ← q;
  end;
end
```
This code is used in section 937.

939. ⟨Enter a hyphenation exception 939⟩ ≡
```
begin incr(n); hc[n] ← cur_lang; str_room(n); h ← 0;
for j ← 1 to n do
  begin h ← (h + h + hc[j]) mod hyph_size; append_char(hc[j]);
  end;
s ← make_string; ⟨Insert the pair (s, p) into the exception table 940⟩;
end
```
This code is used in section 935.

940. ⟨Insert the pair (s,p) into the exception table 940⟩ ≡
```
if hyph_count = hyph_size 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) 941⟩;
  if h > 0 then decr(h) else h ← hyph_size;
  end;
hyph_word[h] ← s; hyph_list[h] ← p
```
This code is used in section 939.

941. ⟨ If the string *hyph_word*[*h*] is less than or equal to *s*, interchange (*hyph_word*[*h*], *hyph_list*[*h*]) with (s, p) 941 ⟩ ≡

$k \leftarrow hyph\_word[h]$;
if $length(k) < length(s)$ **then goto** *found*;
if $length(k) > length(s)$ **then goto** *not_found*;
$u \leftarrow str\_start[k]$; $v \leftarrow str\_start[s]$;
repeat if $str\_pool[u] < str\_pool[v]$ **then goto** *found*;
  if $str\_pool[u] > str\_pool[v]$ **then goto** *not_found*;
  $incr(u)$; $incr(v)$;
until $u = str\_start[k + 1]$;
found: $q \leftarrow hyph\_list[h]$; $hyph\_list[h] \leftarrow p$; $p \leftarrow q$;
  $t \leftarrow hyph\_word[h]$; $hyph\_word[h] \leftarrow s$; $s \leftarrow t$;
not_found:

This code is used in section 940.

942. 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* 826 ⟩ +≡
init ⟨ Declare procedures for preprocessing hyphenation patterns 944 ⟩
tini

943. 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 $hyf\_next[v'] = min\_quarterword$.

TEX computes an appropriate value v with the *new_trie_op* subroutine below, by setting

$$v' \leftarrow new\_trie\_op(0, 1, min\_quarterword), \qquad v \leftarrow new\_trie\_op(3, 2, v').$$

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** [−*trie_op_size* .. *trie_op_size*] **of** 0 .. *trie_op_size*;
{ trie op codes for quadruples }
trie_used: **array** [*ASCII_code*] **of** *quarterword*; { largest opcode used so far for this language }
trie_op_lang: **array** [1 .. *trie_op_size*] **of** *ASCII_code*; { language part of a hashed quadruple }
trie_op_val: **array** [1 .. *trie_op_size*] **of** *quarterword*; { opcode corresponding to a hashed quadruple }
trie_op_ptr: 0 .. *trie_op_size*; { number of stored ops so far }
tini

944. It's tempting to remove the *overflow* stops in the following procedure; *new_trie_op* could return *min_quarterword* (thereby simply ignoring part of a hyphenation pattern) instead of aborting the job. However, that would lead to different hyphenation results on different installations of TEX using the same patterns. The *overflow* stops are necessary for portability of patterns.

⟨Declare procedures for preprocessing hyphenation patterns 944⟩ ≡

```
function new_trie_op(d, n : small_number; v : quarterword): quarterword;
  label exit;
  var h: −trie_op_size .. trie_op_size; { trial hash location }
    u: quarterword; { trial op code }
    l: 0 .. trie_op_size; { pointer to stored data }
  begin h ← abs(n + 313 ∗ d + 361 ∗ v + 1009 ∗ cur_lang) mod (trie_op_size + trie_op_size) − trie_op_size;
  loop begin l ← trie_op_hash[h];
    if l = 0 then { empty position found for a new op }
      begin if trie_op_ptr = trie_op_size then overflow("pattern␣memory␣ops", trie_op_size);
      u ← trie_used[cur_lang];
      if u = max_quarterword then
        overflow("pattern␣memory␣ops␣per␣language", max_quarterword − min_quarterword);
      incr(trie_op_ptr); incr(u); trie_used[cur_lang] ← u; hyf_distance[trie_op_ptr] ← d;
      hyf_num[trie_op_ptr] ← n; hyf_next[trie_op_ptr] ← v; trie_op_lang[trie_op_ptr] ← cur_lang;
      trie_op_hash[h] ← trie_op_ptr; trie_op_val[trie_op_ptr] ← u; new_trie_op ← u; return;
      end;
    if (hyf_distance[l] = d) ∧ (hyf_num[l] = n) ∧ (hyf_next[l] = v) ∧ (trie_op_lang[l] = cur_lang) then
      begin new_trie_op ← trie_op_val[l]; return;
      end;
    if h > −trie_op_size then decr(h) else h ← trie_op_size;
    end;
exit: end;
```

See also sections 948, 949, 953, 957, 959, 960, and 966.
This code is used in section 942.

945. 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 945⟩ ≡

```
  op_start[0] ← −min_quarterword;
  for j ← 1 to 255 do op_start[j] ← op_start[j − 1] + qo(trie_used[j − 1]);
  for j ← 1 to trie_op_ptr do trie_op_hash[j] ← op_start[trie_op_lang[j]] + trie_op_val[j]; { destination }
  for j ← 1 to trie_op_ptr do
    while trie_op_hash[j] > j do
      begin k ← trie_op_hash[j];
      t ← hyf_distance[k]; hyf_distance[k] ← hyf_distance[j]; hyf_distance[j] ← t;
      t ← hyf_num[k]; hyf_num[k] ← hyf_num[j]; hyf_num[j] ← t;
      t ← hyf_next[k]; hyf_next[k] ← hyf_next[j]; hyf_next[j] ← t;
      trie_op_hash[j] ← trie_op_hash[k]; trie_op_hash[k] ← k;
      end
```

This code is used in section 952.

946. 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** $trie\_used[k] \leftarrow min\_quarterword$;
$trie\_op\_ptr \leftarrow 0$;

947. 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 array** [$trie\_pointer$] **of** $packed\_ASCII\_code$; { characters to match }
$trie\_o$: **packed array** [$trie\_pointer$] **of** $quarterword$; { operations to perform }
$trie\_l$: **packed array** [$trie\_pointer$] **of** $trie\_pointer$; { left subtrie links }
$trie\_r$: **packed array** [$trie\_pointer$] **of** $trie\_pointer$; { right subtrie links }
$trie\_ptr$: $trie\_pointer$; { the number of nodes in the trie }
$trie\_hash$: **packed array** [$trie\_pointer$] **of** $trie\_pointer$; { used to identify equivalent subtries }
tini

948. 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 944 ⟩ +≡
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 $h \leftarrow abs(trie\_c[p] + 1009 * trie\_o[p] + 2718 * trie\_l[p] + 3142 * trie\_r[p])$ **mod** $trie\_size$;
loop begin $q \leftarrow trie\_hash[h]$;
if $q = 0$ **then**
begin $trie\_hash[h] \leftarrow p$; $trie\_node \leftarrow p$; **return**;
end;
if $(trie\_c[q] = trie\_c[p]) \land (trie\_o[q] = trie\_o[p]) \land (trie\_l[q] = trie\_l[p]) \land (trie\_r[q] = trie\_r[p])$ **then**
begin $trie\_node \leftarrow q$; **return**;
end;
if $h > 0$ **then** $decr(h)$ **else** $h \leftarrow trie\_size$;
end;
$exit$: **end**;

949. 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 944 ⟩ +≡
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;

950. The compressed trie will be packed into the *trie* array using a "top-down first-fit" procedure. This is a little tricky, so the reader should pay close attention: The *trie_hash* array is cleared to zero again and renamed *trie_ref* for this phase of the operation; later on, *trie_ref*[*p*] will be nonzero only if the linked trie node *p* is the smallest character in a family and if the characters *c* of that family have been allocated to locations *trie_ref*[*p*] + *c* in the *trie* array. Locations of *trie* that are in use will have *trie_link* = 0, while the unused holes in *trie* will be doubly linked with *trie_link* pointing to the next larger vacant location and *trie_back* pointing to the next smaller one. This double linking will have been carried out only as far as *trie_max*, where *trie_max* is the largest index of *trie* that will be needed. To save time at the low end of the trie, we maintain array entries *trie_min*[*c*] pointing to the smallest hole that is greater than *c*. Another array *trie_taken* tells whether or not a given location is equal to *trie_ref*[*p*] for some *p*; this array is used to ensure that distinct nodes in the compressed trie will have distinct *trie_ref* entries.

  define *trie_ref* ≡ *trie_hash* { where linked trie families go into *trie* }
  define *trie_back*(#) ≡ *trie*[#].*lh* { backward links in *trie* holes }

⟨ Global variables 13 ⟩ +≡
  init *trie_taken*: **packed array** [1 .. *trie_size*] **of** *boolean*; { does a family start here? }
  trie_min: **array** [*ASCII_code*] **of** *trie_pointer*; { the first possible slot for each character }
  trie_max: *trie_pointer*; { largest location used in *trie* }
  trie_not_ready: *boolean*; { is the trie still in linked form? }
  tini

951. 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*; *trie_root* ← 0; *trie_c*[0] ← *si*(0); *trie_ptr* ← 0;

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

⟨ Get ready to compress the trie 952 ⟩ ≡
  ⟨ Sort the hyphenation op tables into proper order 945 ⟩;
  for *p* ← 0 **to** *trie_size* **do** *trie_hash*[*p*] ← 0;
  trie_root ← *compress_trie*(*trie_root*); { identify equivalent subtries }
  for *p* ← 0 **to** *trie_ptr* **do** *trie_ref*[*p*] ← 0;
  for *p* ← 0 **to** 255 **do** *trie_min*[*p*] ← *p* + 1;
  trie_link(0) ← 1; *trie_max* ← 0

This code is used in section 966.

953. 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 944 ⟩ +≡
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 $trie\_max \ge h + 256$ 954 ⟩;
    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* 955 ⟩;
  not_found: $z \leftarrow trie\_link(z)$; { move to the next hole }
    end;
found: ⟨ Pack the family into *trie* relative to *h* 956 ⟩;
  end;

954. 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 $trie\_max \ge trie\_link(z)$.

⟨ Ensure that $trie\_max \ge h + 256$ 954 ⟩ ≡
  if $trie\_max < h + 256$ **then**
    begin if $trie\_size \le h + 256$ **then** *overflow*("pattern␣memory", *trie_size*);
    repeat *incr*(*trie_max*); *trie_taken*[*trie_max*] ← *false*; *trie_link*(*trie_max*) ← *trie_max* + 1;
      trie_back(*trie_max*) ← *trie_max* − 1;
    until $trie\_max = h + 256$;
    end

This code is used in section 953.

955. ⟨ If all characters of the family fit relative to *h*, then **goto** *found*, otherwise **goto** *not_found* 955 ⟩ ≡
  $q \leftarrow trie\_r[p]$;
  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 953.

956. ⟨ Pack the family into *trie* relative to *h* 956 ⟩ ≡

```
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 953.

957. To pack the entire linked trie, we use the following recursive procedure.

⟨ Declare procedures for preprocessing hyphenation patterns 944 ⟩ +≡

```
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;
```

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

⟨ Move the data into *trie* 958 ⟩ ≡

```
h.rh ← 0; h.b0 ← min_quarterword; h.b1 ← min_quarterword;
    { trie_link ← 0, trie_op ← min_quarterword, trie_char ← qi(0) }
if trie_root = 0 then { no patterns were given }
  begin for r ← 0 to 256 do trie[r] ← h;
  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); trie[r] ← h; r ← s;
  until r > trie_max;
  end;
trie_char(0) ← qi("?"); { make trie_char(c) ≠ c for all c }
```

This code is used in section 966.

959. 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 944⟩ +≡
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;

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

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

961. 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 961⟩ ≡

```
k ← 0; hyf[0] ← 0; digit_sensed ← false;
loop begin get_x_token;
  case cur_cmd of
  letter, other_char: ⟨Append a new letter or a hyphen level 962⟩;
  spacer, right_brace: begin if k > 0 then ⟨Insert a new pattern into the linked trie 963⟩;
    if cur_cmd = right_brace then goto done;
    k ← 0; hyf[0] ← 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 960.

962. ⟨Append a new letter or a hyphen level 962⟩ ≡

```
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 961.

963. When the following code comes into play, the pattern $p_1 \ldots p_k$ appears in $hc[1\,..\,k]$, and the corresponding sequence of numbers $n_0 \ldots n_k$ appears in $hyf[0\,..\,k]$.

⟨Insert a new pattern into the linked trie 963⟩ ≡
  begin ⟨Compute the trie op code, v, and set $l \leftarrow 0$ 965⟩;
  $q \leftarrow 0$; $hc[0] \leftarrow cur\_lang$;
  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 964⟩;
    $q \leftarrow p$; { now node q represents $p_1 \ldots p_{l-1}$ }
    end;
  if $trie\_o[q] \ne min\_quarterword$ **then**
    begin *print_err*("Duplicate␣pattern"); *help1*("(See␣Appendix␣H.)"); *error*;
    end;
  $trie\_o[q] \leftarrow v$;
  end

This code is used in section 961.

964. ⟨Insert a new trie node between q and p, and make p point to it 964⟩ ≡
  begin if $trie\_ptr = trie\_size$ **then** *overflow*("pattern␣memory", *trie_size*);
  $incr(trie\_ptr)$; $trie\_r[trie\_ptr] \leftarrow p$; $p \leftarrow trie\_ptr$; $trie\_l[p] \leftarrow 0$;
  if *first_child* **then** $trie\_l[q] \leftarrow p$ **else** $trie\_r[q] \leftarrow p$;
  $trie\_c[p] \leftarrow si(c)$; $trie\_o[p] \leftarrow min\_quarterword$;
  end

This code is used in section 963.

965. ⟨Compute the trie op code, v, and set $l \leftarrow 0$ 965⟩ ≡
  if $hc[1] = 0$ **then** $hyf[0] \leftarrow 0$;
  if $hc[k] = 0$ **then** $hyf[k] \leftarrow 0$;
  $l \leftarrow k$; $v \leftarrow min\_quarterword$;
  loop begin if $hyf[l] \ne 0$ **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 963.

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

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

967. 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.

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

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* 969 ⟩;
    whatsit_node, *mark_node*, *ins_node*: **begin** *prev_p* ← *p*; *p* ← *link*(*prev_p*);
      end;
    glue_node, *kern_node*, *penalty_node*: **begin** *q* ← *p*; *p* ← *link*(*q*); *link*(*q*) ← *null*; *link*(*prev_p*) ← *p*;
      flush_node_list(*q*);
      end;
    othercases *confusion*("pruning")
    endcases;
  prune_page_top ← *link*(*temp_head*);
  end;

969. ⟨ Insert glue for *split_top_skip* and set *p* ← *null* 969 ⟩ ≡
  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 968.

970. 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 972 ⟩;
prev_p ← *p*; *p* ← *link*(*prev_p*);
end;
done: *vert_break* ← *best_place*;
end;

971. 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 }

972. 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 972 ⟩ ≡
  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 973 ⟩;
  ⟨ 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 974 ⟩;
  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 976 ⟩;
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 970.

973. ⟨ 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 973 ⟩ ≡
  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* 1365 ⟩;
  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 972.

974. **define** *deplorable* ≡ 100000 { more than *inf_bad*, but less than *awful_bad* }

⟨ Check if node *p* is a new champion breakpoint; then **goto** *done* if *p* is a forced break or if the page-so-far is already too full 974 ⟩ ≡

```
if pi < inf_penalty then
  begin ⟨ Compute the badness, b, using awful_bad if the box is too full 975 ⟩;
  if b < awful_bad then
    if pi ≤ eject_penalty then b ← pi
    else if b < inf_bad then b ← b + pi
      else b ← deplorable;
  if b ≤ least_cost then
    begin best_place ← p; least_cost ← b; best_height_plus_depth ← cur_height + prev_dp;
    end;
  if (b = awful_bad) ∨ (pi ≤ eject_penalty) then goto done;
  end
```

This code is used in section 972.

975. ⟨ Compute the badness, *b*, using *awful_bad* if the box is too full 975 ⟩ ≡

```
if cur_height < h then
  if (active_height[3] ≠ 0) ∨ (active_height[4] ≠ 0) ∨ (active_height[5] ≠ 0) then b ← 0
  else b ← badness(h − cur_height, active_height[2])
else if cur_height − h > active_height[6] then b ← awful_bad
  else b ← badness(cur_height − h, active_height[6])
```

This code is used in section 974.

976. 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* 976 ⟩ ≡

```
if type(p) = kern_node then q ← p
else begin q ← glue_ptr(p);
  active_height[2 + stretch_order(q)] ← active_height[2 + stretch_order(q)] + stretch(q);
  active_height[6] ← active_height[6] + shrink(q);
  if (shrink_order(q) ≠ normal) ∧ (shrink(q) ≠ 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 ← new_spec(q);
    shrink_order(r) ← normal; delete_glue_ref(q); glue_ptr(p) ← r; q ← r;
    end;
  end;
cur_height ← cur_height + prev_dp + width(q); prev_dp ← 0
```

This code is used in section 972.

