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

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

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

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

A similar test suite called the "`SELLETTE` test" is available for helping to determine whether a particular implementation deserves to be known as 'PRoTE'.

#define *eTeX_version* 2 /* `\eTeXversion` */
#define *eTeX_revision* `".6"` /* `\eTeXrevision` */
#define *eTeX_version_string* `"-2.6"` /* current ε-TEX version */

#define *TeX_banner* `"This␣is␣TeX,␣Version␣3.141592653"` /* printed when TEX starts */

#define `TEX` `ETEX` /* change program name into `ETEX` */

#define *eTeX_states* 1 /* number of ε-TEX state variables in *eqtb* */

#define *Prote_version_string* `"3.141592653-2.6-1.1.0"` /* current PRoTE version */
#define *Prote_version* 1 /* `\Proteversion` */
#define *Prote_revision* `".1.0"` /* `\Proterevision` */

#define *Prote_banner* `"This␣is␣Prote,␣Version␣"` *Prote_version_string*
 /* printed when PRoTE starts */

#define *banner* *Prote_banner*

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 — ε-TeX, however, does use '**var**' parameters for the *reverse* function; there are no tag fields on variant records; there are no assignments **double** = **int**; 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 following is an outline of the program, whose components will be filled in later, using the conventions of `cweb`. 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.

The program starts with inserting header files and occassionaly a function must be placed before declaring TeX's macros, because the function uses identifiers that TeXwill declare as macros.

⟨ Header files and function declarations 9 ⟩
⟨ Preprocessor definitions ⟩
enum { ⟨ Constants in the outer block 11 ⟩ };
⟨ Types in the outer block 18 ⟩
⟨ Forward declarations 52 ⟩
⟨ Global variables 13 ⟩
static void *initialize*(**void**) /∗ this procedure gets things started properly ∗/
{ ⟨ Local variables for initialization 19 ⟩
  ⟨ Initialize whatever TeX might access 8 ⟩;
}
⟨ Basic printing procedures 55 ⟩
⟨ Error handling procedures 71 ⟩

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. There is no need to declare labels in C.

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 '**#ifdef** `DEBUG` ... **#endif**', with apologies to people who wish to preserve the purity of English.

Similarly, there is some conditional code delimited by '**#ifdef** `STAT` ... **#endif**' that is intended for use when statistics are to be kept about TEX's memory usage. The **#ifdef** `STAT` ... **#endif** code also implements diagnostic information for `\tracingparagraphs`, `\tracingpages`, and `\tracingrestores`.

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 '**#ifdef** `INIT` ... **#endif**'.

TEX Live has established the common practice to select the initialization code at runtime using the *iniversion* variable.

⟨Initialize whatever TEX might access 8⟩ ≡
  ⟨Set initial values of key variables 21⟩
#ifdef `INIT`
  if (*iniversion*)   /∗ TEX Live ∗/
  { ⟨Initialize table entries (done by `INITEX` only) 163⟩ }
#endif

This code is used in section 4.

9. The declaration of all basic type definitions needed by HiTEX are contained in a system dependent header file.

⟨Header files and function declarations 9⟩ ≡
#include `<stdbool.h>`
#include `<string.h>`
#include `<math.h>`

See also sections 1681, 1694, 1731, and 1732.

This code is used in section 4.

10. Further it is necessary to define some build in primitives of Pascal that are otherwise not available in C.

#define $odd(X)$ $((X)\,\&\,1)$
#define $chr(X)$ ((**unsigned char**)(X))
#define $ord(X)$ ((**unsigned int**)(X))
#define $abs(X)$ $((X) > -(X)\ ?\ (X) : -(X))$
#define $round(X)$ ((**int**)$((X) \geq 0.0\ ?\ floor((X) + 0.5) : ceil((X) - 0.5)))$

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 = 5000000, /* 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 = 2000000, /* 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 = 79, /* width of context lines on terminal error messages */
half_error_line = 50, /* 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 = 5000, /* maximum number of simultaneous input sources */
max_in_open = 15,
/* maximum number of input files and error insertions that can be going on simultaneously */
font_max = 255, /* maximum internal font number; must not exceed *max_quarterword* and must be at most *font_base* + 256 */
font_mem_size = 8000000, /* number of words of *font_info* for all fonts */
param_size = 10000, /* maximum number of simultaneous macro parameters */
nest_size = 500, /* maximum number of semantic levels simultaneously active */
max_strings = 500000, /* maximum number of strings; must not exceed *max_halfword* */
string_vacancies = 90000, /* 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 = 6250000, /* 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 = 100000,
/* space for saving values outside of current group; must be at most *max_halfword* */
trie_size = 1000000, /* space for hyphenation patterns; should be larger for `INITEX` than it is in production versions of TeX */
trie_op_size = 35111, /* space for "opcodes" in the hyphenation patterns */
dvi_buf_size = 16384, /* size of the output buffer; must be a multiple of 8 */
file_name_size = 1024, /* file names shouldn't be longer than this */
xchg_buffer_size = 64, /* must be at least 64 */
/* size of *eight_bits* buffer for exchange with system routines */
empty_string = 256 /* the empty string follows after 256 characters */

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* 5000000 /∗ 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* 45000 /∗ maximum number of control sequences; it should be at most about $(mem\_max - mem\_min)/(\mathbf{double})\ 10$ ∗/
#define *hash_prime* 35999 /∗ a prime number equal to about 85% of *hash_size* ∗/
#define *hyph_size* 8191 /∗ 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 ⟩ ≡
static int *bad*; /∗ is some "constant" wrong? ∗/

See also sections 20, 26, 30, 32, 39, 53, 72, 75, 78, 95, 103, 114, 115, 116, 117, 123, 164, 172, 180, 212, 245, 252, 255, 270, 285, 296, 300, 303, 304, 307, 308, 309, 332, 360, 381, 386, 387, 409, 437, 446, 479, 488, 492, 511, 512, 526, 531, 538, 548, 549, 554, 591, 594, 604, 615, 645, 646, 660, 683, 718, 723, 764, 769, 813, 820, 822, 824, 827, 832, 838, 846, 871, 891, 899, 904, 906, 920, 925, 942, 946, 949, 970, 979, 981, 988, 1031, 1073, 1265, 1280, 1298, 1304, 1330, 1341, 1344, 1382, 1390, 1432, 1455, 1496, 1498, 1517, 1528, 1529, 1537, 1541, 1565, 1580, 1626, 1637, 1638, 1663, 1669, 1683, 1689, 1711, 1720, 1736, 1740, 1741, 1749, 1750, 1753, and 1768.

This code is used in section 4.

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

⟨ Check the "constant" values for consistency 14 ⟩ ≡
$bad = 0;$
if $((half\_error\_line < 30) \lor (half\_error\_line > error\_line - 15))$ $bad = 1;$
if $(max\_print\_line < 60)$ $bad = 2;$
if $(dvi\_buf\_size \,\%\, 8 \neq 0)$ $bad = 3;$
if $(mem\_bot + 1100 > mem\_top)$ $bad = 4;$
if $(hash\_prime > hash\_size)$ $bad = 5;$
if $(max\_in\_open \geq 128)$ $bad = 6;$
if $(mem\_top < 256 + 11)$ $bad = 7;$ /∗ we will want $null\_list > 255$ ∗/

See also sections 110, 289, and 1248.

This code is used in section 1331.

15. Labels are given symbolic names by the following definitions, so that occasional **goto** statements will be meaningful. We insert the label '*end*' just before the '}' of a procedure in which we have used the '**goto** *end*' 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 '*resume*'. 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.

16. Here are some macros for common programming idioms.

#**define** *incr*(A) $A = A + 1$ /∗ increase a variable by unity ∗/
#**define** *decr*(A) $A = A - 1$ /∗ decrease a variable by unity ∗/
#**define** *negate*(A) $A = -A$ /∗ change the sign of a variable ∗/
#**define** **loop** **while** (*true*) /∗ repeat over and over until a **goto** happens ∗/
format *loop* *else* /∗ WEB's **else** acts like '**while** *true* **do**' ∗/
#**define** *do_nothing* /∗ empty statement ∗/
#**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 $65 = 0101$, 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 ⟩ ≡
  typedef uint8_t ASCII_code; /* eight-bit numbers */

See also sections 25, 38, 100, 108, 112, 149, 211, 268, 299, 547, 593, 919, 924, 1408, and 1631.

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 040 through 0176; 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 **unsigned char** for the characters in text files, even though there now are more than 64 such characters, while other Pascals consider **unsigned 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 unsigned 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 ⟩ ≡
  int *i*;

See also sections 162 and 926.