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

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

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

```
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 ← box(n);
  if split_first_mark ≠ null then
    begin delete_token_ref(split_first_mark); split_first_mark ← null; delete_token_ref(split_bot_mark);
    split_bot_mark ← null;
    end;
  ⟨Dispense with trivial cases of void or bad boxes 978⟩;
  q ← vert_break(list_ptr(v), h, split_max_depth);
  ⟨Look at all the marks in nodes before the break, and set the final link to null at the break 979⟩;
  q ← prune_page_top(q); p ← list_ptr(v); free_node(v, box_node_size);
  if q = null then box(n) ← null   { the eq_level of the box stays the same }
  else box(n) ← vpack(q, natural);
  vsplit ← vpackage(p, h, exactly, split_max_depth);
exit: end;
```

978. ⟨Dispense with trivial cases of void or bad boxes 978⟩ ≡

```
  if v = null then
    begin vsplit ← null; return;
    end;
  if type(v) ≠ vlist_node then
    begin print_err(""); print_esc("vsplit"); print("␣needs␣a␣"); print_esc("vbox");
    help2("The␣box␣you␣are␣trying␣to␣split␣is␣an␣\hbox.")
    ("I␣can´t␣split␣such␣a␣box,␣so␣I´ll␣leave␣it␣alone."); error; vsplit ← null; return;
    end
```

This code is used in section 977.

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

⟨Look at all the marks in nodes before the break, and set the final link to *null* at the break 979⟩ ≡

```
p ← list_ptr(v);
if p = q then list_ptr(v) ← null
else loop begin if type(p) = mark_node then
      if split_first_mark = null then
        begin split_first_mark ← mark_ptr(p); split_bot_mark ← split_first_mark;
        token_ref_count(split_first_mark) ← token_ref_count(split_first_mark) + 2;
        end
      else begin delete_token_ref(split_bot_mark); split_bot_mark ← mark_ptr(p);
        add_token_ref(split_bot_mark);
        end;
    if link(p) = q then
      begin link(p) ← null; goto done;
      end;
    p ← link(p);
    end;
done:
```

This code is used in section 977.

980. 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* }

981. 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 $r_1 = link(page\_ins\_head)$; node r_j is followed by $r_{j+1} = link(r_j)$; and if there are n items we have $r_{n+1} = page\_ins\_head$. The *subtype* field of each node in this list refers to an insertion number; for example, '`\insert 250`' would correspond to a node whose *subtype* is *qi*(250) (the same as the *subtype* field of the relevant *ins_node*). These *subtype* fields are in increasing order, and *subtype*(*page_ins_head*) = *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 *type*(r) = *inserting*, then *height*(r) contains the total of the height-plus-depth dimensions of the box and all its inserts seen so far. If *type*(r) = *split_up*, then no more insertions will be made into this box, because at least one previous insertion was too big to fit on the current page; *broken_ptr*(r) points to the node where that insertion will be split, if TEX decides to split it, *broken_ins*(r) points to the insertion node that was tentatively split, and *height*(r) includes also the natural height plus depth of the part that would be split off.

In both cases, *last_ins_ptr*(r) points to the last *ins_node* encountered for box *qo*(*subtype*(r)) that would be at least partially inserted on the next page; and *best_ins_ptr*(r) points to the last such *ins_node* that should actually be inserted, to get the page with minimum badness among all page breaks considered so far. We have *best_ins_ptr*(r) = *null* if and only if no insertion for this box should be made to produce this optimum page.

The data structure definitions here use the fact that the *height* field appears in the fourth word of a box node.

define *page_ins_node_size* = 4 { number of words for a page insertion node }
define *inserting* = 0 { an insertion class that has not yet overflowed }
define *split_up* = 1 { an overflowed insertion class }
define *broken_ptr*(#) ≡ *link*(# + 1) { an insertion for this class will break here if anywhere }
define *broken_ins*(#) ≡ *info*(# + 1) { this insertion might break at *broken_ptr* }
define *last_ins_ptr*(#) ≡ *link*(# + 2) { the most recent insertion for this *subtype* }
define *best_ins_ptr*(#) ≡ *info*(# + 2) { the optimum most recent insertion }

⟨ Initialize the special list heads and constant nodes 790 ⟩ +≡
subtype(*page_ins_head*) ← *qi*(255); *type*(*page_ins_head*) ← *split_up*; *link*(*page_ins_head*) ← *page_ins_head*;

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

The variable *page_depth* records the depth of the current page; it has been adjusted so that it is at most *page_max_depth*. The variable *last_glue* points to the glue specification of the most recent node contributed from the contribution list, if this was a glue node; otherwise *last_glue* = *max_halfword*. (If the contribution list is nonempty, however, the value of *last_glue* is not necessarily accurate.) The variables *last_penalty* 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 insertions that were held over }

983. ⟨ 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);

984. ⟨ 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;

985. **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;

986. ⟨ Show the status of the current page 986 ⟩ ≡
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.

987. 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;

988. 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 790 ⟩ +≡
  type(*page_head*) ← *glue_node*; *subtype*(*page_head*) ← *normal*;

989. 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? }

990. ⟨ Set initial values of key variables 21 ⟩ +≡
  output_active ← *false*; *insert_penalties* ← 0;

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

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

This code is used in sections 215 and 1017.

992. 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;
```

993. 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;

994. 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* 1012 ⟩
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* 996 ⟩;
    ⟨ 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 997 ⟩;
  until *link*(*contrib_head*) = *null*;
  ⟨ Make the contribution list empty by setting its tail to *contrib_head* 995 ⟩;
exit: **end**;

995. **define** *contrib_tail* ≡ *nest*[0].*tail_field* { tail of the contribution list }

⟨ Make the contribution list empty by setting its tail to *contrib_head* 995 ⟩ ≡
  if *nest_ptr* = 0 **then** *tail* ← *contrib_head* { vertical mode }
  else *contrib_tail* ← *contrib_head* { other modes }

This code is used in section 994.

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

if *last_glue* $\neq$ *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 994.

997. 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 997 ⟩ ≡

⟨ 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 1000 ⟩;
⟨ 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* 1005 ⟩;
if $(type(p) < glue\_node) \lor (type(p) > kern\_node)$ **then goto** *contribute*;
update_heights: ⟨ Update the current page measurements with respect to the glue or kern specified by node *p* 1004 ⟩;
contribute: ⟨ Make sure that *page_max_depth* is not exceeded 1003 ⟩;
⟨ Link node *p* into the current page and **goto** *done* 998 ⟩;
done1: ⟨ Recycle node *p* 999 ⟩;
done:

This code is used in section 994.

998. ⟨ Link node *p* into the current page and **goto** *done* 998 ⟩ ≡

link(*page_tail*) ← *p*; *page_tail* ← *p*; *link*(*contrib_head*) ← *link*(*p*); *link*(*p*) ← *null*; **goto** *done*

This code is used in section 997.

999. ⟨ Recycle node *p* 999 ⟩ ≡

link(*contrib_head*) ← *link*(*p*); *link*(*p*) ← *null*; *flush_node_list*(*p*)

This code is used in section 997.

1000. 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 1000⟩ ≡

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* 1001⟩
 else ⟨Prepare to move a box or rule node to the current page, then **goto** *contribute* 1002⟩;
whatsit_node: ⟨Prepare to move whatsit *p* to the current page, then **goto** *contribute* 1364⟩;
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* 1008⟩;
othercases *confusion*("page")
endcases

This code is used in section 997.

1001. ⟨Initialize the current page, insert the `\topskip` glue ahead of *p*, and **goto** *continue* 1001⟩ ≡
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 1000.

1002. ⟨Prepare to move a box or rule node to the current page, then **goto** *contribute* 1002⟩ ≡
begin *page_total* ← *page_total* + *page_depth* + *height*(*p*); *page_depth* ← *depth*(*p*); **goto** *contribute*;
end

This code is used in section 1000.

1003. ⟨Make sure that *page_max_depth* is not exceeded 1003⟩ ≡
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 997.

1004. ⟨Update the current page measurements with respect to the glue or kern specified by node p 1004⟩ ≡

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 997.

1005. ⟨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* 1005⟩ ≡

if *pi* < *inf_penalty* **then**
begin ⟨Compute the badness, b, of the current page, using *awful_bad* if the box is too full 1007⟩;
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 1006⟩;
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 997.

1006. ⟨Display the page break cost 1006⟩ ≡
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 1005.

1007. ⟨Compute the badness, b, of the current page, using *awful_bad* if the box is too full 1007⟩ ≡
if *page_total* < *page_goal* **then**
  if (*page_so_far*[3] ≠ 0) ∨ (*page_so_far*[4] ≠ 0) ∨ (*page_so_far*[5] ≠ 0) **then** $b \leftarrow 0$
  else $b \leftarrow$ *badness*(*page_goal* − *page_total*, *page_so_far*[2])
else if *page_total* − *page_goal* > *page_shrink* **then** $b \leftarrow$ *awful_bad*
  else $b \leftarrow$ *badness*(*page_total* − *page_goal*, *page_shrink*)

This code is used in section 1005.

1008. ⟨Append an insertion to the current page and **goto** *contribute* 1008⟩ ≡
begin if *page_contents* = *empty* **then** *freeze_page_specs*(*inserts_only*);
$n \leftarrow$ *subtype*(p); $r \leftarrow$ *page_ins_head*;
while $n \ge$ *subtype*(*link*(r)) **do** $r \leftarrow$ *link*(r);
$n \leftarrow$ *qo*(n);
if *subtype*(r) ≠ *qi*(n) **then** ⟨Create a page insertion node with *subtype*(r) = *qi*(n), and include the glue correction for box n in the current page state 1009⟩;
if *type*(r) = *split_up* **then** *insert_penalties* ← *insert_penalties* + *float_cost*(p)
else begin *last_ins_ptr*(r) ← p; *delta* ← *page_goal* − *page_total* − *page_depth* + *page_shrink*;
  { this much room is left if we shrink the maximum }
  if *count*(n) = 1000 **then** $h \leftarrow$ *height*(p)
  else $h \leftarrow$ *x_over_n*(*height*(p), 1000) ∗ *count*(n); { this much room is needed }
  if (($h \le 0$) ∨ ($h \le$ *delta*)) ∧ (*height*(p) + *height*(r) ≤ *dimen*(n)) **then**
    begin *page_goal* ← *page_goal* − h; *height*(r) ← *height*(r) + *height*(p);
    end
  else ⟨Find the best way to split the insertion, and change *type*(r) to *split_up* 1010⟩;
  end;
goto *contribute*;
end

This code is used in section 1000.

1009. 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 1009⟩ ≡

```
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 1008.

1010. 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* 1010⟩ ≡

```
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 1011⟩;
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 1008.

1011. ⟨ Display the insertion split cost 1011 ⟩ ≡

```
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 1010.

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

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

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

⟨ Declare the procedure called *fire_up* 1012 ⟩ ≡

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

This code is used in section 994.

1013. ⟨Set the value of *output_penalty* 1013⟩ ≡
  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 1012.

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

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

This code is used in section 1012.

1015. ⟨Ensure that box 255 is empty before output 1015⟩ ≡
  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 1014.

1016. ⟨Update the values of *first_mark* and *bot_mark* 1016⟩ ≡
  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 1014.

1017. 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 1017⟩ ≡

```
if p ≠ 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 991⟩; { this sets last_glue ← max_halfword }
if q ≠ hold_head then
  begin link(page_head) ← link(hold_head); page_tail ← q;
  end
```

This code is used in section 1014.

1018. 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 1018⟩ ≡

```
begin r ← link(page_ins_head);
while r ≠ 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 1014.

1019. ⟨Delete the page-insertion nodes 1019⟩ ≡

```
r ← link(page_ins_head);
while r ≠ 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 1014.

1020. 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 1020 ⟩ ≡

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 1021 ⟩
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 $node(p)$ 1022 ⟩;
end

This code is used in section 1014.

1021. ⟨ 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 1021 ⟩ ≡

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 1020.

1022. ⟨ Either append the insertion node p after node q, and remove it from the current page, or delete $node(p)$ 1022 ⟩ ≡

$link(prev\_p) \leftarrow link(p)$; $link(p) \leftarrow null$;
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 1020.

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

⟨Perform the default output routine 1023⟩ ≡
begin if *link*(*page_head*) $\neq$ *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 1012.

1024. ⟨Explain that too many dead cycles have occurred in a row 1024⟩ ≡
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 1012.

1025. ⟨Fire up the user's output routine and **return** 1025⟩ ≡
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 1012.

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

⟨Resume the page builder after an output routine has come to an end 1026⟩ ≡
begin if (*loc* $\neq$ *null*) $\lor$ ((*token_type* $\neq$ *output_text*) $\land$ (*token_type* $\neq$ *backed_up*)) **then**
  ⟨Recover from an unbalanced output routine 1027⟩;
end_token_list; { conserve stack space in case more outputs are triggered }
end_graf; *unsave*; *output_active* ← *false*; *insert_penalties* ← 0;
⟨Ensure that box 255 is empty after output 1028⟩;
if *tail* $\neq$ *head* **then** { current list goes after heldover insertions }
  begin *link*(*page_tail*) ← *link*(*head*); *page_tail* ← *tail*;
  end;
if *link*(*page_head*) $\neq$ *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 1100.

1027. ⟨Recover from an unbalanced output routine 1027⟩ ≡
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 1026.

1028. ⟨Ensure that box 255 is empty after output 1028⟩ ≡
if $box(255) \ne null$ **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 1026.

1029. 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.

1030. We shall concentrate first on the inner loop of *main_control*, deferring consideration of the other cases until later.

```
define big_switch = 60 { go here to branch on the next token of input }
define main_loop = 70 { go here to typeset a string of consecutive characters }
define main_loop_wrapup = 80 { go here to finish a character or ligature }
define main_loop_move = 90 { go here to advance the ligature cursor }
define main_loop_move_lig = 95 { same, when advancing past a generated ligature }
define main_loop_lookahead = 100 { go here to bring in another character, if any }
define main_lig_loop = 110 { go here to check for ligatures or kerning }
define append_normal_space = 120 { go here to append a normal space between words }
⟨Declare action procedures for use by main_control 1043⟩
⟨Declare the procedure called handle_right_brace 1068⟩
procedure main_control; { governs TeX's activities }
  label big_switch, reswitch, main_loop, main_loop_wrapup, main_loop_move, main_loop_move + 1,
        main_loop_move + 2, main_loop_move_lig, main_loop_lookahead, main_loop_lookahead + 1,
        main_lig_loop, main_lig_loop + 1, main_lig_loop + 2, append_normal_space, exit;
  var t: integer; { general-purpose temporary variable }
  begin if every_job ≠ null then begin_token_list(every_job, every_job_text);
big_switch: get_x_token;
reswitch: ⟨Give diagnostic information, if requested 1031⟩;
  case abs(mode) + cur_cmd of
  hmode + letter, hmode + other_char, hmode + char_given: goto main_loop;
  hmode + char_num: begin scan_char_num; cur_chr ← cur_val; goto main_loop; end;
  hmode + no_boundary: begin get_x_token;
    if (cur_cmd = letter) ∨ (cur_cmd = other_char) ∨ (cur_cmd = char_given) ∨ (cur_cmd = char_num)
          then cancel_boundary ← true;
    goto reswitch;
    end;
  hmode + spacer: if space_factor = 1000 then goto append_normal_space
    else app_space;
  hmode + ex_space, mmode + ex_space: goto append_normal_space;
  ⟨Cases of main_control that are not part of the inner loop 1045⟩
  end; { of the big case statement }
  goto big_switch;
main_loop: ⟨Append character cur_chr and the following characters (if any) to the current hlist in the
      current font; goto reswitch when a non-character has been fetched 1034⟩;
append_normal_space: ⟨Append a normal inter-word space to the current list, then goto big_switch 1041⟩;
exit: end;
```

1031. 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 1031⟩ ≡
  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 1030.

1032. The following part of the program was first written in a structured manner, according to the philosophy that "premature optimization is the root of all evil." Then it was rearranged into pieces of spaghetti so that the most common actions could proceed with little or no redundancy.

The original unoptimized form of this algorithm resembles the *reconstitute* procedure, which was described earlier in connection with hyphenation. Again we have an implied "cursor" between characters *cur_l* and *cur_r*. The main difference is that the *lig_stack* can now contain a charnode as well as pseudo-ligatures; that stack is now usually nonempty, because the next character of input (if any) has been appended to it. In *main_control* we have

$$cur\_r = \begin{cases} character(lig\_stack), & \text{if } lig\_stack > null; \\ font\_bchar[cur\_font], & \text{otherwise;} \end{cases}$$

except when $character(lig\_stack) = font\_false\_bchar[cur\_font]$. Several additional global variables are needed.

⟨ Global variables 13 ⟩ +≡
main_f: *internal_font_number*; { the current font }
main_i: *four_quarters*; { character information bytes for *cur_l* }
main_j: *four_quarters*; { ligature/kern command }
main_k: *font_index*; { index into *font_info* }
main_p: *pointer*; { temporary register for list manipulation }
main_s: *integer*; { space factor value }
bchar: *halfword*; { boundary character of current font, or *non_char* }
false_bchar: *halfword*; { nonexistent character matching *bchar*, or *non_char* }
cancel_boundary: *boolean*; { should the left boundary be ignored? }
ins_disc: *boolean*; { should we insert a discretionary node? }

1033. 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*;

1034. We leave the *space_factor* unchanged if *sf_code*(*cur_chr*) = 0; otherwise we set it equal to *sf_code*(*cur_chr*), except that it should never change from a value less than 1000 to a value exceeding 1000. The most common case is *sf_code*(*cur_chr*) = 1000, so we want that case to be fast.