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 ⟩ +≡
  static ASCII_code *xord*[256]; /* specifies conversion of input characters */
  static text_char *xchr*[256]; /* 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';

$xchr[^\circ 116] =$ 'N';
$xchr[^\circ 117] =$ 'O';
$xchr[^\circ 120] =$ 'P';
$xchr[^\circ 121] =$ 'Q';
$xchr[^\circ 122] =$ 'R';
$xchr[^\circ 123] =$ 'S';
$xchr[^\circ 124] =$ 'T';
$xchr[^\circ 125] =$ 'U';
$xchr[^\circ 126] =$ 'V';
$xchr[^\circ 127] =$ 'W';
$xchr[^\circ 130] =$ 'X';
$xchr[^\circ 131] =$ 'Y';
$xchr[^\circ 132] =$ 'Z';
$xchr[^\circ 133] =$ '[';
$xchr[^\circ 134] =$ '\\';
$xchr[^\circ 135] =$ ']';
$xchr[^\circ 136] =$ '^';
$xchr[^\circ 137] =$ '_';
$xchr[^\circ 140] =$ '`';
$xchr[^\circ 141] =$ 'a';
$xchr[^\circ 142] =$ 'b';
$xchr[^\circ 143] =$ 'c';
$xchr[^\circ 144] =$ 'd';
$xchr[^\circ 145] =$ 'e';
$xchr[^\circ 146] =$ 'f';
$xchr[^\circ 147] =$ 'g';
$xchr[^\circ 150] =$ 'h';
$xchr[^\circ 151] =$ 'i';
$xchr[^\circ 152] =$ 'j';
$xchr[^\circ 153] =$ 'k';
$xchr[^\circ 154] =$ 'l';
$xchr[^\circ 155] =$ 'm';
$xchr[^\circ 156] =$ 'n';
$xchr[^\circ 157] =$ 'o';
$xchr[^\circ 160] =$ 'p';
$xchr[^\circ 161] =$ 'q';
$xchr[^\circ 162] =$ 'r';
$xchr[^\circ 163] =$'s';
$xchr[^\circ 164] =$ 't';
$xchr[^\circ 165] =$ 'u';
$xchr[^\circ 166] =$ 'v';
$xchr[^\circ 167] =$ 'w';
$xchr[^\circ 170] =$ 'x';
$xchr[^\circ 171] =$ 'y';
$xchr[^\circ 172] =$ 'z';
$xchr[^\circ 173] =$ '{';
$xchr[^\circ 174] =$ '|';
$xchr[^\circ 175] =$ '}';
$xchr[^\circ 176] =$ '~';

See also sections 23, 24, 73, 76, 79, 96, 165, 214, 253, 256, 271, 286, 382, 438, 480, 489, 550, 555, 592, 595, 605, 647, 661, 684, 770, 927, 989, 1032, 1266, 1281, 1299, 1342, 1433, 1499, 1518, 1530, and 1746.

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 040. To get the most "permissive" character set, change `'␣'` on the right of these assignment statements to *chr*(*i*).

⟨ Set initial values of key variables 21 ⟩ +≡
  for ($i = 0$; $i \le$ °*37*; $i{+}{+}$) *xchr*[*i*] = *chr*(*i*); /∗ TeX Live ∗/
  for ($i =$ °*177*; $i \le$ °*377*; $i{+}{+}$) *xchr*[*i*] = *chr*(*i*); /∗ TeX Live ∗/

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 040 in case there is a coincidence.

⟨ Set initial values of key variables 21 ⟩ +≡
  for (*i* = *first_text_char*; *i* ≤ *last_text_char*; *i*++) *xord*[*chr*(*i*)] = *invalid_code*;
  for ($i =$ °*200*; $i \le$ °*377*; $i{+}{+}$) *xord*[*xchr*[*i*]] = *i*;
  for ($i = 0$; $i \le$ °*176*; $i{+}{+}$) *xord*[*xchr*[*i*]] = *i*;

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

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

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 ⟩ +≡
typedef uint8_t eight_bits; /* unsigned one-byte quantity */
typedef struct { **FILE** *f; **text_char** d; } **alpha_file**; /* files that contain textual data */
typedef struct { **FILE** *f; **eight_bits** d; } **byte_file**; /* 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 ⟩ +≡
static unsigned char *name_of_file0* [*file_name_size* + 1] = {0}, ***const** *name_of_file* = *name_of_file0* − 1;
/* on some systems this may be a **record** variable */
static int *name_length*;
/* this many characters are actually relevant in *name_of_file* (the rest are blank) */

27. To open files, TeX used Pascal's *reset* function. We use the `kpathsearch` library to implement new functions in the section on TeX Live Integration. Here we give only the function prototypes.

TeX's file-opening functions do not issue their own error messages if something goes wrong. If a file identified by *name_of_file* 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) TeX's file-opening functions return *false*. This allows TeX to undertake appropriate corrective action.

```
static FILE *open_in(char *filename, kpse_file_format_type t, const char *rwb);
    /* TeX Live */
  static bool a_open_in(alpha_file *f);    /* open a text file for input */
  static bool b_open_in(byte_file *f);    /* open a binary file for input */
  static bool w_open_in(word_file *f);    /* open a word file for input */
  static FILE *open_out(const char *file_name, const char *file_mode);    /* TeX Live */
  static bool a_open_out(alpha_file *f);    /* open a text file for output */
  static bool b_open_out(byte_file *f);    /* open a binary file for output */
#ifdef INIT
  static bool w_open_out(word_file *f);    /* open a word file for output */
#endif
```

28. Files can be closed with the Pascal-H routine '*pascal_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 *pascal_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.

```
static void a_close(alpha_file *f)    /* close a text file */
{ pascal_close((*f));
}
static void b_close(byte_file *f)    /* close a binary file */
{ pascal_close((*f));
}
static void w_close(word_file *f)    /* close a word file */
{ pascal_close((*f));
}
```

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 ⟩ +≡

```
  static ASCII_code buffer[buf_size + 1];    /* lines of characters being read */
  static int first;    /* the first unused position in buffer */
  static int last;    /* end of the line just input to buffer */
  static int max_buf_stack;    /* 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 $last = first$. In general, the **ASCII_code** numbers that represent the next line of the file are input into $buffer[first]$, $buffer[first+1]$, $\ldots$, $buffer[last-1]$; 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 $last \equiv first$ (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.d*. 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.d* 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.

```
static bool input_ln(alpha_file *f, bool bypass_eoln)    /* inputs the next line or returns false */
{ int last_nonblank;    /* last with trailing blanks removed */
  if (bypass_eoln)
    if (¬eof((*f))) get((*f));    /* input the first character of the line into f.d */
  last = first;    /* cf. Matthew 19:30 */
  if (eof((*f))) return false;
  else { last_nonblank = first;
    while (¬eoln((*f))) { if (last ≥ max_buf_stack) { max_buf_stack = last + 1;
          if (max_buf_stack ≡ buf_size) ⟨Report overflow of the input buffer, and abort 35⟩;
      }
      buffer[last] = xord[(*f).d];
      get((*f));
      incr(last);
      if (buffer[last − 1] ≠ '␣') last_nonblank = last;
    }
    last = last_nonblank;
    return true;
  }
}
```

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⟩ +≡
  static alpha_file term_in;    /* the terminal as an input file */
  static alpha_file term_out;    /* 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* *term_in.f* = *stdin* /* open the terminal for text input */
#define *t_open_out* *term_out.f* = *stdout* /* 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* *fflush*(*term_out.f*) /* empty the terminal output buffer */
#define *clear_terminal* *fflush*(*term_in.f*) /* 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** *exit*(0)' should be replaced by something that quietly terminates the program.)

⟨Report overflow of the input buffer, and abort 35⟩ ≡
  if (*format_ident* ≡ 0) { *write_ln*(*term_out*, "`Buffer␣size␣exceeded!`");
    exit(0);
  }
  else { *cur_input.loc_field* = *first*;
    cur_input.limit_field = *last* − 1;
    overflow("`buffer␣size`", *buf_size*);
  }

This code is used in sections 31, 1438, and 1724.

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 routine calls *input_command_line* to retrieve a possible command line.

```
static bool init_terminal(void)   /* gets the terminal input started */
{ t_open_in;
  if (input_command_line()) return true;   /* TeX Live */
  loop { wake_up_terminal;
    pascal_write(term_out, "**");
    update_terminal;
    if (¬input_ln(&term_in, true))   /* this shouldn't happen */
    { write_ln(term_out);
      pascal_write(term_out, "!␣End␣of␣file␣on␣the␣terminal...␣why?");
      return false;
    }
    loc = first;
    while ((loc < last) ∧ (buffer[loc] ≡ '␣')) incr(loc);
    if (loc < last) { return true;   /* return unless the line was all blank */
    }
    write_ln(term_out, "Please␣type␣the␣name␣of␣your␣input␣file.");
  }
}
```

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

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

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

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

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

#define *si*(*A*) *A* /∗ convert from **ASCII_code** to *packed_ASCII_code* ∗/
#define *so*(*A*) *A* /∗ convert from *packed_ASCII_code* to **ASCII_code** ∗/

⟨ Types in the outer block 18 ⟩ +≡
 typedef int32_t pool_pointer; /∗ for variables that point into *str_pool* ∗/
 typedef int32_t str_number; /∗ for variables that point into *str_start* ∗/
 typedef uint8_t packed_ASCII_code; /∗ elements of *str_pool* array ∗/

39. ⟨ Global variables 13 ⟩ +≡
 static packed_ASCII_code *str_pool*[*pool_size* + 1]; /∗ the characters ∗/
 static pool_pointer *str_start*[*max_strings* + 1]; /∗ the starting pointers ∗/
 static pool_pointer *pool_ptr*; /∗ first unused position in *str_pool* ∗/
 static str_number *str_ptr*; /∗ number of the current string being created ∗/
 static pool_pointer *init_pool_ptr*; /∗ the starting value of *pool_ptr* ∗/
 static str_number *init_str_ptr*; /∗ 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*(*A*) (*str_start*[*A* + 1] − *str_start*[*A*]) /∗ 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(A)    /* put ASCII_code # at the end of str_pool */
        { str_pool[pool_ptr] = si(A);
          incr(pool_ptr);
        }
#define flush_char  decr(pool_ptr)    /* forget the last character in the pool */
#define str_room(A)    /* make sure that the pool hasn't overflowed */
        { if (pool_ptr + A > pool_size) overflow("pool␣size", pool_size − init_pool_ptr);
        }
```