The overall structure of the main loop is presented here. Some program labels are inside the individual sections.

```
define adjust_space_factor ≡
        main_s ← sf_code(cur_chr);
        if main_s = 1000 then space_factor ← 1000
        else if main_s < 1000 then
            begin if main_s > 0 then space_factor ← main_s;
            end
          else if space_factor < 1000 then space_factor ← 1000
            else space_factor ← main_s
```

⟨Append character *cur_chr* and the following characters (if any) to the current hlist in the current font; **goto** *reswitch* when a non-character has been fetched 1034⟩ ≡

```
  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 1035⟩;

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* 1036⟩;

main_loop_lookahead: ⟨Look ahead for another character, or leave *lig_stack* empty if there's none there 1038⟩;

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* 1039⟩;

main_loop_move_lig: ⟨Move the cursor past a pseudo-ligature, then **goto** *main_loop_lookahead* or *main_lig_loop* 1037⟩

This code is used in section 1030.

1035. 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 1035 ⟩ ≡
wrapup(*rt_hit*)

This code is used in section 1034.

1036. ⟨ 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* 1036 ⟩ ≡

```
  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 (cur_chr < font_bc[main_f]) ∨ (cur_chr > font_ec[main_f]) then
    begin char_warning(main_f, cur_chr); free_avail(lig_stack); goto big_switch;
    end;
  main_i ← char_info(main_f)(cur_l);
  if ¬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 1034.

1037. 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* 1037 ⟩ ≡
main_p ← *lig_ptr*(*lig_stack*);
if *main_p* > *null* **then** *tail_append*(*main_p*); { append a single character }
temp_ptr ← *lig_stack*; *lig_stack* ← *link*(*temp_ptr*); *free_node*(*temp_ptr*, *small_node_size*);
main_i ← *char_info*(*main_f*)(*cur_l*); *ligature_present* ← *true*;
if *lig_stack* = *null* **then**
  if *main_p* > *null* **then goto** *main_loop_lookahead*
  else *cur_r* ← *bchar*
else *cur_r* ← *character*(*lig_stack*);
goto *main_lig_loop*

This code is used in section 1034.

1038. 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 1038 ⟩ ≡
get_next; { set only *cur_cmd* and *cur_chr*, for speed }
if *cur_cmd* = *letter* **then goto** *main_loop_lookahead* + 1;
if *cur_cmd* = *other_char* **then goto** *main_loop_lookahead* + 1;
if *cur_cmd* = *char_given* **then goto** *main_loop_lookahead* + 1;
x_token; { now expand and set *cur_cmd*, *cur_chr*, *cur_tok* }
if *cur_cmd* = *letter* **then goto** *main_loop_lookahead* + 1;
if *cur_cmd* = *other_char* **then goto** *main_loop_lookahead* + 1;
if *cur_cmd* = *char_given* **then goto** *main_loop_lookahead* + 1;
if *cur_cmd* = *char_num* **then**
  begin *scan_char_num*; *cur_chr* ← *cur_val*; **goto** *main_loop_lookahead* + 1;
  end;
if *cur_cmd* = *no_boundary* **then** *bchar* ← *non_char*;
cur_r ← *bchar*; *lig_stack* ← *null*; **goto** *main_lig_loop*;
main_loop_lookahead + 1: *adjust_space_factor*; *fast_get_avail*(*lig_stack*); *font*(*lig_stack*) ← *main_f*;
cur_r ← *qi*(*cur_chr*); *character*(*lig_stack*) ← *cur_r*;
if *cur_r* = *false_bchar* **then** *cur_r* ← *non_char* { this prevents spurious ligatures }

This code is used in section 1034.

1039. 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* 1039⟩ ≡

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* 1040⟩;
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 1034.

1040. 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* 1040 ⟩ ≡

```
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 1039.

1041. 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* 1041⟩ ≡
  if *space_skip* = *zero_glue* **then**
    begin ⟨Find the glue specification, *main_p*, for text spaces in the current font 1042⟩;
    temp_ptr ← *new_glue*(*main_p*);
    end
  else *temp_ptr* ← *new_param_glue*(*space_skip_code*);
  link(*tail*) ← *temp_ptr*; *tail* ← *temp_ptr*; **goto** *big_switch*

This code is used in section 1030.

1042. 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 1042⟩ ≡
  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 1041 and 1043.

1043. ⟨Declare action procedures for use by *main_control* 1043⟩ ≡
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 1042⟩;
    main_p ← *new_spec*(*main_p*);
    ⟨Modify the glue specification in *main_p* according to the space factor 1044⟩;
    q ← *new_glue*(*main_p*); *glue_ref_count*(*main_p*) ← *null*;
    end;
  link(*tail*) ← *q*; *tail* ← *q*;
  end;

See also sections 1047, 1049, 1050, 1051, 1054, 1060, 1061, 1064, 1069, 1070, 1075, 1079, 1084, 1086, 1091, 1093, 1095, 1096, 1099, 1101, 1103, 1105, 1110, 1113, 1117, 1119, 1123, 1127, 1129, 1131, 1135, 1136, 1138, 1142, 1151, 1155, 1159, 1160, 1163, 1165, 1172, 1174, 1176, 1181, 1191, 1194, 1200, 1211, 1270, 1275, 1279, 1288, 1293, 1302, 1348, and 1376.

This code is used in section 1030.

1044. ⟨Modify the glue specification in *main_p* according to the space factor 1044⟩ ≡
  if *space_factor* ≥ 2000 **then** *width*(*main_p*) ← *width*(*main_p*) + *extra_space*(*cur_font*);
  stretch(*main_p*) ← *xn_over_d*(*stretch*(*main_p*), *space_factor*, 1000);
  shrink(*main_p*) ← *xn_over_d*(*shrink*(*main_p*), 1000, *space_factor*)

This code is used in section 1043.

1045. 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 1045 ⟩ ≡
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 406 ⟩;
goto *reswitch*;
end;
vmode + *stop*: **if** *its_all_over* **then return**; { this is the only way out }
⟨ Forbidden cases detected in *main_control* 1048 ⟩ *any_mode*(*mac_param*): *report_illegal_case*;
⟨ Math-only cases in non-math modes, or vice versa 1046 ⟩: *insert_dollar_sign*;
⟨ Cases of *main_control* that build boxes and lists 1056 ⟩
⟨ Cases of *main_control* that don't depend on *mode* 1210 ⟩
⟨ Cases of *main_control* that are for extensions to TEX 1347 ⟩

This code is used in section 1030.

1046. 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 1046 ⟩ ≡
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 1045.

1047. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
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;

1048. 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* 1048 ⟩ ≡
vmode + *vmove*, *hmode* + *hmove*, *mmode* + *hmove*, *any_mode*(*last_item*),

See also sections 1098, 1111, and 1144.

This code is used in section 1045.

1049. 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* 1043⟩ +≡
procedure *you_cant*;
  begin *print_err*("You␣can´t␣use␣`"); *print_cmd_chr*(*cur_cmd*, *cur_chr*); *print*("´␣in␣");
  print_mode(*mode*);
  end;

1050. ⟨Declare action procedures for use by *main_control* 1043⟩ +≡
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;

1051. 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* 1043⟩ +≡
function *privileged*: *boolean*;
  begin if $mode > 0$ **then** *privileged* ← *true*
  else begin *report_illegal_case*; *privileged* ← *false*;
    end;
  end;

1052. 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);

1053. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
stop: **if** $chr\_code = 1$ **then** *print_esc*("dump") **else** *print_esc*("end");

1054. 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* 1043⟩ +≡
function *its_all_over*: *boolean*; { do this when `\end` or `\dump` occurs }
  label *exit*;
  begin if *privileged* **then**
    begin if $(page\_head = page\_tail) \land (head = tail) \land (dead\_cycles = 0)$ **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**;

1055. 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.

1056. 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 1056 ⟩ ≡
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 1057, 1063, 1067, 1073, 1090, 1092, 1094, 1097, 1102, 1104, 1109, 1112, 1116, 1122, 1126, 1130, 1134, 1137, 1140, 1150, 1154, 1158, 1162, 1164, 1167, 1171, 1175, 1180, 1190, and 1193.

This code is used in section 1045.

1057. 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 1056 ⟩ +≡
vmode + *vskip*, *hmode* + *hskip*, *mmode* + *hskip*, *mmode* + *mskip*: *append_glue*;
any_mode(*kern*), *mmode* + *mkern*: *append_kern*;

1058. 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*);

1059. ⟨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");
```

1060. 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* 1043⟩ +≡

```
procedure append_glue;
  var s: small_number; { modifier of skip command }
  begin s ← cur_chr;
  case s of
  fil_code: cur_val ← fil_glue;
  fill_code: cur_val ← fill_glue;
  ss_code: cur_val ← ss_glue;
  fil_neg_code: cur_val ← fil_neg_glue;
  skip_code: scan_glue(glue_val);
  mskip_code: scan_glue(mu_val);
  end; { now cur_val points to the glue specification }
  tail_append(new_glue(cur_val));
  if s ≥ skip_code then
    begin decr(glue_ref_count(cur_val));
    if s > skip_code then subtype(tail) ← mu_glue;
    end;
  end;
```

1061. ⟨Declare action procedures for use by *main_control* 1043⟩ +≡

```
procedure append_kern;
  var s: quarterword; { subtype of the kern node }
  begin s ← cur_chr; scan_dimen(s = mu_glue, false, false); tail_append(new_kern(cur_val));
  subtype(tail) ← s;
  end;
```

1062. 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.

1063. 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 1056⟩ +≡
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*;

1064. 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* 1043⟩ +≡
procedure *off_save*;
  var *p*: *pointer*; { inserted token }
  begin if *cur_group* = *bottom_level* **then** ⟨Drop current token and complain that it was unmatched 1066⟩
  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 1065⟩;
    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;

1065. 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 1065 ⟩ ≡
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 1064.

1066. ⟨ Drop current token and complain that it was unmatched 1066 ⟩ ≡
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 1064.

1067. 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 1056 ⟩ +≡
any_mode(*right_brace*): *handle_right_brace*;

1068. ⟨ Declare the procedure called *handle_right_brace* 1068 ⟩ ≡
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 1085 ⟩
othercases *confusion*("rightbrace")
endcases;
end;

This code is used in section 1030.

1069. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
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;
```

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

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

```
procedure normal_paragraph;
  begin if looseness ≠ 0 then eq_word_define(int_base + looseness_code, 0);
  if hang_indent ≠ 0 then eq_word_define(dimen_base + hang_indent_code, 0);
  if hang_after ≠ 1 then eq_word_define(int_base + hang_after_code, 1);
  if par_shape_ptr ≠ null then eq_define(par_shape_loc, shape_ref, null);
  end;
```

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

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

The first problem is solved by putting a "context code" on the *save_stack*, just below the two entries that give the dimensions produced by *scan_spec*. The context code is either a (signed) shift amount, or it is a large integer $\geq$ *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 to 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*);

1072. ⟨ 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");

1073. 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 1056 ⟩ +≡
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);

1074. 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 }

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

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

1076. 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* 1076 ⟩ ≡

```
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 1075.

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

```
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 1075.

1078. ⟨ Append a new leader node that uses *cur_box* 1078 ⟩ ≡

```
begin ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
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 1075.

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

⟨Declare action procedures for use by *main_control* 1043⟩ +≡
procedure *begin_box*(*box_context* : *integer*);
  label *exit*, *done*;
  var *p*, *q*: *pointer*; { run through the current list }
    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* 1080⟩;
  vsplit_code: ⟨Split off part of a vertical box, make *cur_box* point to it 1082⟩;
  othercases ⟨Initiate the construction of an hbox or vbox, then **return** 1083⟩
  endcases;
  box_end(*box_context*); { in simple cases, we use the box immediately }
exit: **end**;

1080. Note that the condition ¬*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* 1080⟩ ≡
  begin *cur_box* ← *null*;
  if *abs*(*mode*) = *mmode* **then**
    begin *you_cant*; *help1*("Sorry;␣this␣\lastbox␣will␣be␣void."); *error*;
    end
  else if (*mode* = *vmode*) ∧ (*head* = *tail*) **then**
      begin *you_cant*; *help2*("Sorry...I␣usually␣can´t␣take␣things␣from␣the␣current␣page.")
      ("This␣\lastbox␣will␣therefore␣be␣void."); *error*;
      end
    else begin 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 1081⟩;
      end;
  end

This code is used in section 1079.

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

```
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 goto done;
      end;
  q ← link(p);
until q = tail;
cur_box ← tail; shift_amount(cur_box) ← 0; tail ← p; link(p) ← null;
done: end
```

This code is used in section 1080.

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

⟨Split off part of a vertical box, make *cur_box* point to it 1082⟩ ≡

```
begin scan_eight_bit_int; n ← cur_val;
if ¬scan_keyword("to") then
  begin print_err("Missing␣`to´␣inserted");
  help2("I´m␣working␣on␣`\vsplit<box␣number>␣to␣<dimen>´;")
  ("will␣look␣for␣the␣<dimen>␣next."); error;
  end;
scan_normal_dimen; cur_box ← vsplit(n, cur_val);
end
```

This code is used in section 1079.

1083. 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** 1083⟩ ≡

```
begin k ← cur_chr − vtop_code; saved(0) ← box_context;
if k = hmode then
  if (box_context < box_flag) ∧ (abs(mode) = vmode) then scan_spec(adjusted_hbox_group, true)
  else scan_spec(hbox_group, true)
else begin if k = vmode then scan_spec(vbox_group, true)
  else begin scan_spec(vtop_group, true); k ← vmode;
    end;
  normal_paragraph;
  end;
push_nest; mode ← −k;
if k = vmode then
  begin prev_depth ← ignore_depth;
  if every_vbox ≠ null then begin_token_list(every_vbox, every_vbox_text);
  end
else begin space_factor ← 1000;
  if every_hbox ≠ null then begin_token_list(every_hbox, every_hbox_text);
  end;
return;
end
```

This code is used in section 1079.

1084. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

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 404 ⟩;
  if *cur_cmd* = *make_box* **then** *begin_box*(*box_context*)
  else if (*box_context* ≥ *leader_flag*) ∧ ((*cur_cmd* = *hrule*) ∨ (*cur_cmd* = *vrule*)) **then**
      begin *cur_box* ← *scan_rule_spec*; *box_end*(*box_context*);
      end
    else begin
      print_err("A␣<box>␣was␣supposed␣to␣be␣here");
      help3("I␣was␣expecting␣to␣see␣\hbox␣or␣\vbox␣or␣\copy␣or␣\box␣or")
      ("something␣like␣that.␣So␣you␣might␣find␣something␣missing␣in")
      ("your␣output.␣But␣keep␣trying;␣you␣can␣fix␣this␣later."); *back_error*;
      end;
  end;

1085. 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 1085 ⟩ ≡
hbox_group: *package*(0);
adjusted_hbox_group: **begin** *adjust_tail* ← *adjust_head*; *package*(0);
  end;
vbox_group: **begin** *end_graf*; *package*(0);
  end;
vtop_group: **begin** *end_graf*; *package*(*vtop_code*);
  end;

See also sections 1100, 1118, 1132, 1133, 1168, 1173, and 1186.

This code is used in section 1068.

1086. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

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 1087 ⟩;
    end;
  pop_nest; *box_end*(*saved*(0));
  end;

1087. 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 1087 ⟩ ≡
  begin *h* ← 0; *p* ← *list_ptr*(*cur_box*);
  if *p* ≠ *null* **then**
    if *type*(*p*) ≤ *rule_node* **then** *h* ← *height*(*p*);
  depth(*cur_box*) ← *depth*(*cur_box*) − *h* + *height*(*cur_box*); *height*(*cur_box*) ← *h*;
  end

This code is used in section 1086.