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.

```
static str_number make_string(void)    /* current string enters the pool */
{ if (str_ptr ≡ max_strings) overflow("number␣of␣strings", max_strings − init_str_ptr);
  incr(str_ptr);
  str_start[str_ptr] = pool_ptr;
  return str_ptr − 1;
}
```

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

```
#define flush_string
        { decr(str_ptr);
          pool_ptr = str_start[str_ptr];
        }
```

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.

```
static bool str_eq_buf(str_number s, int k)    /* test equality of strings */
{    /* loop exit */
  pool_pointer j;    /* running index */
  bool result;    /* result of comparison */
  j = str_start[s];
  while (j < str_start[s + 1]) { if (so(str_pool[j]) ≠ buffer[k]) { result = false;
        goto not_found;
      }
      incr(j);
      incr(k);
  }
  result = true;
not_found: return result;
}
```

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.

```
static bool str_eq_str(str_number s, str_number t)   /* test equality of strings */
{    /* loop exit */
  pool_pointer j, k;   /* running indices */
  bool result;   /* result of comparison */
  result = false;
  if (length(s) ≠ length(t)) goto not_found;
  j = str_start[s];
  k = str_start[t];
  while (j < str_start[s + 1]) { if (str_pool[j] ≠ str_pool[k]) goto not_found;
    incr(j);
    incr(k);
  }
  result = true;
not_found: return result;
}
```

⟨ Declare PRoTE procedures for strings 1564 ⟩

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.

```
static bool get_strings_started(void)   /* initializes the string pool */
{ int k, l;   /* small indices or counters */
  pool_ptr = 0;
  str_ptr = 0;
  str_start[0] = 0;
  ⟨ Make the first 256 strings 48 ⟩;
  ⟨ Add the empty string to the string pool 50 ⟩;
  return true;
}
```

48.

```
#define app_lc_hex(A)  l = A;
    if (l < 10) append_char(l + '0') else append_char(l − 10 + 'a')
```

⟨ Make the first 256 strings 48 ⟩ ≡

```
for (k = 0; k ≤ 255; k++) { if ((⟨ Character k cannot be printed 49 ⟩)) { append_char('^');
      append_char('^');
      if (k < °100) append_char(k + °100)
      else if (k < °200) append_char(k − °100)
      else { app_lc_hex(k/16);
        app_lc_hex(k % 16);
      }
    }
  else append_char(k);
  make_string();
}
```

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 $xchr[°32] \equiv$ '`≠`', would like string 032 to be the single character 032 instead of the three characters 0136, 0136, 0132 (`^^Z`). On the other hand, even people with an extended character set will want to represent string 015 by `^^M`, since 015 is *carriage_return*; the idea is to produce visible strings instead of tabs or line-feeds or carriage-returns or bell-rings or characters that are treated anomalously in text files.

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

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

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

This code is used in section 48.

50. The *pool_file* variable is no longer needed and has been removed.

Instead of reading the other strings from the `TEX.POOL` file, it is sufficient here to add the empty string.

⟨ Add the empty string to the string pool 50 ⟩ ≡
  make_string( );

This code is used in section 47.

51. Without a string pool file there is no need for a pool check sum either. But this is a convenient place to define the function *s_no* that will add literal strings to the string pool at runtime, thereby obtaining their string number.

```
static int s_no(const char *str)
{ if (str[0] ≡ 0) return empty_string;
  if (str[1] ≡ 0) return str[0];
  str_room(strlen(str));
  while (*str ≠ 0) append_char(*str++);
  return make_string();
}
```

52. The function *s_no* is used in *initialize* and needs a forward declaration.

⟨ Forward declarations 52 ⟩ ≡
  static int *s_no*(**const char** **str*);

See also sections 1560, 1562, 1685, 1692, 1705, 1709, and 1725.

This code is used in section 4.

53. 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 $no\_print + 1 \equiv term\_only$, $no\_print + 2 \equiv log\_only$, $term\_only + 2 \equiv log\_only + 1 \equiv term\_and\_log$.

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 ⟩ +≡
```
  static alpha_file log_file;       /* transcript of TEX session */
  static int selector;              /* where to print a message */
  static int8_t dig[23];            /* digits in a number being output */
  static int tally;                 /* the number of characters recently printed */
  static int term_offset;           /* the number of characters on the current terminal line */
  static int file_offset;           /* the number of characters on the current file line */
  static ASCII_code trick_buf[error_line + 1];  /* circular buffer for pseudoprinting */
  static int trick_count;           /* threshold for pseudoprinting, explained later */
  static int first_count;           /* another variable for pseudoprinting */
```

54. ⟨ Initialize the output routines 54 ⟩ ≡
```
  selector = term_only;
  tally = 0;
  term_offset = 0;
  file_offset = 0;
```
See also sections 60, 527, and 532.

This code is used in section 1331.

55. 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.

⟨ Basic printing procedures 55 ⟩ ≡

```
#define put(F) fwrite(&((F).d), sizeof ((F).d), 1, (F).f)
#define get(F) (void) fread(&((F).d), sizeof ((F).d), 1, (F).f)
#define pascal_close(F) fclose((F).f)
#define eof(F) feof((F).f)
#define eoln(F) ((F).d ≡ '\n' ∨ eof(F))
#define erstat(F) ((F).f ≡ Λ ? −1 : ferror((F).f))
#define pascal_read(F, X) ((X) = (F).d, get(F))
#define read_ln(F) do get(F); while (¬eoln(F))
#define pascal_write(F, FMT, ...) fprintf(F.f, FMT, ##__VA_ARGS__)
#define write_ln(F, ...) pascal_write(F, __VA_ARGS__"\n")
#define wterm(FMT, ...) pascal_write(term_out, FMT, ##__VA_ARGS__)
#define wterm_ln(FMT, ...) wterm(FMT"\n", ##__VA_ARGS__)
#define wterm_cr pascal_write(term_out, "\n")
#define wlog(FMT, ...) pascal_write(log_file, FMT, ##__VA_ARGS__)
#define wlog_ln(FMT, ...) wlog(FMT"\n", ##__VA_ARGS__)
#define wlog_cr pascal_write(log_file, "\n")
```

See also sections 56, 57, 58, 59, 61, 62, 63, 64, 261, 262, 517, 698, 1354, 1504, and 1721.
This code is used in section 4.

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

⟨ Basic printing procedures 55 ⟩ +≡

```
static void print_ln(void)  /* prints an end-of-line */
{ switch (selector) {
  case term_and_log:
    { wterm_cr;
      wlog_cr;
      term_offset = 0;
      file_offset = 0;
    } break;
  case log_only:
    { wlog_cr;
      file_offset = 0;
    } break;
  case term_only:
    { wterm_cr;
      term_offset = 0;
    } break;
  case no_print: case pseudo: case new_string: do_nothing; break;
  default: write_ln(write_file[selector]);
  }
}  /* tally is not affected */
```

57. 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 55 ⟩ +≡

```
static void print_char(ASCII_code s)   /* prints a single character */
{ if (⟨ Character s is the current new-line character 243 ⟩)
      if (selector < pseudo) { print_ln();
        return;
      }
    switch (selector) {
    case term_and_log:
      { wterm("%c", xchr[s]);
        wlog("%c", xchr[s]);
        incr(term_offset);
        incr(file_offset);
        if (term_offset ≡ max_print_line) { wterm_cr;
          term_offset = 0;
        }
        if (file_offset ≡ max_print_line) { wlog_cr;
          file_offset = 0;
        }
      } break;
    case log_only:
      { wlog("%c", xchr[s]);
        incr(file_offset);
        if (file_offset ≡ max_print_line) print_ln();
      } break;
    case term_only:
      { wterm("%c", xchr[s]);
        incr(term_offset);
        if (term_offset ≡ max_print_line) print_ln();
      } break;
    case no_print: do_nothing; break;
    case pseudo:
      if (tally < trick_count) trick_buf[tally % error_line] = s; break;
    case new_string:
      { if (pool_ptr < pool_size) append_char(s);
      } break;   /* we drop characters if the string space is full */
    default: pascal_write(write_file[selector], "%c", xchr[s]);
    }
    incr(tally);
}
```