1088. 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);

1089. ⟨ 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`");

1090. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
vmode + *start_par*: *new_graf* (*cur_chr* > 0);
vmode + *letter*, *vmode* + *other_char*, *vmode* + *char_num*, *vmode* + *char_given*, *vmode* + *math_shift*,
vmode + *un_hbox*, *vmode* + *vrule*, *vmode* + *accent*, *vmode* + *discretionary*, *vmode* + *hskip*,
vmode + *valign*, *vmode* + *ex_space*, *vmode* + *no_boundary*:
begin *back_input*; *new_graf* (*true*);
end;

1091. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
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*; **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;

1092. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
hmode + *start_par*, *mmode* + *start_par*: *indent_in_hmode*;

1093. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
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;

1094. 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 1056⟩ +≡
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*;

1095. ⟨Declare action procedures for use by *main_control* 1043⟩ +≡
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;

1096. ⟨Declare action procedures for use by *main_control* 1043⟩ +≡
procedure *end_graf*;
  begin if *mode* = *hmode* **then**
    begin if *head* = *tail* **then** *pop_nest* { null paragraphs are ignored }
    else *line_break*(*widow_penalty*);
    normal_paragraph; *error_count* ← 0;
    end;
  end;

1097. Insertion and adjustment and mark nodes are constructed by the following pieces of the program.

⟨Cases of *main_control* that build boxes and lists 1056⟩ +≡
any_mode(*insert*), *hmode* + *vadjust*, *mmode* + *vadjust*: *begin_insert_or_adjust*;
any_mode(*mark*): *make_mark*;

1098. ⟨Forbidden cases detected in *main_control* 1048⟩ +≡
  vmode + *vadjust*,

1099. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
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;

1100. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1085 ⟩ +≡
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 1026 ⟩;

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

1102. Penalty nodes get into a list via the *break_penalty* command.

⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
any_mode(*break_penalty*): *append_penalty*;

1103. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
procedure *append_penalty*;
  begin *scan_int*; *tail_append*(*new_penalty*(*cur_val*));
  if *mode* = *vmode* **then** *build_page*;
  end;

1104. 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 1056 ⟩ +≡
any_mode(*remove_item*): *delete_last*;

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

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

```
procedure delete_last;
  label exit;
  var p, q: pointer;   { run through the current list }
    m: quarterword;   { the length of a replacement list }
  begin if (mode = vmode) ∧ (tail = head) then
    ⟨ Apologize for inability to do the operation now, unless \unskip follows non-glue 1106 ⟩
  else begin 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;
```

1106. ⟨ Apologize for inability to do the operation now, unless `\unskip` follows non-glue 1106 ⟩ ≡

```
  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 1105.

1107. ⟨ 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);
```

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

```
remove_item: if chr_code = glue_node then print_esc("unskip")
  else if chr_code = kern_node then print_esc("unkern")
    else print_esc("unpenalty");
un_hbox: if chr_code = copy_code then print_esc("unhcopy")
  else print_esc("unhbox");
un_vbox: if chr_code = copy_code then print_esc("unvcopy")
  else print_esc("unvbox");
```

1109. The *un_hbox* and *un_vbox* commands unwrap one of the 256 current boxes.

⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡

```
vmode + un_vbox, hmode + un_hbox, mmode + un_hbox: unpackage;
```

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

```
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;
```

1111. ⟨ Forbidden cases detected in *main_control* 1048 ⟩ +≡

```
  vmode + ital_corr,
```

1112. 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 1056 ⟩ +≡

```
hmode + ital_corr: append_italic_correction;
mmode + ital_corr: tail_append(new_kern(0));
```

1113. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
procedure *append_italic_correction*;
  label *exit*;
  var *p*: *pointer*; { *char_node* at the tail of the current list }
    f: *internal_font_number*; { the font in the *char_node* }
  begin if *tail* ≠ *head* **then**
    begin if *is_char_node*(*tail*) **then** *p* ← *tail*
    else if *type*(*tail*) = *ligature_node* **then** *p* ← *lig_char*(*tail*)
      else return;
    f ← *font*(*p*); *tail_append*(*new_kern*(*char_italic*(*f*)(*char_info*(*f*)(*character*(*p*)))));
    subtype(*tail*) ← *explicit*;
    end;
exit: **end**;

1114. 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);

1115. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
discretionary: **if** *chr_code* = 1 **then** *print_esc*("`-`") **else** *print_esc*("`discretionary`");

1116. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
hmode + *discretionary*, *mmode* + *discretionary*: *append_discretionary*;

1117. 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* 1043 ⟩ +≡
procedure *append_discretionary*;
  var *c*: *integer*; { hyphen character }
  begin *tail_append*(*new_disc*);
  if *cur_chr* = 1 **then**
    begin *c* ← *hyphen_char*[*cur_font*];
    if *c* ≥ 0 **then**
      if *c* < 256 **then** *pre_break*(*tail*) ← *new_character*(*cur_font*, *c*);
    end
  else begin *incr*(*save_ptr*); *saved*(−1) ← 0; *new_save_level*(*disc_group*); *scan_left_brace*; *push_nest*;
    mode ← −*hmode*; *space_factor* ← 1000;
    end;
  end;

1118. 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 1085 ⟩ +≡
disc_group: *build_discretionary*;

1119. ⟨Declare action procedures for use by *main_control* 1043⟩ +≡

```
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 1121⟩;
  p ← link(head); pop_nest;
  case saved(−1) of
  0: pre_break(tail) ← p;
  1: post_break(tail) ← p;
  2: ⟨Attach list p to the current list, and record its length; then finish up and return 1120⟩;
  end; { there are no other cases }
  incr(saved(−1)); new_save_level(disc_group); scan_left_brace; push_nest; mode ← −hmode;
  space_factor ← 1000;
exit: end;
```

1120. ⟨Attach list *p* to the current list, and record its length; then finish up and **return** 1120⟩ ≡

```
  begin if (n > 0) ∧ (abs(mode) = mmode) then
    begin print_err("Illegal␣math␣"); print_esc("discretionary");
    help2("Sorry:␣The␣third␣part␣of␣a␣discretionary␣break␣must␣be")
    ("empty,␣in␣math␣formulas.␣I␣had␣to␣delete␣your␣third␣part."); flush_node_list(p); n ← 0;
    error;
    end
  else link(tail) ← p;
  if n ≤ max_quarterword then replace_count(tail) ← n
  else begin print_err("Discretionary␣list␣is␣too␣long");
    help2("Wow---I␣never␣thought␣anybody␣would␣tweak␣me␣here.")
    ("You␣can´t␣seriously␣need␣such␣a␣huge␣discretionary␣list?"); error;
    end;
  if n > 0 then tail ← q;
  decr(save_ptr); return;
  end
```

This code is used in section 1119.

1121. 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 1121 ⟩ ≡

```
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 1119.

1122. 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 1056 ⟩ +≡
hmode + *accent*: *make_accent*;

1123. 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* 1043 ⟩ +≡

```
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 1124 ⟩;
    if q ≠ null then ⟨ Append the accent with appropriate kerns, then set p ← q 1125 ⟩;
    link(tail) ← p; tail ← p; space_factor ← 1000;
    end;
  end;
```

1124. ⟨ Create a character node q for the next character, but set $q \leftarrow null$ if problems arise 1124 ⟩ ≡

$q \leftarrow null$; $f \leftarrow cur\_font$;
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 1123.

1125. 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 $p \leftarrow q$ 1125 ⟩ ≡

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 1123.

1126. 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 1056 ⟩ +≡

$any\_mode(car\_ret), any\_mode(tab\_mark)$: $align\_error$;
$any\_mode(no\_align)$: no_align_error;
$any\_mode(omit)$: $omit\_error$;

1127. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
procedure align_error;
  begin if abs(align_state) > 2 then
    ⟨ Express consternation over the fact that no alignment is in progress 1128 ⟩
  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;
```

1128. ⟨ Express consternation over the fact that no alignment is in progress 1128 ⟩ ≡

```
  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 1127.

1129. 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* 1043 ⟩ +≡

```
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;
```

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

⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
vmode + *halign*, *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*;

1131. 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* 1043 ⟩ +≡
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;

1132. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1085 ⟩ +≡
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;

1133. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1085 ⟩ +≡
no_align_group: **begin** *end_graf*; *unsave*; *align_peek*;
 end;

1134. Finally, `\endcsname` is not supposed to get through to *main_control*.

⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
any_mode(*end_cs_name*): *cs_error*;

1135. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
procedure *cs_error*;
 begin *print_err*("Extra␣"); *print_esc*("endcsname");
 help1("I´m␣ignoring␣this,␣since␣I␣wasn´t␣doing␣a␣\csname."); *error*;
 end;

1136. 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* 1043 ⟩ +≡
procedure *push_math*(*c* : *group_code*);
  begin *push_nest*; *mode* ← −*mmode*; *incompleat_noad* ← *null*; *new_save_level*(*c*);
  end;

1137. 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 1056 ⟩ +≡
hmode + *math_shift*: *init_math*;

1138. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
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 1145 ⟩
  else begin *back_input*; ⟨ Go into ordinary math mode 1139 ⟩;
    end;
  end;

1139. ⟨ Go into ordinary math mode 1139 ⟩ ≡
  begin *push_math*(*math_shift_group*); *eq_word_define*(*int_base* + *cur_fam_code*, −1);
  if *every_math* ≠ *null* **then** *begin_token_list*(*every_math*, *every_math_text*);
  end

This code is used in sections 1138 and 1142.

1140. 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 1056 ⟩ +≡
mmode + *eq_no*: **if** *privileged* **then**
    if *cur_group* = *math_shift_group* **then** *start_eq_no*
    else *off_save*;

1141. ⟨ Put each of TeX's primitives into the hash table 226 ⟩ +≡
  primitive("`eqno`", *eq_no*, 0); *primitive*("`leqno`", *eq_no*, 1);

1142. 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* 1043⟩ +≡
procedure *start_eq_no*;
  begin *saved*(0) ← *cur_chr*; *incr*(*save_ptr*); ⟨Go into ordinary math mode 1139⟩;
  end;

1143. ⟨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");

1144. ⟨Forbidden cases detected in *main_control* 1048⟩ +≡
  non_math(*eq_no*),

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

⟨Go into display math mode 1145⟩ ≡
  begin 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 1146⟩;
    end; {now we are in vertical mode, working on the list that will contain the display}
  ⟨Calculate the length, *l*, and the shift amount, *s*, of the display lines 1149⟩;
  push_math(*math_shift_group*); *mode* ← *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 1138.

1146. ⟨Calculate the natural width, *w*, by which the characters of the final line extend to the right of the reference point, plus two ems; or set *w* ← *max_dimen* if the non-blank information on that line is affected by stretching or shrinking 1146⟩ ≡
  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* 1147⟩;
    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 1145.

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

```
reswitch: if is_char_node(p) then
    begin f ← font(p); d ← char_width(f)(char_info(f)(character(p))); goto found;
    end;
  case type(p) of
  hlist_node, vlist_node, rule_node: begin d ← width(p); goto found;
    end;
  ligature_node: ⟨Make node p look like a char_node and goto reswitch 652⟩;
  kern_node, math_node: d ← width(p);
  glue_node: ⟨Let d be the natural width of this glue; if stretching or shrinking, set v ← max_dimen; goto
      found in the case of leaders 1148⟩;
  whatsit_node: ⟨Let d be the width of the whatsit p 1361⟩;
  othercases d ← 0
  endcases
```

This code is used in section 1146.

1148. 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 1148⟩ ≡

```
  begin q ← glue_ptr(p); d ← width(q);
  if glue_sign(just_box) = stretching then
    begin if (glue_order(just_box) = stretch_order(q)) ∧ (stretch(q) ≠ 0) then v ← max_dimen;
    end
  else if glue_sign(just_box) = shrinking then
      begin if (glue_order(just_box) = shrink_order(q)) ∧ (shrink(q) ≠ 0) then v ← max_dimen;
      end;
  if subtype(p) ≥ a_leaders then goto found;
  end
```

This code is used in section 1147.

1149. A displayed equation is considered to be three lines long, so we calculate the length and offset of line number $prev\_graf + 2$.

⟨Calculate the length, l, and the shift amount, s, of the display lines 1149⟩ ≡

```
  if par_shape_ptr = null then
    if (hang_indent ≠ 0) ∧ (((hang_after ≥ 0) ∧ (prev_graf + 2 > hang_after)) ∨
          (prev_graf + 1 < −hang_after)) then
      begin l ← hsize − abs(hang_indent);
      if hang_indent > 0 then s ← hang_indent else s ← 0;
      end
    else begin l ← hsize; s ← 0;
      end
  else begin n ← info(par_shape_ptr);
    if prev_graf + 2 ≥ n then p ← par_shape_ptr + 2 ∗ n
    else p ← par_shape_ptr + 2 ∗ (prev_graf + 2);
    s ← mem[p − 1].sc; l ← mem[p].sc;
    end
```

This code is used in section 1145.

1150. 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.

⟨Cases of *main_control* that build boxes and lists 1056⟩ +≡
mmode + *left_brace*: **begin** *tail_append*(*new_noad*); *back_input*; *scan_math*(*nucleus*(*tail*));
  end;

1151. 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* 1043⟩ +≡
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 404⟩;
reswitch: **case** *cur_cmd* **of**
  letter, *other_char*, *char_given*: **begin** *c* ← *ho*(*math_code*(*cur_chr*));
    if *c* = ´*100000* **then**
      begin ⟨Treat *cur_chr* as an active character 1152⟩;
      goto *restart*;
      end;
    end;
  char_num: **begin** *scan_char_num*; *cur_chr* ← *cur_val*; *cur_cmd* ← *char_given*; **goto** *reswitch*;
    end;
  math_char_num: **begin** *scan_fifteen_bit_int*; *c* ← *cur_val*;
    end;
  math_given: *c* ← *cur_chr*;
  delim_num: **begin** *scan_twenty_seven_bit_int*; *c* ← *cur_val* **div** ´*10000*;
    end;
  othercases ⟨Scan a subformula enclosed in braces and **return** 1153⟩
  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**;

1152. An active character that is an *outer_call* is allowed here.

⟨Treat *cur_chr* as an active character 1152⟩ ≡
  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 1151 and 1155.

1153. The pointer p is placed on *save_stack* while a complex subformula is being scanned.

⟨Scan a subformula enclosed in braces and **return** 1153⟩ ≡
  begin *back_input*; *scan_left_brace*;
  saved(0) ← *p*; *incr*(*save_ptr*); *push_math*(*math_group*); **return**;
  end

This code is used in section 1151.

1154. 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 1056⟩ +≡
mmode + *letter*, *mmode* + *other_char*, *mmode* + *char_given*: *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;

1155. 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* 1043⟩ +≡
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 1152⟩
  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;

1156. 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*);

1157. ⟨ 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");

1158. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
mmode + *math_comp*: **begin** *tail_append*(*new_noad*); *type*(*tail*) ← *cur_chr*; *scan_math*(*nucleus*(*tail*));
  end;
mmode + *limit_switch*: *math_limit_switch*;

1159. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
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**;

1160. 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* 1043 ⟩ +≡
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 404 ⟩;
    case *cur_cmd* **of**
    letter, *other_char*: *cur_val* ← *del_code*(*cur_chr*);
    delim_num: *scan_twenty_seven_bit_int*;
    othercases *cur_val* ← −1
    endcases;
    end;
  if *cur_val* < 0 **then**
    ⟨ Report that an invalid delimiter code is being changed to null; set *cur_val* ← 0 1161 ⟩;
  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;

1161. ⟨ Report that an invalid delimiter code is being changed to null; set *cur_val* ← 0 1161 ⟩ ≡

```
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 1160.

1162. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡

```
mmode + radical: math_radical;
```

1163. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
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;
```

1164. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡

```
mmode + accent, mmode + math_accent: math_ac;
```

1165. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
procedure math_ac;
  begin if cur_cmd = accent then ⟨ Complain that the user should have said \mathaccent 1166 ⟩;
  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;
```

1166. ⟨ Complain that the user should have said `\mathaccent` 1166 ⟩ ≡

```
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 1165.

1167. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡

```
mmode + vcenter: begin scan_spec(vcenter_group, false); normal_paragraph; push_nest; mode ← −vmode;
  prev_depth ← ignore_depth;
  if every_vbox ≠ null then begin_token_list(every_vbox, every_vbox_text);
  end;
```

1168. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1085 ⟩ +≡
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;

1169. 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*);

1170. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
math_style: *print_style*(*chr_code*);

1171. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
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*;

1172. 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* 1043 ⟩ +≡
procedure *append_choices*;
begin *tail_append*(*new_choice*); *incr*(*save_ptr*); *saved*(−1) ← 0; *push_math*(*math_choice_group*);
scan_left_brace;
end;

1173. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1085 ⟩ +≡
math_choice_group: *build_choices*;

1174. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
⟨ Declare the function called *fin_mlist* 1184 ⟩
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**;

1175. 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 1056 ⟩ +≡
mmode + *sub_mark*, *mmode* + *sup_mark*: *sub_sup*;

1176. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
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 1177 ⟩;
  scan_math(*p*);
  end;

1177. ⟨ Insert a dummy noad to be sub/superscripted 1177 ⟩ ≡
  begin *tail_append*(*new_noad*); *p* ← *supscr*(*tail*) + *cur_cmd* − *sup_mark*; { *supscr* or *subscr* }
  if *t* ≠ *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 1176.

1178. 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*);

1179. ⟨ 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;

1180. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
mmode + *above*: *math_fraction*;

1181. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
procedure *math_fraction*;
var *c*: *small_number*; { the type of generalized fraction we are scanning }
begin $c \leftarrow$ *cur_chr*;
if *incompleat_noad* ≠ *null* **then**
⟨ Ignore the fraction operation and complain about this ambiguous case 1183 ⟩
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 1182 ⟩;
end;
end;

1182. ⟨ Use code *c* to distinguish between generalized fractions 1182 ⟩ ≡
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 1181.

1183. ⟨ Ignore the fraction operation and complain about this ambiguous case 1183 ⟩ ≡

```
begin if c ≥ delimited_code then
  begin scan_delimiter(garbage, false); scan_delimiter(garbage, false);
  end;
if c mod delimited_code = above_code then scan_normal_dimen;
print_err("Ambiguous;␣you␣need␣another␣{␣and␣}");
help3("I´m␣ignoring␣this␣fraction␣specification,␣since␣I␣don´t")
("know␣whether␣a␣construction␣like␣`x␣\over␣y␣\over␣z´")
("means␣`{x␣\over␣y}␣\over␣z´␣or␣`x␣\over␣{y␣\over␣z}´."); error;
end
```

This code is used in section 1181.

1184. 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* 1184 ⟩ ≡

```
function fin_mlist(p : pointer): pointer;
  var q: pointer; { the mlist to return }
  begin if incompleat_noad ≠ null then ⟨ Compleat the incompleat noad 1185 ⟩
  else begin link(tail) ← p; q ← link(head);
    end;
  pop_nest; fin_mlist ← q;
  end;
```

This code is used in section 1174.

1185. ⟨ Compleat the incompleat noad 1185 ⟩ ≡