58. 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 55 ⟩ +≡

```
static void print(char *s)   /* the simple version */
{ if (s ≡ Λ) s = "???";   /* this can't happen */
  while (*s ≠ 0) print_char(*s++); }

static void printn(int s)   /* prints string s */
{ pool_pointer j;   /* current character code position */
  int nl;   /* new-line character to restore */
  if (s ≥ str_ptr) { print("???");
    return;
  }   /* this can't happen */
  else if (s < 256)
    if (s < 0) { print("???");
      return;
    }   /* can't happen */
    else { if (selector > pseudo) { print_char(s);
          return;   /* internal strings are not expanded */
      }
      if (((⟨ Character s is the current new-line character 243 ⟩)))
        if (selector < pseudo) { print_ln();
          return;
        }
      nl = new_line_char;
      new_line_char = −1;   /* temporarily disable new-line character */
      j = str_start[s];
      while (j < str_start[s + 1]) { print_char(so(str_pool[j]));
        incr(j);
      }
      new_line_char = nl;
      return;
    }
  j = str_start[s];
  while (j < str_start[s + 1]) { print_char(so(str_pool[j]));
    incr(j);
  }
}
```

59. 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 55 ⟩ +≡

```
static void slow_print(int s)    /* prints string s */
{ pool_pointer j;    /* current character code position */
  if ((s ≥ str_ptr) ∨ (s < 256)) printn(s);
  else { j = str_start[s];
    while (j < str_start[s + 1]) { printn(so(str_pool[j]));
      incr(j);
    }
  }
}
```

60. 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.

According to the conventions of TEX Live, we print the *dump_name* if no format identifier is known.

⟨ Initialize the output routines 54 ⟩ +≡

```
wterm("%s", banner);
if (format_ident ≡ 0) wterm_ln("␣(preloaded␣format=%s)", dump_name);
else { slow_print(format_ident);
  print_ln();
}
update_terminal;
```

61. 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 55 ⟩ +≡

```
static void print_nl(char *s)    /* prints string s at beginning of line */
{ if (((term_offset > 0) ∧ (odd(selector))) ∨
        ((file_offset > 0) ∧ (selector ≥ log_only))) print_ln();
  print(s);
}
```

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

⟨ Basic printing procedures 55 ⟩ +≡

```
static void printn_esc(str_number s)    /* prints escape character, then s */
{ int c;    /* the escape character code */
  ⟨ Set variable c to the current escape character 242 ⟩;
  if (c ≥ 0)
    if (c < 256) printn(c);
  slow_print(s);
}
static void print_esc(char *s)    /* the fast way */
{ int c;    /* the escape character code */
  ⟨ Set variable c to the current escape character 242 ⟩;
  if (c ≥ 0)
    if (c < 256) printn(c);
  print(s);
}
```

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

⟨ Basic printing procedures 55 ⟩ +≡

```
static void print_the_digs(eight_bits k)    /* prints dig[k − 1] . . . dig[0] */
{ while (k > 0) { decr(k);
    if (dig[k] < 10) print_char('0' + dig[k]);
    else print_char('A' − 10 + dig[k]);
  }
}
```

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

⟨ Basic printing procedures 55 ⟩ +≡
```
static void print_int(int n)    /* prints an integer in decimal form */
{ int k;    /* index to current digit; we assume that |n| < 10^23 */
  int m;    /* used to negate n in possibly dangerous cases */
  k = 0;
  if (n < 0) { print_char('-');
    if (n > −100000000) negate(n);
    else { m = −1 − n;
      n = m/10;
      m = (m % 10) + 1;
      k = 1;
      if (m < 10) dig[0] = m;
      else { dig[0] = 0;
        incr(n);
      }
    }
  }
  do { dig[k] = n % 10;
    n = n/10;
    incr(k);
  } while (¬(n ≡ 0));
  print_the_digs(k);
}
```

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

```
static void print_two(int n)    /* prints two least significant digits */
{ n = abs(n) % 100;
  print_char('0' + (n/10));
  print_char('0' + (n % 10));
}
```

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

```
static void print_hex(int n)    /* prints a positive integer in hexadecimal form */
{ int k;    /* index to current digit; we assume that 0 ≤ n < 16^22 */
  k = 0;
  print_char('"');
  do { dig[k] = n % 16;
    n = n/16;
    incr(k);
  } while (¬(n ≡ 0));
  print_the_digs(k);
}
```

67. 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* *printn*

68. 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`.

```
static void print_roman_int(int n)
{ pool_pointer j, k;      /* mysterious indices into mystery */
  nonnegative_integer u, v;      /* mysterious numbers */
  const char mystery[] = "m2d5c2l5x2v5i";

  j = 0;
  v = 1000;
  loop { while (n ≥ v) { print_char(so(mystery[j]));
        n = n − v;
     }
     if (n ≤ 0) return;      /* nonpositive input produces no output */
     k = j + 2;
     u = v/(so(mystery[k − 1]) − '0');
     if (mystery[k − 1] ≡ si('2')) { k = k + 2;
        u = u/(so(mystery[k − 1]) − '0');
     }
     if (n + u ≥ v) { print_char(so(mystery[k]));
        n = n + u;
     }
     else { j = j + 2;
        v = v/(so(mystery[j − 1]) − '0');
     }
  }
}
```

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

```
static void print_current_string(void)      /* prints a yet-unmade string */
{ pool_pointer j;      /* points to current character code */

  j = str_start[str_ptr];
  while (j < pool_ptr) { print_char(so(str_pool[j]));
     incr(j);
  }
}
```

70. 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(A)
        { wake_up_terminal;
          print(A);
          term_input();
        }   /* prints a string and gets a line of input */

  static void term_input(void)   /* gets a line from the terminal */
  { int k;   /* index into buffer */
    update_terminal;   /* now the user sees the prompt for sure */
    if (¬input_ln(&term_in, true)) 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)
      for (k = first; k ≤ last − 1; k++) printn(buffer[k]);
    print_ln();
    incr(selector);   /* restore previous status */
  }
```

71. 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.

⟨Error handling procedures 71⟩ ≡

```
static void print_err(char *s)
{ if (interaction ≡ error_stop_mode) wake_up_terminal;
  if (filelineerrorstylep) print_file_line();    /* TEX Live */
  else print_nl("!␣");
  print(s);
}
```

See also sections 77, 80, 81, 92, 93, and 94.
This code is used in section 4.

72. 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 */
```

⟨Global variables 13⟩ +≡

```
static int interaction;    /* current level of interaction */
```

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

```
if (interaction_option < 0) interaction = error_stop_mode;
else interaction = interaction_option;    /* TEX Live */
```

74. 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* 74⟩ ≡

```
if (interaction ≡ batch_mode) selector = no_print; else selector = term_only
```

This code is used in sections 1264 and 1336.

75. 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⟩ +≡

```
  static bool deletions_allowed; /* is it safe for error to call get_token? */
  static bool set_box_allowed; /* is it safe to do a \setbox assignment? */
  static int history; /* has the source input been clean so far? */
  static int error_count; /* the number of scrolled errors since the last paragraph ended */
```

76. 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 */
```

77. 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 71⟩ +≡