```
begin math_type(denominator(incompleat_noad)) ← sub_mlist;
info(denominator(incompleat_noad)) ← link(head);
if p = null then q ← incompleat_noad
else begin q ← info(numerator(incompleat_noad));
  if type(q) ≠ left_noad then confusion("right");
  info(numerator(incompleat_noad)) ← link(q); link(q) ← incompleat_noad; link(incompleat_noad) ← p;
  end;
end
```

This code is used in section 1184.

1186. 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 1085 ⟩ +≡
math_group: **begin** *unsave*; *decr*(*save_ptr*);
  math_type(*saved*(0)) ← *sub_mlist*; *p* ← *fin_mlist*(*null*); *info*(*saved*(0)) ← *p*;
  if *p* ≠ *null* **then**
    if *link*(*p*) = *null* **then**
      if *type*(*p*) = *ord_noad* **then**
        begin if *math_type*(*subscr*(*p*)) = *empty* **then**
          if *math_type*(*supscr*(*p*)) = *empty* **then**
            begin *mem*[*saved*(0)].*hh* ← *mem*[*nucleus*(*p*)].*hh*; *free_node*(*p*, *noad_size*);
            end;
        end
      else if *type*(*p*) = *accent_noad* **then**
          if *saved*(0) = *nucleus*(*tail*) **then**
            if *type*(*tail*) = *ord_noad* **then** ⟨ Replace the tail of the list by *p* 1187 ⟩;
  end;

1187. ⟨ Replace the tail of the list by *p* 1187 ⟩ ≡
  begin *q* ← *head*;
  while *link*(*q*) ≠ *tail* **do** *q* ← *link*(*q*);
  link(*q*) ← *p*; *free_node*(*tail*, *noad_size*); *tail* ← *p*;
  end

This code is used in section 1186.

1188. 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*];

1189. ⟨ 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`");

1190. ⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
mmode + *left_right*: *math_left_right*;

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

procedure *math_left_right*;
  var *t*: *small_number*; { *left_noad* or *right_noad* }
    p: *pointer*; { new noad }
  begin $t \leftarrow$ *cur_chr*;
  if $(t = right\_noad) \land (cur\_group \neq math\_left\_group)$ **then** ⟨ Try to recover from mismatched `\right` 1192 ⟩
  else begin $p \leftarrow$ *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;

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

  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 1191.

1193. Here is the only way out of math mode.

⟨ Cases of *main_control* that build boxes and lists 1056 ⟩ +≡
mmode + *math_shift*: **if** *cur_group* = *math_shift_group* **then** *after_math*
  else *off_save*;

1194. ⟨Declare action procedures for use by *main_control* 1043⟩ +≡

procedure *after_math*;
  var *l*: *boolean*; {'\leqno' instead of '\eqno'}
    danger: *boolean*; {not enough symbol fonts are present}
    m: *integer*; {*mmode* or −*mmode*}
    p: *pointer*; {the formula}
    a: *pointer*; {box containing equation number}
    ⟨Local variables for finishing a displayed formula 1198⟩
  begin *danger* ← *false*; ⟨Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1195⟩;
  m ← *mode*; *l* ← *false*; *p* ← *fin_mlist*(*null*); {this pops the nest}
  if *mode* = −*m* **then** {end of equation number}
    begin ⟨Check that another $ follows 1197⟩;
    cur_mlist ← *p*; *cur_style* ← *text_style*; *mlist_penalties* ← *false*; *mlist_to_hlist*;
    a ← *hpack*(*link*(*temp_head*), *natural*); *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* 1195⟩;
    m ← *mode*; *p* ← *fin_mlist*(*null*);
    end
  else *a* ← *null*;
  if *m* < 0 **then** ⟨Finish math in text 1196⟩
  else begin if *a* = *null* **then** ⟨Check that another $ follows 1197⟩;
    ⟨Finish displayed math 1199⟩;
    end;
  end;

1195. ⟨Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1195⟩ ≡

  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 1194 and 1194.

1196. 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 1196⟩ ≡
  begin *tail_append*(*new_math*(*math_surround*, *before*)); *cur_mlist* ← *p*; *cur_style* ← *text_style*;
  mlist_penalties ← (*mode* > 0); *mlist_to_hlist*; *link*(*tail*) ← *link*(*temp_head*);
  while *link*(*tail*) ≠ *null* **do** *tail* ← *link*(*tail*);
  tail_append(*new_math*(*math_surround*, *after*)); *space_factor* ← 1000; *unsave*;
  end

This code is used in section 1194.

1197. TEX gets to the following part of the program when the first '`$`' ending a display has been scanned.

⟨Check that another `$` follows 1197⟩ ≡
  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 1194, 1194, and 1206.

1198. 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 1198⟩ ≡
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 1194.

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

⟨ Finish displayed math 1199 ⟩ ≡
cur_mlist ← 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 $e \leftarrow 0$ 1201 ⟩;
⟨ Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming that l = *false* 1202 ⟩;
⟨ Append the glue or equation number preceding the display 1203 ⟩;
⟨ Append the display and perhaps also the equation number 1204 ⟩;
⟨ Append the glue or equation number following the display 1205 ⟩;
resume_after_display

This code is used in section 1194.

1200. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
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 443 ⟩;
if *nest_ptr* = 1 **then** *build_page*;
end;

1201. 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 $e \leftarrow 0$ 1201 ⟩ ≡
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 1199.

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

⟨Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming that $l = false$ 1202⟩ ≡

```
d ← half(z − w);
if (e > 0) ∧ (d < 2 ∗ e) then {too close}
  begin d ← half(z − w − e);
  if p ≠ null then
    if ¬is_char_node(p) then
      if type(p) = glue_node then d ← 0;
  end
```

This code is used in section 1199.

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

⟨Append the glue or equation number preceding the display 1203⟩ ≡

```
tail_append(new_penalty(pre_display_penalty));
if (d + s ≤ pre_display_size) ∨ l then {not enough clearance}
  begin g1 ← above_display_skip_code; g2 ← below_display_skip_code;
  end
else begin g1 ← above_display_short_skip_code; g2 ← below_display_short_skip_code;
  end;
if l ∧ (e = 0) then {it follows that type(a) = hlist_node}
  begin shift_amount(a) ← 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 1199.

1204. ⟨Append the display and perhaps also the equation number 1204⟩ ≡

```
if e ≠ 0 then
  begin r ← new_kern(z − w − e − d);
  if l then
    begin link(a) ← r; link(r) ← b; b ← a; d ← 0;
    end
  else begin link(b) ← r; link(r) ← a;
    end;
  b ← hpack(b, natural);
  end;
shift_amount(b) ← s + d; append_to_vlist(b)
```

This code is used in section 1199.

1205. ⟨ Append the glue or equation number following the display 1205 ⟩ ≡

if $(a \neq null) \land (e = 0) \land \neg l$ **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 1199.

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

⟨ Finish an alignment in a display 1206 ⟩ ≡

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

This code is used in section 812.

1207. ⟨ Pontificate about improper alignment in display 1207 ⟩ ≡

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 1206.

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

Assignments to values in *eqtb* can be global or local. Furthermore, a control sequence can be defined to be '`\long`' 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);

1209. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 227⟩ +≡
prefix: **if** *chr_code* = 1 **then** *print_esc*("`long`")
  else if *chr_code* = 2 **then** *print_esc*("`outer`")
    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`");

1210. 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* 1210⟩ ≡
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*(*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 1268, 1271, 1274, 1276, 1285, and 1290.

This code is used in section 1045.

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

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

1212. ⟨ Discard erroneous prefixes and **return** 1212 ⟩ ≡
  begin *print_err*("You␣can´t␣use␣a␣prefix␣with␣`"); *print_cmd_chr*(*cur_cmd*, *cur_chr*);
  print_char("´"); *help1*("I´ll␣pretend␣you␣didn´t␣say␣\long␣or␣\outer␣or␣\global.");
  back_error; **return**;
  end

This code is used in section 1211.

1213. ⟨ Discard the prefixes \long and \outer if they are irrelevant 1213 ⟩ ≡
  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 1211.

1214. The previous routine does not have to adjust a so that $a \bmod 4 = 0$, since the following routines test for the `\global` prefix as follows.

define *global* $\equiv (a \ge 4)$
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` 1214⟩ ≡
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 1211.

1215. 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* 1215⟩ ≡
procedure *get_r_token*;
  label *restart*;
  begin *restart*: **repeat** *get_token*;
  until *cur_tok* $\neq$ *space_token*;
  if $(cur\_cs = 0) \lor (cur\_cs > frozen\_control\_sequence)$ **then**
    begin *print_err*("Missing␣control␣sequence␣inserted");
    help5("Please␣don´t␣say␣`\def␣cs{...}´,␣say␣`\def\cs{...}´.")
    ("I´ve␣inserted␣an␣inaccessible␣control␣sequence␣so␣that␣your")
    ("definition␣will␣be␣completed␣without␣mixing␣me␣up␣too␣badly.")
    ("You␣can␣recover␣graciously␣from␣this␣error,␣if␣you´re")
    ("careful;␣see␣exercise␣27.2␣in␣The␣TeXbook.");
    if $cur\_cs = 0$ **then** *back_input*;
    $cur\_tok \leftarrow cs\_token\_flag + frozen\_protection$; *ins_error*; **goto** *restart*;
    end;
  end;

See also sections 1229, 1236, 1243, 1244, 1245, 1246, 1247, 1257, and 1265.

This code is used in section 1211.

1216. ⟨Initialize table entries (done by `INITEX` only) 164⟩ +≡
text(*frozen_protection*) ← "inaccessible";

1217. Here's an example of the way many of the following routines operate. (Unfortunately, they aren't all as simple as this.)

⟨Assignments 1217⟩ ≡
set_font: *define*(*cur_font_loc*, *data*, *cur_chr*);

See also sections 1218, 1221, 1224, 1225, 1226, 1228, 1232, 1234, 1235, 1241, 1242, 1248, 1252, 1253, 1256, and 1264.

This code is used in section 1211.

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

⟨ Assignments 1217 ⟩ +≡
def: **begin if** $odd(cur\_chr) \land \neg global \land (global\_defs \ge 0)$ **then** $a \leftarrow a + 4$;
  $e \leftarrow (cur\_chr \ge 2)$; *get_r_token*; $p \leftarrow cur\_cs$; $q \leftarrow scan\_toks(true, e)$; $define(p, call + (a \bmod 4), def\_ref)$;
  end;

1219. 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);

1220. ⟨ 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`");

1221. ⟨ Assignments 1217 ⟩ +≡
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;

1222. 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` }

⟨ 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*);

1223. ⟨ 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");
  othercases print_esc("toksdef")
  endcases;
char_given: begin print_esc("char"); print_hex(chr_code);
  end;
math_given: begin print_esc("mathchar"); print_hex(chr_code);
  end;
```

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

⟨ Assignments 1217 ⟩ +≡

```
shorthand_def: begin n ← cur_chr; get_r_token; p ← cur_cs; define(p, relax, 256); scan_optional_equals;
  case n of
  char_def_code: begin scan_char_num; define(p, char_given, cur_val);
    end;
  math_char_def_code: begin scan_fifteen_bit_int; define(p, math_given, cur_val);
    end;
  othercases begin scan_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;
```

1225. ⟨ Assignments 1217 ⟩ +≡

```
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;
```

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

⟨ Assignments 1217 ⟩ +≡
toks_register, *assign_toks*: **begin** $q \leftarrow cur\_cs$;
  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 404 ⟩;
  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* 1227 ⟩;
  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;

1227. ⟨ If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1227 ⟩ ≡
  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 1226.

1228. Similar routines are used to assign values to the numeric parameters.

⟨ Assignments 1217 ⟩ +≡
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;

1229. 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* 1215 ⟩ +≡
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;

1230. 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*("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*);

1231. ⟨ 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* = *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*);

1232. The different types of code values have different legal ranges; the following program is careful to check each case properly.

⟨ Assignments 1217 ⟩ +≡
def_code: **begin** ⟨ Let *n* be the largest legal code value, based on *cur_chr* 1233 ⟩;
  p ← *cur_chr*; *scan_char_num*; *p* ← *p* + *cur_val*; *scan_optional_equals*; *scan_int*;
  if ((*cur_val* < 0) ∧ (*p* < *del_code_base*)) ∨ (*cur_val* > *n*) **then**
    begin *print_err*("Invalid␣code␣("); *print_int*(*cur_val*);
    if *p* < *del_code_base* **then** *print*("),␣should␣be␣in␣the␣range␣0..")
    else *print*("),␣should␣be␣at␣most␣");
    print_int(*n*); *help1*("I´m␣going␣to␣use␣0␣instead␣of␣that␣illegal␣code␣value.");
    error; *cur_val* ← 0;
    end;
  if *p* < *math_code_base* **then** *define*(*p*, *data*, *cur_val*)
  else if *p* < *del_code_base* **then** *define*(*p*, *data*, *hi*(*cur_val*))
    else *word_define*(*p*, *cur_val*);
  end;

1233. ⟨ Let n be the largest legal code value, based on *cur_chr* 1233 ⟩ ≡

if *cur_chr* = *cat_code_base* **then** $n \leftarrow$ *max_char_code*
else if *cur_chr* = *math_code_base* **then** $n \leftarrow$ ´100000
else if *cur_chr* = *sf_code_base* **then** $n \leftarrow$ ´77777
else if *cur_chr* = *del_code_base* **then** $n \leftarrow$ ´77777777
else $n \leftarrow 255$

This code is used in section 1232.

1234. ⟨ Assignments 1217 ⟩ +≡

def_family: **begin** $p \leftarrow$ *cur_chr*; *scan_four_bit_int*; $p \leftarrow p +$ *cur_val*; *scan_optional_equals*; *scan_font_ident*;
define(*p*, *data*, *cur_val*);
end;

1235. Next we consider changes to TEX's numeric registers.

⟨ Assignments 1217 ⟩ +≡

register, *advance*, *multiply*, *divide*: *do_register_command*(*a*);

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

⟨ Declare subprocedures for *prefixed_command* 1215 ⟩ +≡

procedure *do_register_command*(*a* : *small_number*);
label *found*, *exit*;
var *l*, *q*, *r*, *s*: *pointer*; { for list manipulation }
p: *int_val* .. *mu_val*; { type of register involved }
begin $q \leftarrow$ *cur_cmd*; ⟨ Compute the register location l and its type p; but **return** if invalid 1237 ⟩;
if $q =$ *register* **then** *scan_optional_equals*
else if *scan_keyword*("by") **then** *do_nothing*; { optional 'by' }
arith_error ← *false*;
if $q <$ *multiply* **then** ⟨ Compute result of *register* or *advance*, put it in *cur_val* 1238 ⟩
else ⟨ Compute result of *multiply* or *divide*, put it in *cur_val* 1240 ⟩;
if *arith_error* **then**
begin *print_err*("Arithmetic␣overflow");
help2("I␣can´t␣carry␣out␣that␣multiplication␣or␣division,")
("since␣the␣result␣is␣out␣of␣range.");
if $p \geq$ *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**;

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

⟨Compute the register location l and its type p; but **return** if invalid 1237⟩ ≡
  begin if $q \neq register$ **then**
    begin *get_x_token*;
    if $(cur\_cmd \geq assign\_int) \land (cur\_cmd \leq assign\_mu\_glue)$ **then**
      begin $l \leftarrow cur\_chr$; $p \leftarrow cur\_cmd - assign\_int$; **goto** *found*;
      end;
    if $cur\_cmd \neq register$ **then**
      begin *print_err*("You␣can´t␣use␣`"); *print_cmd_chr*(*cur_cmd*, *cur_chr*); *print*("´␣after␣");
      print_cmd_chr(*q*, 0); *help1*("I´m␣forgetting␣what␣you␣said␣and␣not␣changing␣anything.");
      error; **return**;
      end;
    end;
  $p \leftarrow cur\_chr$; *scan_eight_bit_int*;
  case *p* **of**
  int_val: $l \leftarrow cur\_val + count\_base$;
  dimen_val: $l \leftarrow cur\_val + scaled\_base$;
  glue_val: $l \leftarrow cur\_val + skip\_base$;
  mu_val: $l \leftarrow cur\_val + mu\_skip\_base$;
  end; { there are no other cases }
  end;
found:

This code is used in section 1236.

1238. ⟨Compute result of *register* or *advance*, put it in *cur_val* 1238⟩ ≡
  if $p < glue\_val$ **then**
    begin if $p = int\_val$ **then** *scan_int* **else** *scan_normal_dimen*;
    if $q = advance$ **then** $cur\_val \leftarrow cur\_val + eqtb[l].int$;
    end
  else begin *scan_glue*(*p*);
    if $q = advance$ **then** ⟨Compute the sum of two glue specs 1239⟩;
    end

This code is used in section 1236.

1239. ⟨Compute the sum of two glue specs 1239⟩ ≡
  begin $q \leftarrow new\_spec(cur\_val)$; $r \leftarrow equiv(l)$; *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 1238.

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

```
begin scan_int;
if p < glue_val then
  if q = multiply then
    if p = int_val then cur_val ← mult_integers(eqtb[l].int, cur_val)
    else cur_val ← nx_plus_y(eqtb[l].int, cur_val, 0)
  else cur_val ← x_over_n(eqtb[l].int, cur_val)
else begin s ← equiv(l); r ← new_spec(s);
  if q = multiply then
    begin width(r) ← nx_plus_y(width(s), cur_val, 0); stretch(r) ← nx_plus_y(stretch(s), cur_val, 0);
    shrink(r) ← nx_plus_y(shrink(s), cur_val, 0);
    end
  else begin width(r) ← x_over_n(width(s), cur_val); stretch(r) ← x_over_n(stretch(s), cur_val);
    shrink(r) ← x_over_n(shrink(s), cur_val);
    end;
  cur_val ← r;
  end;
end
```

This code is used in section 1236.

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

⟨Assignments 1217⟩ +≡