```
  static void normalize_selector(void);
  static void get_token(void);
  static void term_input(void);
  static void show_context(void);
  static void begin_file_reading(void);
  static void open_log_file(void);
  static void close_files_and_terminate(void);
  static void clear_for_error_prompt(void);
  static void give_err_help(void);
#ifdef DEBUG
  static void debug_help(void);
#else
#define debug_help() do_nothing
#endif
```

78. 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*(*A*) *help_line*[0] = *A*; }

#define *hlp2*(*A*, *B*) *help_line*[1] = *A*;
    help_line[0] = *B*; }

#define *hlp3*(*A*, *B*, *C*) *help_line*[2] = *A*;
    help_line[1] = *B*;
    help_line[0] = *C*; }

#define *hlp4*(*A*, *B*, *C*, *D*) *help_line*[3] = *A*;
    help_line[2] = *B*;
    help_line[1] = *C*;
    help_line[0] = *D*; }

#define *hlp5*(*A*, *B*, *C*, *D*, *E*) *help_line*[4] = *A*;
    help_line[3] = *B*;
    help_line[2] = *C*;
    help_line[1] = *D*;
    help_line[0] = *E*; }

#define *hlp6*(*A*, *B*, *C*, *D*, *E*, *F*) *help_line*[5] = *A*;
    help_line[4] = *B*;
    help_line[3] = *C*;
    help_line[2] = *D*;
    help_line[1] = *E*;
    help_line[0] = *F*; }

#define *help0* *help_ptr* = 0 /∗ sometimes there might be no help ∗/
#define *help1*(*A*) { *help_ptr* = 1; *hlp1*(*A*) /∗ use this with one help line ∗/
#define *help2*(*A*, *B*) { *help_ptr* = 2; *hlp2*(*A*, *B*) /∗ use this with two help lines ∗/
#define *help3*(*A*, *B*, *C*) { *help_ptr* = 3; *hlp3*(*A*, *B*, *C*) /∗ use this with three help lines ∗/
#define *help4*(*A*, *B*, *C*, *D*) { *help_ptr* = 4; *hlp4*(*A*, *B*, *C*, *D*) /∗ use this with four help lines ∗/
#define *help5*(*A*, *B*, *C*, *D*, *E*) { *help_ptr* = 5; *hlp5*(*A*, *B*, *C*, *D*, *E*) /∗ use this with five help lines ∗/
#define *help6*(*A*, *B*, *C*, *D*, *E*, *F*) { *help_ptr* = 6; *hlp6*(*A*, *B*, *C*, *D*, *E*, *F*)
    /∗ use this with six help lines ∗/

⟨ Global variables 13 ⟩ +≡
  static char ∗*help_line*[6]; /∗ helps for the next **error** ∗/
  static int *help_ptr*; /∗ the number of help lines present ∗/
  static bool *use_err_help*; /∗ should the *err_help* list be shown? ∗/

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

80. 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 71⟩ +≡
```
static void jump_out(void)
{ close_files_and_terminate();
  exit(0);
}
```

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

⟨Error handling procedures 71⟩ +≡
```
static void error(void)   /* completes the job of error reporting */
{ ASCII_code c;   /* what the user types */
  int s1, s2, s3, s4;   /* used to save global variables when deleting tokens */
  if (history < error_message_issued) history = error_message_issued;
  print_char('.');
  show_context();
  if (interaction ≡ error_stop_mode) ⟨Get user's advice and return 82⟩;
  incr(error_count);
  if (error_count ≡ 100) { print_nl("(That␣makes␣100␣errors;␣please␣try␣again.)");
    history = fatal_error_stop;
    jump_out();
  }
  ⟨Put help message on the transcript file 89⟩;
}
```

82. ⟨Get user's advice and **return** 82⟩ ≡
```
loop { resume:
  if (interaction ≠ error_stop_mode) return;
  clear_for_error_prompt();
  prompt_input("?␣");
  if (last ≡ first) return;
  c = buffer[first];
  if (c ≥ 'a') c = c + 'A' − 'a';   /* convert to uppercase */
  ⟨Interpret code c and return if done 83⟩;
}
```
This code is used in section 81.

83. 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 83⟩ ≡
  switch (c) {
  case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8':
    case '9':
    if (deletions_allowed) ⟨Delete c − "0" tokens and goto resume 87⟩ break;

#ifdef DEBUG
  case 'D':
    { debug_help();
      goto resume; }
#endif
  case 'E':
    if (base_ptr > 0)
      if (input_stack[base_ptr].name_field ≥ 256) { 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();
      } break;
  case 'H': ⟨Print the help information and goto resume 88⟩
  case 'I': ⟨Introduce new material from the terminal and return 86⟩
  case 'Q': case 'R': case 'S': ⟨Change the interaction level and return 85⟩
  case 'X':
    { interaction = scroll_mode;
      jump_out();
    } break;
  default: do_nothing;
  }
  ⟨Print the menu of available options 84⟩
```

This code is used in section 82.

84. ⟨Print the menu of available options 84⟩ ≡

```
  { 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)
      if (input_stack[base_ptr].name_field ≥ 256) print("E␣to␣edit␣your␣file,");
    if (deletions_allowed) print_nl("1␣or␣...␣or␣9␣to␣ignore␣the␣next␣1␣to␣9␣tokens␣of␣input,");
    print_nl("H␣for␣help,␣X␣to␣quit.");
  }
```

This code is used in section 83.

85. 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** 85 ⟩ ≡

```
{ error_count = 0;
  interaction = batch_mode + c − 'Q';
  print("OK,␣entering␣");
  switch (c) {
  case 'Q':
    { print_esc("batchmode");
      decr(selector);
    } break;
  case 'R': print_esc("nonstopmode"); break;
  case 'S': print_esc("scrollmode");
  }      /* there are no other cases */
  print("...");
  print_ln();
  update_terminal;
  return;
}
```

This code is used in section 83.

86. 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** 86 ⟩ ≡

```
{ 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) { loc = first + 1;
    buffer[first] = '␣';
  }
  else { prompt_input("insert>");
    loc = first;
  }
  first = last;
  cur_input.limit_field = last − 1;      /* no end_line_char ends this line */
  return;
}
```

This code is used in section 83.

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

⟨ Delete $c-$ "0" tokens and **goto** *resume* 87 ⟩ ≡

```
{ 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'))
    c = c * 10 + buffer[first + 1] − '0' * 11;
  else c = c − '0';
  while (c > 0) { get_token();    /* one-level recursive call of error is possible */
    decr(c);
  }
  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 resume;
}
```

This code is used in section 83.

88. ⟨ Print the help information and **goto** *resume* 88 ⟩ ≡

```
{ if (use_err_help) { give_err_help();
    use_err_help = false;
  }
  else { if (help_ptr ≡ 0) help2("Sorry,␣I␣don't␣know␣how␣to␣help␣in␣this␣situation.",
        "Maybe␣you␣should␣try␣asking␣a␣human?");
    do { decr(help_ptr);
      print(help_line[help_ptr]);
      print_ln();
    } while (¬(help_ptr ≡ 0));
  }
  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 resume;
}
```

This code is used in section 83.

89. ⟨ Put help message on the transcript file 89 ⟩ ≡

```
if (interaction > batch_mode) decr(selector);   /* avoid terminal output */
if (use_err_help) { print_ln();
  give_err_help();
}
else
  while (help_ptr > 0) { decr(help_ptr);
    print_nl(help_line[help_ptr]);
  }
print_ln();
if (interaction > batch_mode) incr(selector);   /* re-enable terminal output */
print_ln()
```

This code is used in section 81.

90. 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:

```
static void int_error(int n)
{ print("␣(");
  print_int(n);
  print_char(')');
  error ( ) ;
}
```

91. 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.

```
static void normalize_selector(void)
{ if (log_opened) selector = term_and_log;
  else selector = term_only;
  if (job_name ≡ 0) open_log_file();
  if (interaction ≡ batch_mode) decr(selector);
}
```

92. The following procedure prints TEX's last words before dying.

```
#define succumb
        { if (interaction ≡ error_stop_mode) interaction = scroll_mode;   /* no more interaction */
          if (log_opened)
            error ( ) ;
          if (interaction > batch_mode) debug_help();
          history = fatal_error_stop;
          jump_out();   /* irrecoverable error */
        }
```

⟨ Error handling procedures 71 ⟩ +≡

```
static void fatal_error(char *s)   /* prints s, and that's it */
{ normalize_selector();
  print_err("Emergency␣stop");
  help1(s);
  succumb;
}
```

93. Here is the most dreaded error message.

⟨ Error handling procedures 71 ⟩ +≡

```
static void overflow(char *s, int n)  /* stop due to finiteness */
{ 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;
}
```

94. 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 71 ⟩ +≡

```
static void confusion(char *s)  /* consistency check violated; s tells where */
{ normalize_selector();
  if (history < error_message_issued) { print_err("This␣can't␣happen␣(");
    print(s);
    print_char(')');
    help1("I'm␣broken.␣Please␣show␣this␣to␣someone␣who␣can␣fix␣can␣fix");
  }
  else { 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.");
  }
  succumb;
}
```

95. 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
        { if (interrupt ≠ 0) pause_for_instructions();
        }
```

⟨ Global variables 13 ⟩ +≡

```
static int interrupt;  /* should TeX pause for instructions? */
static bool OK_to_interrupt;  /* should interrupts be observed? */
```

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

```
interrupt = 0;
OK_to_interrupt = true;
```

97. 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.

```
static void pause_for_instructions(void)
{ if (OK_to_interrupt) { interaction = error_stop_mode;
    if ((selector ≡ log_only) ∨ (selector ≡ no_print)) 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;
  }
}
```

98. 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 / with a possibly negative numerator. These are harmless; see / 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 % as well as for /.)

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

```
static int half(int x)
{ if (odd(x)) return (x + 1)/2;
  else return x/2;
}
```

100. 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⟩ +≡

```
typedef int scaled;   /* this type is used for scaled integers */
typedef int32_t nonnegative_integer;   /* 0 ≤ x < 2^31 */
typedef int8_t small_number;   /* this type is self-explanatory */
```

101. The following function is used to create a scaled integer from a given decimal fraction $(.d_0d_1\ldots d_{k-1})$, where $0 \le k \le 17$. The digit d_i is given in $dig[i]$, and the calculation produces a correctly rounded result.

```
static scaled round_decimals(small_number k)   /* converts a decimal fraction */
{ int a;   /* the accumulator */
  a = 0;
  while (k > 0) { decr(k);
    a = (a + dig[k] * two)/10;
  }
  return (a + 1)/2;
}
```

102. 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.