```
set_box: begin scan_eight_bit_int;
  if global then n ← 256 + cur_val else n ← cur_val;
  scan_optional_equals;
  if set_box_allowed then scan_box(box_flag + n)
  else begin print_err("Improper␣"); print_esc("setbox");
    help2("Sorry,␣\setbox␣is␣not␣allowed␣after␣\halign␣in␣a␣display,")
    ("or␣between␣\accent␣and␣an␣accented␣character."); error;
    end;
  end;
```

1242. 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 1217⟩ +≡

```
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;
```

1243. ⟨Declare subprocedures for *prefixed_command* 1215⟩ +≡

```
procedure alter_aux;
  var c: halfword;  { hmode or vmode }
  begin if cur_chr ≠ abs(mode) then report_illegal_case
  else begin c ← cur_chr; scan_optional_equals;
    if c = vmode then
      begin scan_normal_dimen; prev_depth ← cur_val;
      end
    else begin scan_int;
      if (cur_val ≤ 0) ∨ (cur_val > 32767) then
        begin print_err("Bad␣space␣factor");
        help1("I␣allow␣only␣values␣in␣the␣range␣1..32767␣here."); int_error(cur_val);
        end
      else space_factor ← cur_val;
      end;
    end;
  end;
```

1244. ⟨Declare subprocedures for *prefixed_command* 1215⟩ +≡

```
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;
```

1245. ⟨Declare subprocedures for *prefixed_command* 1215⟩ +≡

```
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;
```

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

```
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;
```

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

```
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;
```

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

⟨ Assignments 1217 ⟩ +≡

```
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;
```

1249. 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 ← 41;
```

1250. 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);
```

1251. ⟨ 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");
```

1252. ⟨ Assignments 1217 ⟩ +≡

```
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;
```

1253. 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 1217 ⟩ +≡
assign_font_dimen: **begin** *find_font_dimen*(*true*); $k \leftarrow$ *cur_val*; *scan_optional_equals*; *scan_normal_dimen*;
  font_info[*k*].*sc* ← *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*] ← *cur_val* **else** *skew_char*[*f*] ← *cur_val*;
  end;

1254. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive("hyphenchar", *assign_font_int*, 0); *primitive*("skewchar", *assign_font_int*, 1);

1255. ⟨ 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");

1256. Here is where the information for a new font gets loaded.

⟨ Assignments 1217 ⟩ +≡
def_font: *new_font*(*a*);

1257. ⟨ Declare subprocedures for *prefixed_command* 1215 ⟩ +≡
procedure *new_font*(*a* : *small_number*);
  label *common_ending*;
  var *u*: *pointer*; { user's font identifier }
    s: *scaled*; { stated "at" size, or negative of scaled magnification }
    f: *internal_font_number*; { runs through existing fonts }
    t: *str_number*; { name for the frozen font identifier }
    old_setting: 0 .. *max_selector*; { holds *selector* setting }
    flushable_string: *str_number*; { string not yet referenced }
  begin if *job_name* = 0 **then** *open_log_file*; { avoid confusing `texput` with the font name }
  get_r_token; $u \leftarrow$ *cur_cs*;
  if $u \geq$ *hash_base* **then** $t \leftarrow$ *text*(*u*)
  else if $u \geq$ *single_base* **then**
      if $u =$ *null_cs* **then** $t \leftarrow$ "FONT" **else** $t \leftarrow u -$ *single_base*
    else begin *old_setting* ← *selector*; *selector* ← *new_string*; *print*("FONT"); *print*($u -$ *active_base*);
      selector ← *old_setting*; *str_room*(1); $t \leftarrow$ *make_string*;
      end;
  define(*u*, *set_font*, *null_font*); *scan_optional_equals*; *scan_file_name*;
  ⟨ Scan the font size specification 1258 ⟩;
  ⟨ If this font has already been loaded, set *f* to the internal font number and **goto** *common_ending* 1260 ⟩;
  $f \leftarrow$ *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;

1258. ⟨ Scan the font size specification 1258 ⟩ ≡

```
name_in_progress ← true; { this keeps cur_name from being changed }
if scan_keyword("at") then ⟨ Put the (positive) 'at' size into s 1259 ⟩
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 1257.

1259. ⟨ Put the (positive) 'at' size into s 1259 ⟩ ≡

```
begin scan_normal_dimen; s ← cur_val;
if (s ≤ 0) ∨ (s ≥ '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 ← 10 ∗ unity;
  end;
end
```

This code is used in section 1258.

1260. 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* 1260 ⟩ ≡

```
flushable_string ← str_ptr − 1;
for f ← 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 cur_name = flushable_string then
      begin flush_string; cur_name ← font_name[f];
      end;
    if s > 0 then
      begin if s = font_size[f] then goto common_ending;
      end
    else if font_size[f] = xn_over_d(font_dsize[f], −s, 1000) then goto common_ending;
    end
```

This code is used in section 1257.

1261. ⟨ 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;
```

1262. ⟨ 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*);

1263. ⟨ 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;

1264. ⟨ Assignments 1217 ⟩ +≡
set_interaction: *new_interaction*;

1265. ⟨ Declare subprocedures for *prefixed_command* 1215 ⟩ +≡
procedure *new_interaction*;
begin *print_ln*; *interaction* ← *cur_chr*; ⟨ Initialize the print *selector* based on *interaction* 75 ⟩;
if *log_opened* **then** *selector* ← *selector* + 2;
end;

1266. 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 }

1267. ⟨ Set initial values of key variables 21 ⟩ +≡
after_token ← 0;

1268. ⟨ Cases of *main_control* that don't depend on *mode* 1210 ⟩ +≡
any_mode(*after_assignment*): **begin** *get_token*; *after_token* ← *cur_tok*;
end;

1269. ⟨ Insert a token saved by \afterassignment, if any 1269 ⟩ ≡
if *after_token* ≠ 0 **then**
begin *cur_tok* ← *after_token*; *back_input*; *after_token* ← 0;
end

This code is used in section 1211.

1270. 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* 1043 ⟩ +≡
procedure *do_assignments*;
label *exit*;
begin loop
begin ⟨ Get the next non-blank non-relax non-call token 404 ⟩;
if *cur_cmd* ≤ *max_non_prefixed_command* **then return**;
set_box_allowed ← *false*; *prefixed_command*; *set_box_allowed* ← *true*;
end;
exit: **end**;

1271. ⟨ Cases of *main_control* that don't depend on *mode* 1210 ⟩ +≡
any_mode(*after_group*): **begin** *get_token*; *save_for_after*(*cur_tok*);
  end;

1272. 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);

1273. ⟨ 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`");

1274. ⟨ Cases of *main_control* that don't depend on *mode* 1210 ⟩ +≡
any_mode(*in_stream*): *open_or_close_in*;

1275. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
procedure *open_or_close_in*;
  var *c*: 0 . . 1; { 1 for `\openin`, 0 for `\closein` }
    n: 0 . . 15; { stream number }
  begin *c* ← *cur_chr*; *scan_four_bit_int*; *n* ← *cur_val*;
  if *read_open*[*n*] ≠ *closed* **then**
    begin *a_close*(*read_file*[*n*]); *read_open*[*n*] ← *closed*;
    end;
  if *c* ≠ 0 **then**
    begin *scan_optional_equals*; *scan_file_name*;
    if *cur_ext* = "" **then** *cur_ext* ← "`.tex`";
    pack_cur_name;
    if *a_open_in*(*read_file*[*n*]) **then** *read_open*[*n*] ← *just_open*;
    end;
  end;

1276. The user can issue messages to the terminal, regardless of the current mode.

⟨ Cases of *main_control* that don't depend on *mode* 1210 ⟩ +≡
any_mode(*message*): *issue_message*;

1277. ⟨ Put each of TEX's primitives into the hash table 226 ⟩ +≡
  primitive("`message`", *message*, 0); *primitive*("`errmessage`", *message*, 1);

1278. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
message: **if** *chr_code* = 0 **then** *print_esc*("`message`")
  else *print_esc*("`errmessage`");

1279. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡

```
procedure issue_message;
  var old_setting: 0 .. max_selector; { holds selector setting }
    c: 0 .. 1; { identifies \message and \errmessage }
    s: str_number; { the message }
  begin c ← cur_chr; link(garbage) ← scan_toks(false, true); old_setting ← selector;
  selector ← new_string; token_show(def_ref); selector ← old_setting; flush_list(def_ref); str_room(1);
  s ← make_string;
  if c = 0 then ⟨ Print string s on the terminal 1280 ⟩
  else ⟨ Print string s as an error message 1283 ⟩;
  flush_string;
  end;
```

1280. ⟨ Print string *s* on the terminal 1280 ⟩ ≡

```
  begin if term_offset + length(s) > max_print_line − 2 then print_ln
  else if (term_offset > 0) ∨ (file_offset > 0) then print_char("␣");
  slow_print(s); update_terminal;
  end
```

This code is used in section 1279.

1281. 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? }
```

1282. ⟨ Set initial values of key variables 21 ⟩ +≡

```
  long_help_seen ← false;
```

1283. ⟨ Print string *s* as an error message 1283 ⟩ ≡

```
  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 1279.

1284. 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;
```

1285. 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* 1210 ⟩ +≡

```
any_mode(case_shift): shift_case;
```

1286. ⟨ 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*);

1287. ⟨ 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");

1288. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
procedure *shift_case*;
 var *b*: *pointer*; { *lc_code_base* or *uc_code_base* }
  p: *pointer*; { runs through the token list }
  t: *halfword*; { token }
  c: *eight_bits*; { character code }
 begin $b \leftarrow cur\_chr$; $p \leftarrow scan\_toks(false, false)$; $p \leftarrow link(def\_ref)$;
 while $p \neq null$ **do**
  begin ⟨ Change the case of the token in *p*, if a change is appropriate 1289 ⟩;
  $p \leftarrow link(p)$;
  end;
 back_list(*link*(*def_ref*)); *free_avail*(*def_ref*); { omit reference count }
 end;

1289. 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 1289 ⟩ ≡
 $t \leftarrow info(p)$;
 if $t < cs\_token\_flag + single\_base$ **then**
  begin $c \leftarrow t$ **mod** 256;
  if $equiv(b + c) \neq 0$ **then** $info(p) \leftarrow t - c + equiv(b + c)$;
  end

This code is used in section 1288.

1290. 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* 1210 ⟩ +≡
any_mode(*xray*): *show_whatever*;

1291. **define** *show_code* = 0 { \show }
 define *show_box_code* = 1 { \showbox }
 define *show_the_code* = 2 { \showthe }
 define *show_lists_code* = 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_code*);

1292. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 227 ⟩ +≡
xray: **case** *chr_code* **of**
 show_box_code: *print_esc*("showbox");
 show_the_code: *print_esc*("showthe");
 show_lists_code: *print_esc*("showlists");
 othercases *print_esc*("show")
 endcases;

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

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

1294. ⟨ Show the current meaning of a token, then **goto** *common_ending* 1294 ⟩ ≡

```
  begin get_token;
  if interaction = error_stop_mode then wake_up_terminal;
  print_nl(">␣");
  if cur_cs ≠ 0 then
    begin sprint_cs(cur_cs); print_char("=");
    end;
  print_meaning; goto common_ending;
  end
```

This code is used in section 1293.

1295. ⟨ 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");
```

1296. ⟨Show the current contents of a box 1296⟩ ≡
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 1293.

1297. ⟨Show the current value of some parameter or register, then **goto** *common_ending* 1297⟩ ≡
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 1293.

1298. ⟨Complete a potentially long \show command 1298⟩ ≡
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 1293.

1299. 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*;

1300. ⟨ Set initial values of key variables 21 ⟩ +≡
  format_ident ← 0;

1301. ⟨ Initialize table entries (done by `INITEX` only) 164 ⟩ +≡
  format_ident ← "␣(INITEX)";

1302. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
  init procedure *store_fmt_file*;
  label *found1*, *found2*, *done1*, *done2*;
  var *j*, *k*, *l*: *integer*; { all-purpose indices }
    p, *q*: *pointer*; { all-purpose pointers }
    x: *integer*; { something to dump }
    w: *four_quarters*; { four ASCII codes }
  begin ⟨ If dumping is not allowed, abort 1304 ⟩;
  ⟨ Create the *format_ident*, open the format file, and inform the user that dumping has begun 1328 ⟩;
  ⟨ Dump constants for consistency check 1307 ⟩;
  ⟨ Dump the string pool 1309 ⟩;
  ⟨ Dump the dynamic memory 1311 ⟩;
  ⟨ Dump the table of equivalents 1313 ⟩;
  ⟨ Dump the font information 1320 ⟩;
  ⟨ Dump the hyphenation tables 1324 ⟩;
  ⟨ Dump a couple more things and the closing check word 1326 ⟩;
  ⟨ Close the format file 1329 ⟩;
  end;
  tini

1303. 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* 524 ⟩
function *load_fmt_file*: *boolean*;
label *bad_fmt*, *exit*;
var *j*, *k*: *integer*; { all-purpose indices }
  p, *q*: *pointer*; { all-purpose pointers }
  x: *integer*; { something undumped }
  w: *four_quarters*; { four ASCII codes }
begin ⟨ Undump constants for consistency check 1308 ⟩;
⟨ Undump the string pool 1310 ⟩;
⟨ Undump the dynamic memory 1312 ⟩;
⟨ Undump the table of equivalents 1314 ⟩;
⟨ Undump the font information 1321 ⟩;
⟨ Undump the hyphenation tables 1325 ⟩;
⟨ Undump a couple more things and the closing check word 1327 ⟩;
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**;

1304. 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 1304 ⟩ ≡
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 1302.

1305. Format files consist of *memory_word* items, and we use the following macros to dump words of different types:

define *dump_wd*(#) ≡
  begin *fmt_file*↑ ← #; *put*(*fmt_file*); **end**
define *dump_int*(#) ≡
  begin *fmt_file*↑.*int* ← #; *put*(*fmt_file*); **end**
define *dump_hh*(#) ≡
  begin *fmt_file*↑.*hh* ← #; *put*(*fmt_file*); **end**
define *dump_qqqq*(#) ≡
  begin *fmt_file*↑.*qqqq* ← #; *put*(*fmt_file*); **end**

⟨ Global variables 13 ⟩ +≡
fmt_file: *word_file*; { for input or output of format information }

1306. 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 $a \le x \le b$. System error messages should be suppressed when undumping.

define *undump_wd*(#) ≡
  begin *get*(*fmt_file*); # ← *fmt_file*↑; **end**
define *undump_int*(#) ≡
  begin *get*(*fmt_file*); # ← *fmt_file*↑.*int*; **end**
define *undump_hh*(#) ≡
  begin *get*(*fmt_file*); # ← *fmt_file*↑.*hh*; **end**
define *undump_qqqq*(#) ≡
  begin *get*(*fmt_file*); # ← *fmt_file*↑.*qqqq*; **end**
define *undump_end_end*(#) ≡ # ← x; **end**
define *undump_end*(#) ≡ (x > #) **then goto** *bad_fmt* **else** *undump_end_end*
define *undump*(#) ≡
 begin *undump_int*(x);
 if (x < #) ∨ *undump_end*
define *undump_size_end_end*(#) ≡ *too_small*(#) **else** *undump_end_end*
define *undump_size_end*(#) ≡
  if x > # **then** *undump_size_end_end*
define *undump_size*(#) ≡
 begin *undump_int*(x);
 if x < # **then goto** *bad_fmt*;
 undump_size_end

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

⟨ Dump constants for consistency check 1307 ⟩ ≡
 dump_int(@$);
 dump_int(*mem_bot*);
 dump_int(*mem_top*);
 dump_int(*eqtb_size*);
 dump_int(*hash_prime*);
 dump_int(*hyph_size*)

This code is used in section 1302.

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

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

This code is used in section 1303.

1309. **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 1309⟩ ≡
dump_int(*pool_ptr*); *dump_int*(*str_ptr*);
for $k \leftarrow 0$ **to** *str_ptr* **do** *dump_int*(*str_start*[*k*]);
$k \leftarrow 0$;
while $k + 4 < pool\_ptr$ **do**
begin *dump_four_ASCII*; $k \leftarrow k + 4$;
end;
$k \leftarrow pool\_ptr - 4$; *dump_four_ASCII*; *print_ln*; *print_int*(*str_ptr*);
print("␣strings␣of␣total␣length␣"); *print_int*(*pool_ptr*)

This code is used in section 1302.

1310. **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 1310⟩ ≡
undump_size(0)(*pool_size*)(´string␣pool␣size´)(*pool_ptr*);
undump_size(0)(*max_strings*)(´max␣strings´)(*str_ptr*);
for $k \leftarrow 0$ **to** *str_ptr* **do** *undump*(0)(*pool_ptr*)(*str_start*[*k*]);
$k \leftarrow 0$;
while $k + 4 < pool\_ptr$ **do**
begin *undump_four_ASCII*; $k \leftarrow k + 4$;
end;
$k \leftarrow pool\_ptr - 4$; *undump_four_ASCII*; $init\_str\_ptr \leftarrow str\_ptr$; $init\_pool\_ptr \leftarrow pool\_ptr$

This code is used in section 1303.