```
static void print_scaled(scaled s)    /* prints scaled real, rounded to five digits */
{ scaled delta;    /* amount of allowable inaccuracy */
  if (s < 0) { print_char('-');
    negate(s);    /* print the sign, if negative */
  }
  print_int(s/unity);    /* print the integer part */
  print_char('.');
  s = 10 * (s % unity) + 5;
  delta = 10;
  do {
    if (delta > unity) s = s + °100000 − 50000;    /* round the last digit */
    print_char('0' + (s/unity));
    s = 10 * (s % unity);
    delta = delta * 10;
  } while (¬(s ≤ delta));
}
```

103. 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*) *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, *rem*, holds the remainder after a division.

⟨ Global variables 13 ⟩ +≡

```
static bool arith_error;    /* has arithmetic overflow occurred recently? */
static scaled rem;    /* amount subtracted to get an exact division */
```

104. 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(A, B, C)  mult_and_add(A, B, C, °7777777777)
#define mult_integers(A, B)  mult_and_add(A, B, 0, °17777777777)

  static scaled mult_and_add(int n, scaled x, scaled y, scaled max_answer)
  { if (n < 0) { negate(x);
        negate(n);
     }
     if (n ≡ 0) return y;
     else if (((x ≤ (max_answer − y)/n) ∧ (−x ≤ (max_answer + y)/n))) return n ∗ x + y;
     else { arith_error = true;
        return 0;
     }
  }
```

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

```
  static scaled x_over_n(scaled x, int n)
  { bool negative;    /∗ should rem be negated? ∗/
     scaled x_over_n;

     negative = false;
     if (n ≡ 0) { arith_error = true;
        x_over_n = 0;
        rem = x;
     }
     else { if (n < 0) { negate(x);
           negate(n);
           negative = true;
        }
        if (x ≥ 0) { x_over_n = x/n;
           rem = x % n;
        }
        else { x_over_n = −((−x)/n);
           rem = −((−x) % n);
        }
     }
     if (negative) negate(rem);
     return x_over_n;
  }
```

106. Then comes the multiplication of a scaled number by a fraction $n/(\textbf{double})\ d$, 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.

```
static scaled xn_over_d(scaled x, int n, int d)
{ bool positive;    /* was x ≥ 0? */
  nonnegative_integer t, u, v;    /* intermediate quantities */
  scaled xn_over_d;
  if (x ≥ 0) positive = true;
  else { negate(x);
    positive = false;
  }
  t = (x % °100000) * n;
  u = (x/°100000) * n + (t/°100000);
  v = (u % d) * °100000 + (t % °100000);
  if (u/d ≥ °100000) arith_error = true;
  else u = °100000 * (u/d) + (v/d);
  if (positive) { xn_over_d = u;
    rem = v % d;
  }
  else { xn_over_d = −u;
    rem = −(v % d);
  }
  return xn_over_d;
}
```

107. 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 */

⟨ Declare PRoTE arithmetic routines 1628 ⟩

```
static halfword badness(scaled t, scaled s)   /* compute badness, given t ≥ 0 */
{ int r;   /* approximation to αt/s, where α³ ≈ 100 · 2¹⁸ */
  if (t ≡ 0) return 0;
  else if (s ≤ 0) return inf_bad;
  else { if (t ≤ 7230584) r = (t * 297)/s;   /* 297³ = 99.94 × 2¹⁸ */
    else if (s ≥ 1663497) r = t/(s/297);
    else r = t;
    if (r > 1290) return inf_bad;   /* 1290³ < 2³¹ < 1291³ */
    else return (r * r * r + °400000)/°1000000;
  }   /* that was r³/2¹⁸, rounded to the nearest integer */
}
```

108. 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 **int** (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(A) A = 0.0   /* store the representation of zero ratio */
#define set_glue_ratio_one(A) A = 1.0   /* store the representation of unit ratio */
#define unfix(A) ((double)(A))   /* convert from glue_ratio to type double */
#define fix(A) ((glue_ratio)(A))   /* convert from double to type glue_ratio */
#define float_constant(A) ((double)(A))   /* convert int constant to double */
```

⟨ Types in the outer block 18 ⟩ +≡

```
#if __SIZEOF_FLOAT__ ≡ 4
  typedef float float32_t;
#else
#error float type must have size 4
#endif
  typedef float glue_ratio;   /* one-word representation of a glue expansion factor */
```

109. 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:

$x.i$	(an **int**)
$x.sc$	(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 **double** 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* 65535 /∗ largest allowable value in a *quarterword* ∗/
#**define** *min_halfword* 0 /∗ smallest allowable value in a **halfword** ∗/
#**define** *max_halfword* #3FFFFFFF /∗ largest allowable value in a **halfword** ∗/

110. 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 ⟩ +≡

```
#ifdef INIT
  if ((mem_min ≠ mem_bot) ∨ (mem_max ≠ mem_top)) bad = 10;
#endif
  if ((mem_min > mem_bot) ∨ (mem_max < mem_top)) bad = 10;
  if ((min_quarterword > 0) ∨ (max_quarterword < 127)) bad = 11;
  if ((min_halfword > 0) ∨ (max_halfword < 32767)) bad = 12;
  if ((min_quarterword < min_halfword) ∨
        (max_quarterword > max_halfword)) bad = 13;
  if ((mem_min < min_halfword) ∨ (mem_max ≥ max_halfword) ∨
        (mem_bot − mem_min > max_halfword + 1)) bad = 14;
  if ((font_base < min_quarterword) ∨ (font_max > max_quarterword)) bad = 15;
  if (font_max > font_base + 256) bad = 16;
  if ((save_size > max_halfword) ∨ (max_strings > max_halfword)) bad = 17;
  if (buf_size > max_halfword) bad = 18;
  if (max_quarterword − min_quarterword < 255) bad = 19;
```

111. 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* $\equiv 0$.

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

112. The reader should study the following definitions closely:

```
#define sc i   /* scaled data is equivalent to int */
⟨Types in the outer block 18⟩ +≡
  typedef uint16_t quarterword;   /* 1/4 of a word */
  typedef int32_t halfword;   /* 1/2 of a word */
  typedef int8_t two_choices;   /* used when there are two variants in a record */
  typedef int8_t four_choices;   /* used when there are four variants in a record */
  typedef struct {
    halfword rh;
    union {
      halfword lh;
      struct {
        quarterword b0;
        quarterword b1;
      };
    };
  } two_halves;
  typedef struct {
    quarterword b0;
    quarterword b1;
    quarterword b2;
    quarterword b3;
  } four_quarters;
  typedef struct {
    union {
      int i;
      glue_ratio gr;
      two_halves hh;
      four_quarters qqqq;
    };
  } memory_word;
  typedef struct { FILE *f; memory_word d; } word_file;
```

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

```
#ifdef DEBUG
  static void print_word(memory_word w)      /* prints w in all ways */
  { print_int(w.i);
    print_char(' ');
    print_scaled(w.sc);
    print_char(' ');
    print_scaled(round(unity * unfix(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);
  }
#endif
```

114. 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 ⟩ +≡
  static pointer temp_ptr;   /* a pointer variable for occasional emergency use */
```

115. 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 ⟩ +≡
  static memory_word mem0[mem_max − mem_min + 1], *const mem = mem0 − mem_min;
    /* the big dynamic storage area */
  static pointer lo_mem_max;   /* the largest location of variable-size memory in use */
  static pointer hi_mem_min;   /* the smallest location of one-word memory in use */
```

116. 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 **#ifdef** `STAT` ... **#endif** 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 ⟩ +≡
  static int var_used, dyn_used;   /* how much memory is in use */
#ifdef STAT
#define incr_dyn_used incr(dyn_used)
#define decr_dyn_used decr(dyn_used)
#else
#define incr_dyn_used
#define decr_dyn_used
#endif
```

117. 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(A) mem[A].hh.rh      /* the link field of a memory word */
#define info(A) mem[A].hh.lh      /* the info field of a memory word */
```

⟨ Global variables 13 ⟩ +≡

```
  static pointer avail;      /* head of the list of available one-word nodes */
  static pointer mem_end;    /* the last one-word node used in mem */
```

118. 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* 291 ⟩
⟨ Declare the procedure called *runaway* 305 ⟩

119. 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.

```
  static pointer get_avail(void)      /* single-word node allocation */
  { pointer p;      /* the new node being got */
    p = avail;      /* get top location in the avail stack */
    if (p ≠ null) avail = link(avail);      /* and pop it off */
    else if (mem_end < mem_max)      /* or go into virgin territory */
    { incr(mem_end);
      p = mem_end;
    }
    else { decr(hi_mem_min);
      p = hi_mem_min;
      if (hi_mem_min ≤ lo_mem_max) { 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 */
      }
    }
    link(p) = null;      /* provide an oft-desired initialization of the new node */
#ifdef STAT
    incr(dyn_used);
#endif      /* maintain statistics */
    return p;
  }
```

120. 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(A)   /* single-word node liberation */
        { link(A) = avail;
          avail = A;
          decr_dyn_used;
        }
```

121. 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(A)
        { A = avail;   /* avoid get_avail if possible, to save time */
          if (A ≡ null) A = get_avail();
          else { avail = link(A);
            link(A) = null;
            incr_dyn_used;
          }
        }
```

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

```
  static void flush_list(pointer p)   /* makes list of single-word nodes available */
  { pointer q, r;   /* list traversers */
    if (p ≠ null) { r = p;
      do { q = r;
        r = link(r);
#ifdef STAT
        decr(dyn_used);
#endif
      } while (¬(r ≡ null));   /* now q is the last node on the list */
      link(q) = avail;
      avail = p;
    }
  }
```

123. 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(A) (link(A) ≡ empty_flag)   /* tests for empty node */
#define node_size(A) info(A)   /* the size field in empty variable-size nodes */
#define llink(A) info(A + 1)   /* left link in doubly-linked list of empty nodes */
#define rlink(A) link(A + 1)   /* right link in doubly-linked list of empty nodes */
⟨Global variables 13⟩ +≡
  static pointer rover;   /* points to some node in the list of empties */
```

124. 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*.

```
static pointer get_node(int s)    /* variable-size node allocation */
{ pointer p;    /* the node currently under inspection */
  pointer q;    /* the node physically after node p */
  int r;    /* the newly allocated node, or a candidate for this honor */
  int t;    /* temporary register */
restart: p = rover;    /* start at some free node in the ring */
  do { ⟨Try to allocate within node p and its physical successors, and goto found if allocation was
          possible 126⟩;
    p = rlink(p);    /* move to the next node in the ring */
  } while (¬(p ≡ rover));    /* repeat until the whole list has been traversed */
  if (s ≡ °10000000000) { return max_halfword;
  }
  if (lo_mem_max + 2 < hi_mem_min)
    if (lo_mem_max + 2 ≤ mem_bot + max_halfword)
      ⟨Grow more variable-size memory and goto restart 125⟩;
  overflow("main␣memory␣size", mem_max + 1 − mem_min);    /* sorry, nothing satisfactory is left */
found: link(r) = null;    /* this node is now nonempty */
#ifdef STAT
  var_used = var_used + s;    /* maintain usage statistics */
#endif
  return r;
}
```

125. 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 125⟩ ≡
{ if (hi_mem_min − lo_mem_max ≥ 1998) t = lo_mem_max + 1000;
  else t = lo_mem_max + 1 + (hi_mem_min − lo_mem_max)/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) 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;
}
```

This code is used in section 124.

126. 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 126 ⟩ ≡

```
q = p + node_size(p);   /* find the physical successor */
while (is_empty(q))   /* merge node p with node q */
{ t = rlink(q);
  if (q ≡ rover) rover = t;
  llink(t) = llink(q);
  rlink(llink(q)) = t;
  q = q + node_size(q);
}
r = q − s;
if (r > p + 1) ⟨ Allocate from the top of node p and goto found 127 ⟩;
if (r ≡ p)
  if (rlink(p) ≠ p) ⟨ Allocate entire node p and goto found 128 ⟩;
node_size(p) = q − p   /* reset the size in case it grew */
```

This code is used in section 124.

127. ⟨ Allocate from the top of node p and **goto** *found* 127 ⟩ ≡

```
{ node_size(p) = r − p;   /* store the remaining size */
  rover = p;   /* start searching here next time */
  goto found;
}
```

This code is used in section 126.

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

⟨ Allocate entire node p and **goto** *found* 128 ⟩ ≡

```
{ rover = rlink(p);
  t = llink(p);
  llink(rover) = t;
  rlink(t) = rover;
  goto found;
}
```

This code is used in section 126.

129. 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.

```
static void free_node(pointer p, halfword s)   /* variable-size node liberation */
{ pointer q;   /* llink(rover) */
  node_size(p) = s;
  link(p) = empty_flag;
  q = llink(rover);
  llink(p) = q;
  rlink(p) = rover;   /* set both links */
  llink(rover) = p;
  rlink(q) = p;   /* insert p into the ring */
#ifdef STAT
  var_used = var_used − s;
#endif   /* maintain statistics */
}
```

130. 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.

```
#ifdef INIT
  static void sort_avail(void)    /* sorts the available variable-size nodes by location */
  { pointer p, q, r;    /* indices into mem */
    pointer old_rover;    /* initial rover setting */
    p = get_node(°10000000000);    /* merge adjacent free areas */
    p = rlink(rover);
    rlink(rover) = max_halfword;
    old_rover = rover;
    while (p ≠ old_rover) ⟨Sort p into the list starting at rover and advance p to rlink(p) 131⟩;
    p = rover;
    while (rlink(p) ≠ max_halfword) { llink(rlink(p)) = p;
      p = rlink(p);
    }
    rlink(p) = rover;
    llink(rover) = p;
  }
#endif
```

131. 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) 131⟩ ≡
  if (p < rover) { q = p;
    p = rlink(q);
    rlink(q) = rover;
    rover = q;
  }
  else { q = rover;
    while (rlink(q) < p) q = rlink(q);
    r = rlink(p);
    rlink(p) = rlink(q);
    rlink(q) = p;
    p = r;
  }
```

This code is used in section 130.

132. 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*(*A*) *mem*[*A*].*hh*.*b0* /∗ identifies what kind of node this is ∗/
#define *subtype*(*A*) *mem*[*A*].*hh*.*b1* /∗ secondary identification in some cases ∗/

133. 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 $font \equiv font\_base$, 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*(*A*) ($A \geq hi\_mem\_min$) /∗ does the argument point to a *char_node*? ∗/
#define *font*(*A*) *type*(*A*) /∗ the font code in a *char_node* ∗/
#define *character*(*A*) *subtype*(*A*) /∗ the character code in a *char_node* ∗/

134. 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(A) mem[A + width_offset].sc /* width of the box, in sp */
#define depth(A) mem[A + depth_offset].sc /* depth of the box, in sp */
#define height(A) mem[A + height_offset].sc /* height of the box, in sp */
#define shift_amount(A) mem[A + 4].sc /* repositioning distance, in sp */
#define list_offset 5 /* position of list_ptr field in a box node */
#define list_ptr(A) link(A + list_offset) /* beginning of the list inside the box */
#define glue_order(A) subtype(A + list_offset) /* applicable order of infinity */
#define glue_sign(A) type(A + 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(A) mem[A + glue_offset].gr /* a word of type glue_ratio for glue setting */
```

135. 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.)

```
static pointer new_null_box(void) /* creates a new box node */
{ pointer p; /* the new node */
  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));
  return p;
}
```

136. 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 */
```

137. 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*(A) $(A \equiv null\_flag)$ /* tests for a running dimension */

138. 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.

static pointer *new_rule*(**void**)
{ **pointer** p; /* the new node */
  $p = get\_node(rule\_node\_size)$;
  $type(p) = rule\_node$;
  $subtype(p) = 0$; /* the *subtype* is not used */
  $width(p) = null\_flag$;
  $depth(p) = null\_flag$;
  $height(p) = null\_flag$;
  return p;
}

139. 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*(A) $mem[A+1].i$ /* the *floating_penalty* to be used */
#**define** *ins_ptr*(A) $info(A+4)$ /* the vertical list to be inserted */
#**define** *split_top_ptr*(A) $link(A+4)$ /* the *split_top_skip* to be used */

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

#**define** *mark_node* 4 /* *type* of a mark node */
#**define** *small_node_size* 2 /* number of words to allocate for most node types */
#**define** *mark_ptr*(A) $link(A+1)$ /* head of the token list for a mark */
#**define** *mark_class*(A) $info(A+1)$ /* the mark class */

141. 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*(A) $mem[A+1].i$ /* vertical list to be moved out of horizontal list */

142. 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(A) A+1    /* the word where the ligature is to be found */
#define lig_ptr(A) link(lig_char(A))    /* the list of characters */
```

143. 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.

```
static pointer new_ligature(quarterword f, quarterword c, pointer q)
{ pointer p;    /* the new node */
  p = get_node(small_node_size);
  type(p) = ligature_node;
  font(lig_char(p)) = f;
  character(lig_char(p)) = c;
  lig_ptr(p) = q;
  subtype(p) = 0;
  return p;
}
static pointer new_lig_item(quarterword c)
{ pointer p;    /* the new node */
  p = get_node(small_node_size);
  character(p) = c;
  lig_ptr(p) = null;
  return p;
}
```

144. 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(A) subtype(A)   /* how many subsequent nodes to replace */
#define pre_break(A) llink(A)   /* text that precedes a discretionary break */
#define post_break(A) rlink(A)   /* text that follows a discretionary break */

  static pointer new_disc(void)   /* creates an empty disc_node */
  { pointer p;   /* the new node */
    p = get_node(small_node_size);
    type(p) = disc_node;
    replace_count(p) = 0;
    pre_break(p) = null;
    post_break(p) = null;
    return p;
  }
```

145. 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 */
```

146. 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 */

  static pointer new_math(scaled w, small_number s)
  { pointer p;   /* the new node */
    p = get_node(small_node_size);
    type(p) = math_node;
    subtype(p) = s;
    width(p) = w;
    return p;
  }
```

147. 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*(*A*) (*type*(*A*) < *math_node*)
#**define** *non_discardable*(*A*) (*type*(*A*) < *math_node*)

148. 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*(*A*) *llink*(*A*) /* pointer to a glue specification */
#**define** *leader_ptr*(*A*) *rlink*(*A*) /* pointer to box or rule node for leaders */