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

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

⟨Dump the dynamic memory 1311⟩ ≡
sort_avail; $var\_used \leftarrow 0$; *dump_int*(*lo_mem_max*); *dump_int*(*rover*); $p \leftarrow mem\_bot$; $q \leftarrow rover$; $x \leftarrow 0$;
repeat for $k \leftarrow p$ **to** $q + 1$ **do** *dump_wd*(*mem*[*k*]);
$x \leftarrow x + q + 2 - p$; $var\_used \leftarrow var\_used + q - p$; $p \leftarrow q + node\_size(q)$; $q \leftarrow rlink(q)$;
until $q = rover$;
$var\_used \leftarrow var\_used + lo\_mem\_max - p$; $dyn\_used \leftarrow mem\_end + 1 - hi\_mem\_min$;
for $k \leftarrow p$ **to** *lo_mem_max* **do** *dump_wd*(*mem*[*k*]);
$x \leftarrow x + lo\_mem\_max + 1 - p$; *dump_int*(*hi_mem_min*); *dump_int*(*avail*);
for $k \leftarrow hi\_mem\_min$ **to** *mem_end* **do** *dump_wd*(*mem*[*k*]);
$x \leftarrow x + mem\_end + 1 - hi\_mem\_min$; $p \leftarrow avail$;
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 1302.

1312. ⟨Undump the dynamic memory 1312⟩ ≡
undump(*lo_mem_stat_max* + 1000)(*hi_mem_stat_min* − 1)(*lo_mem_max*);
undump(*lo_mem_stat_max* + 1)(*lo_mem_max*)(*rover*); $p \leftarrow$ *mem_bot*; $q \leftarrow$ *rover*;
repeat for $k \leftarrow p$ **to** $q + 1$ **do** *undump_wd*(*mem*[*k*]);
  $p \leftarrow q +$ *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*;
for $k \leftarrow p$ **to** *lo_mem_max* **do** *undump_wd*(*mem*[*k*]);
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*;
for $k \leftarrow$ *hi_mem_min* **to** *mem_end* **do** *undump_wd*(*mem*[*k*]);
undump_int(*var_used*); *undump_int*(*dyn_used*)

This code is used in section 1303.

1313. ⟨Dump the table of equivalents 1313⟩ ≡
⟨Dump regions 1 to 4 of *eqtb* 1315⟩;
⟨Dump regions 5 and 6 of *eqtb* 1316⟩;
dump_int(*par_loc*); *dump_int*(*write_loc*);
⟨Dump the hash table 1318⟩

This code is used in section 1302.

1314. ⟨Undump the table of equivalents 1314⟩ ≡
⟨Undump regions 1 to 6 of *eqtb* 1317⟩;
undump(*hash_base*)(*frozen_control_sequence*)(*par_loc*); *par_token* ← *cs_token_flag* + *par_loc*;
undump(*hash_base*)(*frozen_control_sequence*)(*write_loc*);
⟨Undump the hash table 1319⟩

This code is used in section 1303.

1315. 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* 1315⟩ ≡

```
k ← active_base;
repeat j ← k;
  while j < int_base − 1 do
    begin if (equiv(j) = equiv(j + 1)) ∧ (eq_type(j) = eq_type(j + 1)) ∧ (eq_level(j) = eq_level(j + 1))
          then goto found1;
    incr(j);
    end;
  l ← int_base; goto done1;  { j = int_base − 1 }
found1: incr(j); l ← j;
  while j < int_base − 1 do
    begin if (equiv(j) ≠ equiv(j + 1)) ∨ (eq_type(j) ≠ eq_type(j + 1)) ∨ (eq_level(j) ≠ eq_level(j + 1))
          then goto done1;
    incr(j);
    end;
done1: dump_int(l − k);
  while k < l do
    begin dump_wd(eqtb[k]); incr(k);
    end;
  k ← j + 1; dump_int(k − l);
until k = int_base
```

This code is used in section 1313.

1316. ⟨Dump regions 5 and 6 of *eqtb* 1316⟩ ≡

```
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);
  while k < l do
    begin dump_wd(eqtb[k]); incr(k);
    end;
  k ← j + 1; dump_int(k − l);
until k > eqtb_size
```

This code is used in section 1313.

1317. ⟨ Undump regions 1 to 6 of *eqtb* 1317 ⟩ ≡

$k \leftarrow active\_base$;
repeat $undump\_int(x)$;
  if $(x < 1) \lor (k + x > eqtb\_size + 1)$ **then goto** bad_fmt;
  for $j \leftarrow k$ **to** $k + x - 1$ **do** $undump\_wd(eqtb[j])$;
  $k \leftarrow k + x$; $undump\_int(x)$;
  if $(x < 0) \lor (k + x > eqtb\_size + 1)$ **then goto** bad_fmt;
  for $j \leftarrow k$ **to** $k + x - 1$ **do** $eqtb[j] \leftarrow eqtb[k - 1]$;
  $k \leftarrow k + x$;
until $k > eqtb\_size$

This code is used in section 1314.

1318. A different scheme is used to compress the hash table, since its lower region is usually sparse. When $text(p) \neq 0$ for $p \leq hash\_used$, we output two words, p and $hash[p]$. The hash table is, of course, densely packed for $p \geq hash\_used$, so the remaining entries are output in a block.

⟨ Dump the hash table 1318 ⟩ ≡

$dump\_int(hash\_used)$; $cs\_count \leftarrow frozen\_control\_sequence - 1 - hash\_used$;
for $p \leftarrow hash\_base$ **to** $hash\_used$ **do**
  if $text(p) \neq 0$ **then**
    begin $dump\_int(p)$; $dump\_hh(hash[p])$; $incr(cs\_count)$;
    end;
for $p \leftarrow hash\_used + 1$ **to** $undefined\_control\_sequence - 1$ **do** $dump\_hh(hash[p])$;
$dump\_int(cs\_count)$;
$print\_ln$; $print\_int(cs\_count)$; $print($"␣multiletter␣control␣sequences"$)$

This code is used in section 1313.

1319. ⟨ Undump the hash table 1319 ⟩ ≡

$undump(hash\_base)(frozen\_control\_sequence)(hash\_used)$; $p \leftarrow hash\_base - 1$;
repeat $undump(p + 1)(hash\_used)(p)$; $undump\_hh(hash[p])$;
until $p = hash\_used$;
for $p \leftarrow hash\_used + 1$ **to** $undefined\_control\_sequence - 1$ **do** $undump\_hh(hash[p])$;
$undump\_int(cs\_count)$

This code is used in section 1314.

1320. ⟨ Dump the font information 1320 ⟩ ≡

$dump\_int(fmem\_ptr)$;
for $k \leftarrow 0$ **to** $fmem\_ptr - 1$ **do** $dump\_wd(font\_info[k])$;
$dump\_int(font\_ptr)$;
for $k \leftarrow null\_font$ **to** $font\_ptr$ **do** ⟨ Dump the array info for internal font number k 1322 ⟩;
$print\_ln$; $print\_int(fmem\_ptr - 7)$; $print($"␣words␣of␣font␣info␣for␣"$)$;
$print\_int(font\_ptr - font\_base)$; $print($"␣preloaded␣font"$)$;
if $font\_ptr \neq font\_base + 1$ **then** $print\_char($"s"$)$

This code is used in section 1302.

1321. ⟨ Undump the font information 1321 ⟩ ≡

$undump\_size(7)(font\_mem\_size)($´font␣mem␣size´$)(fmem\_ptr)$;
for $k \leftarrow 0$ **to** $fmem\_ptr - 1$ **do** $undump\_wd(font\_info[k])$;
$undump\_size(font\_base)(font\_max)($´font␣max´$)(font\_ptr)$;
for $k \leftarrow null\_font$ **to** $font\_ptr$ **do** ⟨ Undump the array info for internal font number k 1323 ⟩

This code is used in section 1303.

1322. ⟨ Dump the array info for internal font number *k* 1322 ⟩ ≡
begin *dump_qqqq*(*font_check*[*k*]); *dump_int*(*font_size*[*k*]); *dump_int*(*font_dsize*[*k*]);
dump_int(*font_params*[*k*]);
dump_int(*hyphen_char*[*k*]); *dump_int*(*skew_char*[*k*]);
dump_int(*font_name*[*k*]); *dump_int*(*font_area*[*k*]);
dump_int(*font_bc*[*k*]); *dump_int*(*font_ec*[*k*]);
dump_int(*char_base*[*k*]); *dump_int*(*width_base*[*k*]); *dump_int*(*height_base*[*k*]);
dump_int(*depth_base*[*k*]); *dump_int*(*italic_base*[*k*]); *dump_int*(*lig_kern_base*[*k*]);
dump_int(*kern_base*[*k*]); *dump_int*(*exten_base*[*k*]); *dump_int*(*param_base*[*k*]);
dump_int(*font_glue*[*k*]);
dump_int(*bchar_label*[*k*]); *dump_int*(*font_bchar*[*k*]); *dump_int*(*font_false_bchar*[*k*]);
print_nl("\font"); *print_esc*(*font_id_text*(*k*)); *print_char*("=");
print_file_name(*font_name*[*k*], *font_area*[*k*], "");
if *font_size*[*k*] $\neq$ *font_dsize*[*k*] **then**
begin *print*("␣at␣"); *print_scaled*(*font_size*[*k*]); *print*("pt");
end;
end

This code is used in section 1320.

1323. ⟨ Undump the array info for internal font number *k* 1323 ⟩ ≡
begin *undump_qqqq*(*font_check*[*k*]);
undump_int(*font_size*[*k*]); *undump_int*(*font_dsize*[*k*]);
undump(*min_halfword*)(*max_halfword*)(*font_params*[*k*]);
undump_int(*hyphen_char*[*k*]); *undump_int*(*skew_char*[*k*]);
undump(0)(*str_ptr*)(*font_name*[*k*]); *undump*(0)(*str_ptr*)(*font_area*[*k*]);
undump(0)(255)(*font_bc*[*k*]); *undump*(0)(255)(*font_ec*[*k*]);
undump_int(*char_base*[*k*]); *undump_int*(*width_base*[*k*]); *undump_int*(*height_base*[*k*]);
undump_int(*depth_base*[*k*]); *undump_int*(*italic_base*[*k*]); *undump_int*(*lig_kern_base*[*k*]);
undump_int(*kern_base*[*k*]); *undump_int*(*exten_base*[*k*]); *undump_int*(*param_base*[*k*]);
undump(*min_halfword*)(*lo_mem_max*)(*font_glue*[*k*]);
undump(0)(*fmem_ptr* − 1)(*bchar_label*[*k*]); *undump*(*min_quarterword*)(*non_char*)(*font_bchar*[*k*]);
undump(*min_quarterword*)(*non_char*)(*font_false_bchar*[*k*]);
end

This code is used in section 1321.

1324. ⟨ Dump the hyphenation tables 1324 ⟩ ≡

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

This code is used in section 1302.

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

⟨ Undump the hyphenation tables 1325 ⟩ ≡

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

This code is used in section 1303.

1326. We have already printed a lot of statistics, so we set *tracing_stats* ← 0 to prevent them from appearing again.

⟨ Dump a couple more things and the closing check word 1326 ⟩ ≡
dump_int(*interaction*); *dump_int*(*format_ident*); *dump_int*(69069); *tracing_stats* ← 0

This code is used in section 1302.

1327. ⟨ Undump a couple more things and the closing check word 1327 ⟩ ≡
undump(*batch_mode*)(*error_stop_mode*)(*interaction*); *undump*(0)(*str_ptr*)(*format_ident*); *undump_int*(*x*);
if $(x \neq 69069) \lor$ *eof*(*fmt_file*) **then goto** *bad_fmt*

This code is used in section 1303.

1328. ⟨ Create the *format_ident*, open the format file, and inform the user that dumping has begun 1328 ⟩ ≡
selector ← *new_string*; *print*("␣(preloaded␣format="); *print*(*job_name*); *print_char*("␣");
print_int(*year*); *print_char*("."); *print_int*(*month*); *print_char*("."); *print_int*(*day*); *print_char*(")");
if *interaction* = *batch_mode* **then** *selector* ← *log_only*
else *selector* ← *term_and_log*;
str_room(1); *format_ident* ← *make_string*; *pack_job_name*(*format_extension*);
while ¬*w_open_out*(*fmt_file*) **do** *prompt_file_name*("format␣file␣name", *format_extension*);
print_nl("Beginning␣to␣dump␣on␣file␣"); *slow_print*(*w_make_name_string*(*fmt_file*)); *flush_string*;
print_nl(""); *slow_print*(*format_ident*)

This code is used in section 1302.

1329. ⟨ Close the format file 1329 ⟩ ≡
w_close(*fmt_file*)

This code is used in section 1302.

1330. 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 1333 ⟩

1331. 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* $\leftarrow$ 314159. (3) Soon `VIRTEX` will print '`*`', waiting for more input; and at this point we interrupt the program and save its core image in some form that the operating system can reload speedily. (4) When that core image is activated, the program starts again at the beginning; but now *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 }

1332. Now this is really it: TEX starts and ends here.

The initial test involving *ready_already* should be deleted if the Pascal runtime system is smart enough to detect such a "mistake."

```
begin { start_here }
history ← fatal_error_stop; { in case we quit during initialization }
t_open_out; { open the terminal for output }
if ready_already = 314159 then goto start_of_TEX;
⟨Check the "constant" values for consistency 14⟩
if bad > 0 then
  begin wterm_ln(´Ouch---my␣internal␣constants␣have␣been␣clobbered!´, ´---case␣´, bad : 1);
  goto final_end;
  end;
initialize; { set global variables to their starting values }
init if ¬get_strings_started then goto final_end;
init_prim; { call primitive for each primitive }
init_str_ptr ← str_ptr; init_pool_ptr ← pool_ptr; fix_date_and_time;
tini
ready_already ← 314159;
start_of_TEX: ⟨Initialize the output routines 55⟩;
⟨Get the first line of input and prepare to start 1337⟩;
history ← spotless; { ready to go! }
main_control; { come to life }
final_cleanup; { prepare for death }
end_of_TEX: close_files_and_terminate;
final_end: ready_already ← 0;
end.
```

1333. 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.)

If *final_cleanup* is bypassed, this program doesn't bother to close the input files that may still be open.

```
⟨Last-minute procedures 1333⟩ ≡
procedure close_files_and_terminate;
var k: integer; { all-purpose index }
begin ⟨Finish the extensions 1378⟩;
new_line_char ← −1;
stat if tracing_stats > 0 then ⟨Output statistics about this job 1334⟩; tats
wake_up_terminal; ⟨Finish the DVI file 642⟩;
if log_opened then
  begin wlog_cr; a_close(log_file); selector ← selector − 2;
  if selector = term_only then
    begin print_nl("Transcript␣written␣on␣"); slow_print(log_name); print_char(".");
    end;
  end;
end;
```

See also sections 1335, 1336, and 1338.

This code is used in section 1330.

1334. 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 1334⟩ ≡

```
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);
  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 1333.

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

⟨ Last-minute procedures 1333 ⟩ +≡
procedure *final_cleanup*;
  label *exit*;
  var *c*: *small_number*; { 0 for `\end`, 1 for `\dump` }
  begin *c* ← *cur_chr*;
  if $c \neq 1$ **then** *new_line_char* ← −1;
  if *job_name* = 0 **then** *open_log_file*;
  while *input_ptr* > 0 **do**
    if *state* = *token_list* **then** *end_token_list* **else** *end_file_reading*;
  while *open_parens* > 0 **do**
    begin *print*("␣)"); *decr*(*open_parens*);
    end;
  if *cur_level* > *level_one* **then**
    begin *print_nl*("("); *print_esc*("end␣occurred␣"); *print*("inside␣a␣group␣at␣level␣");
    print_int(*cur_level* − *level_one*); *print_char*(")");
    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**;

1336. ⟨ Last-minute procedures 1333 ⟩ +≡
  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

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

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

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

This code is used in section 1332.

1338. 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 1333 ⟩ +≡
  debug procedure debug_help;   { routine to display various things }
  label breakpoint, exit;
  var k, l, m, n: integer;
  begin clear_terminal;
  loop
    begin wake_up_terminal; print_nl("debug␣#␣(-1␣to␣exit):"); update_terminal; read(term_in, m);
    if m < 0 then return
    else if m = 0 then
        begin goto breakpoint;
          { go to every declared label at least once }
      breakpoint: m ← 0; @{´BREAKPOINT´@}
        end
      else begin read(term_in, n);
        case m of
        ⟨ Numbered cases for debug_help 1339 ⟩
        othercases print("?")
        endcases;
        end;
    end;
exit: end;
  gubed
```

1339. ⟨ Numbered cases for *debug_help* 1339 ⟩ ≡
1: *print_word*(*mem*[*n*]); { display *mem*[*n*] in all forms }
2: *print_int*(*info*(*n*));
3: *print_int*(*link*(*n*));
4: *print_word*(*eqtb*[*n*]);
5: *print_word*(*font_info*[*n*]);
6: *print_word*(*save_stack*[*n*]);
7: *show_box*(*n*); { show a box, abbreviated by *show_box_depth* and *show_box_breadth* }
8: **begin** *breadth_max* ← 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* ← 0 **to** *n* **do** *print*(*buffer*[*k*]);
15: **begin** *font_in_short_display* ← *null_font*; *short_display*(*n*);
end;
16: *panicking* ← ¬*panicking*;

This code is used in section 1338.

1340. 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.

1341. 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*(#) ≡ *link*(# + 1) { language number, in the range 0 .. 255 }
define *what_lhm*(#) ≡ *type*(# + 1) { minimum left fragment, in the range 1 .. 63 }
define *what_rhm*(#) ≡ *subtype*(# + 1) { minimum right fragment, in the range 1 .. 63 }
define *write_tokens*(#) ≡ *link*(# + 1) { reference count of token list to write }
define *write_stream*(#) ≡ *info*(# + 1) { stream number (0 to 17) }
define *open_name*(#) ≡ *link*(# + 1) { string number of file name to open }
define *open_area*(#) ≡ *info*(# + 2) { string number of file area for *open_name* }
define *open_ext*(#) ≡ *link*(# + 2) { string number of file extension for *open_name* }

1342. 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*;

1343. ⟨ Set initial values of key variables 21 ⟩ +≡
for $k \leftarrow 0$ **to** 17 **do** *write_open*[k] ← *false*;

1344. 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*);
primitive("immediate", *extension*, *immediate_code*);
primitive("setlanguage", *extension*, *set_language_code*);

1345. 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` }

1346. ⟨ 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;

1347. 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 1347 ⟩ ≡
any_mode(*extension*): *do_extension*;

This code is used in section 1045.

1348. ⟨ Declare action procedures for use by *main_control* 1043 ⟩ +≡
⟨ Declare procedures needed in *do_extension* 1349 ⟩
procedure *do_extension*;
  var *i, j, k*: *integer*; { all-purpose integers }
    p, q, r: *pointer*; { all-purpose pointers }
  begin case *cur_chr* **of**
  open_node: ⟨ Implement `\openout` 1351 ⟩;
  write_node: ⟨ Implement `\write` 1352 ⟩;
  close_node: ⟨ Implement `\closeout` 1353 ⟩;
  special_node: ⟨ Implement `\special` 1354 ⟩;
  immediate_code: ⟨ Implement `\immediate` 1375 ⟩;
  set_language_code: ⟨ Implement `\setlanguage` 1377 ⟩;
  othercases *confusion*("ext1")
  endcases;
  end;

1349. 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* 1349 ⟩ ≡
procedure *new_whatsit*(*s* : *small_number*; *w* : *small_number*);
  var *p*: *pointer*; { the new node }
  begin $p \leftarrow get\_node(w)$; $type(p) \leftarrow whatsit\_node$; $subtype(p) \leftarrow s$; $link(tail) \leftarrow p$; $tail \leftarrow p$;
  end;

See also section 1350.

This code is used in section 1348.

1350. 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* 1349 ⟩ +≡
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** $cur\_val \leftarrow 17$
    else if $cur\_val > 15$ **then** $cur\_val \leftarrow 16$;
    end;
  $write\_stream(tail) \leftarrow cur\_val$;
  end;

1351. ⟨ Implement `\openout` 1351 ⟩ ≡
  begin *new_write_whatsit*(*open_node_size*); *scan_optional_equals*; *scan_file_name*;
  $open\_name(tail) \leftarrow cur\_name$; $open\_area(tail) \leftarrow cur\_area$; $open\_ext(tail) \leftarrow cur\_ext$;
  end

This code is used in section 1348.

1352. 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 1352 ⟩ ≡
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 1348.

1353. ⟨ Implement \closeout 1353 ⟩ ≡
begin $new\_write\_whatsit(write\_node\_size)$; $write\_tokens(tail) \leftarrow null$;
end

This code is used in section 1348.

1354. When '\special{...}' appears, we expand the macros in the token list as in \xdef and \mark.

⟨ Implement \special 1354 ⟩ ≡
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 1348.

1355. 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;

1356. ⟨ Display the whatsit node p 1356 ⟩ ≡
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.

1357. ⟨Make a partial copy of the whatsit node p and make r point to it; set *words* to the number of initial words not yet copied 1357⟩ ≡

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.

1358. ⟨Wipe out the whatsit node p and **goto** *done* 1358⟩ ≡

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.

1359. ⟨Incorporate a whatsit node into a vbox 1359⟩ ≡
do_nothing

This code is used in section 669.

1360. ⟨Incorporate a whatsit node into an hbox 1360⟩ ≡
do_nothing

This code is used in section 651.

1361. ⟨Let d be the width of the whatsit p 1361⟩ ≡
$d \leftarrow 0$

This code is used in section 1147.

1362. **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 1362⟩ ≡ *adv_past*(*cur_p*)

This code is used in section 866.

1363. ⟨Advance past a whatsit node in the pre-hyphenation loop 1363⟩ ≡ *adv_past*(s)

This code is used in section 896.

1364. ⟨Prepare to move whatsit p to the current page, then **goto** *contribute* 1364⟩ ≡
goto *contribute*

This code is used in section 1000.

1365. ⟨ Process whatsit *p* in *vert_break* loop, **goto** *not_found* 1365 ⟩ ≡

goto *not_found*

This code is used in section 973.

1366. ⟨ Output the whatsit node *p* in a vlist 1366 ⟩ ≡

out_what(*p*)

This code is used in section 631.

1367. ⟨ Output the whatsit node *p* in an hlist 1367 ⟩ ≡

out_what(*p*)

This code is used in section 622.

1368. 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* 1368 ⟩ ≡

```
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 1370 and 1373.

This code is used in section 619.

1369. 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;
```

1370. ⟨Declare procedures needed in *hlist_out*, *vlist_out* 1368⟩ +≡
procedure *write_out*(*p* : *pointer*);
  var *old_setting*: 0 . . *max_selector*; { holds print *selector* }
    old_mode: *integer*; { saved *mode* }
    j: *small_number*; { write stream number }
    q, *r*: *pointer*; { temporary variables for list manipulation }
  begin ⟨Expand macros in the token list and make *link*(*def_ref*) point to the result 1371⟩;
  old_setting ← *selector*; *j* ← *write_stream*(*p*);
  if *write_open*[*j*] **then** *selector* ← *j*
  else begin { write to the terminal if file isn't open }
    if $(j = 17) \land (selector = term\_and\_log)$ **then** *selector* ← *log_only*;
    print_nl("");
    end;
  token_show(*def_ref*); *print_ln*; *flush_list*(*def_ref*); *selector* ← *old_setting*;
  end;

1371. 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 1371⟩ ≡
  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 1372⟩;
  mode ← *old_mode*; *end_token_list* { conserve stack space }

This code is used in section 1370.

1372. ⟨Recover from an unbalanced write command 1372⟩ ≡
  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 1371.

1373. The *out_what* procedure takes care of outputting whatsit nodes for *vlist_out* and *hlist_out*.

⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1368 ⟩ +≡
procedure *out_what*(*p* : *pointer*);
  var *j*: *small_number*; { write stream number }
  begin case *subtype*(*p*) **of**
  open_node, *write_node*, *close_node*: ⟨ Do some work that has been queued up for `\write` 1374 ⟩;
  special_node: *special_out*(*p*);
  language_node: *do_nothing*;
  othercases *confusion*("ext4")
  endcases;
  end;

1374. 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` 1374 ⟩ ≡
  if ¬*doing_leaders* **then**
    begin *j* ← *write_stream*(*p*);
    if *subtype*(*p*) = *write_node* **then** *write_out*(*p*)
    else begin if *write_open*[*j*] **then** *a_close*(*write_file*[*j*]);
      if *subtype*(*p*) = *close_node* **then** *write_open*[*j*] ← *false*
      else if *j* < 16 **then**
          begin *cur_name* ← *open_name*(*p*); *cur_area* ← *open_area*(*p*); *cur_ext* ← *open_ext*(*p*);
          if *cur_ext* = "" **then** *cur_ext* ← ".tex";
          pack_cur_name;
          while ¬*a_open_out*(*write_file*[*j*]) **do** *prompt_file_name*("output␣file␣name", ".tex");
          write_open[*j*] ← *true*;
          end;
      end;
    end

This code is used in section 1373.

1375. 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` 1375 ⟩ ≡
  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 1348.

1376. 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* 1043 ⟩ +≡
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 $l \ne$ *clang* **then**
    begin *new_whatsit*(*language_node*, *small_node_size*); *what_lang*(*tail*) $\leftarrow l$; *clang* $\leftarrow l$;
    what_lhm(*tail*) $\leftarrow$ *norm_min*(*left_hyphen_min*); *what_rhm*(*tail*) $\leftarrow$ *norm_min*(*right_hyphen_min*);
    end;
  end;

1377. ⟨ Implement `\setlanguage` 1377 ⟩ ≡
  if *abs*(*mode*) $\ne$ *hmode* **then** *report_illegal_case*
  else begin *new_whatsit*(*language_node*, *small_node_size*); *scan_int*;
    if *cur_val* ≤ 0 **then** *clang* $\leftarrow 0$
    else if *cur_val* > 255 **then** *clang* $\leftarrow 0$
      else *clang* $\leftarrow$ *cur_val*;
    what_lang(*tail*) $\leftarrow$ *clang*; *what_lhm*(*tail*) $\leftarrow$ *norm_min*(*left_hyphen_min*);
    what_rhm(*tail*) $\leftarrow$ *norm_min*(*right_hyphen_min*);
    end

This code is used in section 1348.

1378. ⟨ Finish the extensions 1378 ⟩ ≡
  for $k \leftarrow 0$ **to** 15 **do**
    if *write_open*[k] **then** *a_close*(*write_file*[k])

This code is used in section 1333.

1379. 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.

1380. 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.

⟨ Check single-word *avail* list 168 ⟩ Used in section 167.
⟨ Check that another $ follows 1197 ⟩ Used in sections 1194, 1194, and 1206.
⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger* ← *true* 1195 ⟩ Used in sections 1194 and 1194.
⟨ Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto** *done1* 899 ⟩ Used in section 894.
⟨ Check the "constant" values for consistency 14, 111, 290, 522, 1249 ⟩ Used in section 1332.
⟨ 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 865 ⟩ Used in sections 815 and 863.
⟨ Clear dimensions to zero 650 ⟩ Used in sections 649 and 668.
⟨ Clear off top level from *save_stack* 282 ⟩ Used in section 281.
⟨ Close the format file 1329 ⟩ Used in section 1302.
⟨ Coerce glue to a dimension 451 ⟩ Used in sections 449 and 455.
⟨ Compiler directives 9 ⟩ Used in section 4.
⟨ Complain about an undefined family and set *cur_i* null 723 ⟩ Used in section 722.
⟨ Complain about an undefined macro 370 ⟩ Used in section 367.
⟨ Complain about missing `\endcsname` 373 ⟩ Used in section 372.
⟨ Complain about unknown unit and **goto** *done2* 459 ⟩ Used in section 458.
⟨ Complain that `\the` can't do this; give zero result 428 ⟩ Used in section 413.
⟨ Complain that the user should have said `\mathaccent` 1166 ⟩ Used in section 1165.
⟨ Compleat the incompleat noad 1185 ⟩ Used in section 1184.
⟨ Complete a potentially long `\show` command 1298 ⟩ Used in section 1293.
⟨ Compute result of *multiply* or *divide*, put it in *cur_val* 1240 ⟩ Used in section 1236.
⟨ Compute result of *register* or *advance*, put it in *cur_val* 1238 ⟩ Used in section 1236.
⟨ Compute the amount of skew 741 ⟩ Used in section 738.
⟨ Compute the badness, b, of the current page, using *awful_bad* if the box is too full 1007 ⟩ Used in section 1005.
⟨ Compute the badness, b, using *awful_bad* if the box is too full 975 ⟩ Used in section 974.
⟨ Compute the demerits, d, from r to *cur_p* 859 ⟩ Used in section 855.
⟨ Compute the discretionary *break_width* values 840 ⟩ Used in section 837.
⟨ Compute the hash code h 261 ⟩ Used in section 259.
⟨ Compute the magic offset 765 ⟩ Used in section 1337.
⟨ Compute the minimum suitable height, w, and the corresponding number of extension steps, n; also set *width*(b) 714 ⟩ Used in section 713.
⟨ Compute the new line width 850 ⟩ Used in section 835.
⟨ Compute the register location l and its type p; but **return** if invalid 1237 ⟩ Used in section 1236.
⟨ Compute the sum of two glue specs 1239 ⟩ Used in section 1238.
⟨ Compute the trie op code, v, and set $l \leftarrow 0$ 965 ⟩ Used in section 963.
⟨ Compute the values of *break_width* 837 ⟩ Used in section 836.
⟨ Consider a node with matching width; **goto** *found* if it's a hit 612 ⟩ Used in section 611.
⟨ Consider the demerits for a line from r to *cur_p*; deactivate node r if it should no longer be active; then **goto** *continue* if a line from r to *cur_p* is infeasible, otherwise record a new feasible break 851 ⟩ Used in section 829.
⟨ Constants in the outer block 11 ⟩ Used in section 4.
⟨ Construct a box with limits above and below it, skewed by *delta* 750 ⟩ Used in section 749.
⟨ Construct a sub/superscript combination box x, with the superscript offset by *delta* 759 ⟩ Used in section 756.
⟨ Construct a subscript box x when there is no superscript 757 ⟩ Used in section 756.
⟨ Construct a superscript box x 758 ⟩ Used in section 756.
⟨ Construct a vlist box for the fraction, according to *shift_up* and *shift_down* 747 ⟩ Used in section 743.
⟨ Construct an extensible character in a new box b, using recipe *rem_byte*(q) and font f 713 ⟩ Used in section 710.
⟨ Contribute an entire group to the current parameter 399 ⟩ Used in section 392.

⟨ Incorporate a whatsit node into an hbox 1360 ⟩ Used in section 651.
⟨ Incorporate box dimensions into the dimensions of the hbox that will contain it 653 ⟩ Used in section 651.
⟨ Incorporate box dimensions into the dimensions of the vbox that will contain it 670 ⟩ Used in section 669.
⟨ Incorporate character dimensions into the dimensions of the hbox that will contain it, then move to the next node 654 ⟩ Used in section 651.
⟨ Incorporate glue into the horizontal totals 656 ⟩ Used in section 651.
⟨ Incorporate glue into the vertical totals 671 ⟩ Used in section 669.
⟨ Increase the number of parameters in the last font 580 ⟩ Used in section 578.
⟨ Initialize for hyphenating a paragraph 891 ⟩ Used in section 863.
⟨ Initialize table entries (done by INITEX only) 164, 222, 228, 232, 240, 250, 258, 552, 946, 951, 1216, 1301, 1369 ⟩ Used in section 8.
⟨ Initialize the current page, insert the \topskip glue ahead of p, and **goto** *continue* 1001 ⟩ Used in section 1000.
⟨ Initialize the input routines 331 ⟩ Used in section 1337.
⟨ Initialize the output routines 55, 61, 528, 533 ⟩ Used in section 1332.
⟨ Initialize the print *selector* based on *interaction* 75 ⟩ Used in sections 1265 and 1337.
⟨ Initialize the special list heads and constant nodes 790, 797, 820, 981, 988 ⟩ Used in section 164.
⟨ Initialize variables as *ship_out* begins 617 ⟩ Used in section 640.
⟨ Initialize whatever TEX might access 8 ⟩ Used in section 4.
⟨ Initiate or terminate input from a file 378 ⟩ Used in section 367.
⟨ Initiate the construction of an hbox or vbox, then **return** 1083 ⟩ Used in section 1079.
⟨ Input and store tokens from the next line of the file 483 ⟩ Used in section 482.
⟨ Input for \read from the terminal 484 ⟩ Used in section 483.
⟨ 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] 485 ⟩ Used in section 483.
⟨ Input the next line of *read_file*[m] 486 ⟩ Used in section 483.
⟨ Insert a delta node to prepare for breaks at *cur_p* 843 ⟩ Used in section 836.
⟨ Insert a delta node to prepare for the next active node 844 ⟩ Used in section 836.
⟨ Insert a dummy noad to be sub/superscripted 1177 ⟩ Used in section 1176.
⟨ Insert a new active node from *best_place*[*fit_class*] to *cur_p* 845 ⟩ Used in section 836.
⟨ 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 963 ⟩ Used in section 961.
⟨ Insert a new trie node between q and p, and make p point to it 964 ⟩ Used in section 963.
⟨ Insert a token containing *frozen_endv* 375 ⟩ Used in section 366.
⟨ Insert a token saved by \afterassignment, if any 1269 ⟩ Used in section 1211.
⟨ Insert glue for *split_top_skip* and set $p \leftarrow null$ 969 ⟩ Used in section 968.
⟨ Insert hyphens as specified in *hyph_list*[h] 932 ⟩ Used in section 931.
⟨ Insert macro parameter and **goto** *restart* 359 ⟩ Used in section 357.
⟨ Insert the appropriate mark text into the scanner 386 ⟩ Used in section 367.
⟨ Insert the current list into its environment 812 ⟩ Used in section 800.
⟨ Insert the pair (s,p) into the exception table 940 ⟩ Used in section 939.
⟨ Insert the $\langle v_j \rangle$ template and **goto** *restart* 789 ⟩ 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 $cur\_val = fmem\_ptr$ 579 ⟩ Used in section 578.
⟨ Justify the line ending at breakpoint *cur_p*, and append it to the current vertical list, together with associated penalties and other insertions 880 ⟩ Used in section 877.
⟨ Labels in the outer block 6 ⟩ Used in section 4.
⟨ Last-minute procedures 1333, 1335, 1336, 1338 ⟩ Used in section 1330.