149. 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*(*A*) *link*(*A*) /* reference count of a glue specification */
#**define** *stretch*(*A*) *mem*[*A* + 2].*sc* /* the stretchability of this glob of glue */
#**define** *shrink*(*A*) *mem*[*A* + 3].*sc* /* the shrinkability of this glob of glue */
#**define** *stretch_order*(*A*) *type*(*A*) /* order of infinity for stretching */
#**define** *shrink_order*(*A*) *subtype*(*A*) /* 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⟩ +≡
  typedef int8_t glue_ord; /* infinity to the 0, 1, 2, or 3 power */

150. 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.

```
static pointer new_spec(pointer p)    /* duplicates a glue specification */
{ pointer q;    /* the new spec */
  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);
  return q;
}
```

151. 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`.

```
static pointer new_param_glue(small_number n){ pointer p;    /* the new node */
    pointer q;    /* the glue specification */
    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 223⟩;
    glue_ptr(p) = q;
    incr(glue_ref_count(q));
    return p;
    }
```

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

```
static pointer new_glue(pointer q)
{ pointer p;    /* the new node */
  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));
  return p;
}
```

153. 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.

```
static pointer new_skip_param(small_number n)
{ pointer p;     /* the new node */
  temp_ptr = new_spec(⟨Current mem equivalent of glue parameter number n 223⟩);
  p = new_glue(temp_ptr);
  glue_ref_count(temp_ptr) = null;
  subtype(p) = n + 1;
  return p;
}
```

154. 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 */
```

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

```
static pointer new_kern(scaled w)
{ pointer p;     /* the new node */
  p = get_node(small_node_size);
  type(p) = kern_node;
  subtype(p) = normal;
  width(p) = w;
  return p;
}
```

156. 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(A) mem[A + 1].i     /* the added cost of breaking a list here */
```

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

```
static pointer new_penalty(int m)
{ pointer p;   /* the new node */
  p = get_node(small_node_size);
  type(p) = penalty_node;
  subtype(p) = 0;   /* the subtype is not used */
  penalty(p) = m;
  return p;
}
```

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

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

160. 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.

161. 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 */
```

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

```
⟨ Local variables for initialization 19 ⟩ +≡
  int k; /* index into mem, eqtb, etc. */
```

163. ⟨Initialize table entries (done by INITEX only) 163⟩ ≡

```
for (k = mem_bot + 1; k ≤ lo_mem_stat_max; k++) mem[k].sc = 0;
     /* all glue dimensions are zeroed */
k = mem_bot; while (k ≤ lo_mem_stat_max)     /* set first words of glue specifications */
{ glue_ref_count(k) = null + 1;
  stretch_order(k) = normal;
  shrink_order(k) = normal;
  k = k + glue_spec_size;
}
stretch(fil_glue) = unity;
stretch_order(fil_glue) = fil;
stretch(fill_glue) = unity;
stretch_order(fill_glue) = fill;
stretch(ss_glue) = unity;
stretch_order(ss_glue) = fil;
shrink(ss_glue) = unity;
shrink_order(ss_glue) = fil;
stretch(fil_neg_glue) = −unity;
stretch_order(fil_neg_glue) = fil;
rover = lo_mem_stat_max + 1;
link(rover) = empty_flag;     /* now initialize the dynamic memory */
node_size(rover) = 1000;     /* which is a 1000-word available node */
llink(rover) = rover;
rlink(rover) = rover;
lo_mem_max = rover + 1000;
link(lo_mem_max) = null;
info(lo_mem_max) = null;
for (k = hi_mem_stat_min; k ≤ mem_top; k++) mem[k] = mem[lo_mem_max];     /* clear list heads */
⟨Initialize the special list heads and constant nodes 789⟩;
avail = null;
mem_end = mem_top;
hi_mem_min = hi_mem_stat_min;     /* initialize the one-word memory */
var_used = lo_mem_stat_max + 1 − mem_bot;
dyn_used = hi_mem_stat_usage;     /* initialize statistics */
```

See also sections 221, 227, 231, 239, 249, 257, 551, 945, 950, 1215, 1300, 1368, 1383, 1500, 1524, 1542, and 1581.

This code is used in section 8.

164. 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 *is_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⟩ +≡

```
#ifdef DEBUG
  static bool is_free0[mem_max − mem_min + 1], *const is_free = is_free0 − mem_min;
    /* free cells */
    static bool was_free0[mem_max − mem_min + 1], *const was_free = was_free0 − mem_min;
    /* previously free cells */
    static pointer was_mem_end, was_lo_max, was_hi_min;
    /* previous mem_end, lo_mem_max, and hi_mem_min */
    static bool panicking;     /* do we want to check memory constantly? */
#endif
```

165. ⟨Set initial values of key variables 21⟩ +≡
```
#ifdef DEBUG
  was_mem_end = mem_min;    /* indicate that everything was previously free */
  was_lo_max = mem_min;
  was_hi_min = mem_max;
  panicking = false;
#endif
```

166. 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.

```
#ifdef DEBUG
  static void check_mem(bool print_locs)
  {    /* loop exits */
    int p, q;    /* current locations of interest in mem */
    bool clobbered;    /* is something amiss? */
    for (p = mem_min; p ≤ lo_mem_max; p++) is_free[p] = false;
        /* you can probably do this faster */
    for (p = hi_mem_min; p ≤ mem_end; p++) is_free[p] = false;    /* ditto */
    ⟨Check single-word avail list 167⟩;
    ⟨Check variable-size avail list 168⟩;
    ⟨Check flags of unavailable nodes 169⟩;
    if (print_locs) ⟨Print newly busy locations 170⟩;
    for (p = mem_min; p ≤ lo_mem_max; p++) was_free[p] = is_free[p];
    for (p = hi_mem_min; p ≤ mem_end; p++) was_free[p] = is_free[p];
        /* was_free = is_free might be faster */
    was_mem_end = mem_end;
    was_lo_max = lo_mem_max;
    was_hi_min = hi_mem_min;
  }
#endif
```

167. ⟨Check single-word *avail* list 167⟩ ≡
```
  p = avail;
  q = null;
  clobbered = false;
  while (p ≠ null) { if ((p > mem_end) ∨ (p < hi_mem_min)) clobbered = true;
    else if (is_free[p]) clobbered = true;
    if (clobbered) { print_nl("AVAIL␣list␣clobbered␣at␣");
      print_int(q);
      goto done1;
    }
    is_free[p] = true;
    q = p;
    p = link(q);
  }
  done1:
```
This code is used in section 166.

168. ⟨ Check variable-size *avail* list 168 ⟩ ≡

```
p = rover;
q = null;
clobbered = false;
do {
  if ((p ≥ lo_mem_max) ∨ (p < mem_min)) clobbered = true;
  else if ((rlink(p) ≥ lo_mem_max) ∨ (rlink(p) < mem_min)) clobbered = true;
  else if (¬(is_empty(p)) ∨ (node_size(p) < 2) ∨
        (p + node_size(p) > lo_mem_max) ∨
        (llink(rlink(p)) ≠ p)) clobbered = true;
  if (clobbered) { print_nl("Double-AVAIL␣list␣clobbered␣at␣");
    print_int(q);
    goto done2;
  }
  for (q = p; q ≤ p + node_size(p) − 1; q++)    /* mark all locations free */
  { if (is_free[q]) { print_nl("Doubly␣free␣location␣at␣");
      print_int(q);
      goto done2;
    }
    is_free[q] = true;
  }
  q = p;
  p = rlink(p);
} while (¬(p ≡ rover)); done2:
```

This code is used in section 166.

169. ⟨ Check flags of unavailable nodes 169 ⟩ ≡

```
p = mem_min;
while (p ≤ lo_mem_max)    /* node p should not be empty */
{ if (is_empty(p)) { print_nl("Bad␣flag␣at␣");
    print_int(p);
  }
  while ((p ≤ lo_mem_max) ∧ ¬is_free[p]) incr(p);
  while ((p ≤ lo_mem_max) ∧ is_free[p]) incr(p);
}
```

This code is used in section 166.

170. ⟨ Print newly busy locations 170 ⟩ ≡

```
{ print_nl("New␣busy␣locs:");
  for (p = mem_min; p ≤ lo_mem_max; p++)
    if (¬is_free[p] ∧ ((p > was_lo_max) ∨ was_free[p])) { print_char('␣');
      print_int(p);
    }
  for (p = hi_mem_min; p ≤ mem_end; p++)
    if (¬is_free[p] ∧ ((p < was_hi_min) ∨ (p > was_mem_end) ∨ was_free[p])) { print_char('␣');
      print_int(p);
    }
}
```

This code is used in section 166.

171. 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.

```
#ifdef DEBUG
  static void search_mem(pointer p)    /* look for pointers to p */
  { int q;    /* current position being searched */
    for (q = mem_min; q ≤ lo_mem_max; q++) { if (link(q) ≡ p) { print_nl("LINK(");
        print_int(q);
        print_char(')');
      }
      if (info(q) ≡ p) { print_nl("INFO(");
        print_int(q);
        print_char(')');
      }
    }
    for (q = hi_mem_min; q ≤ mem_end; q++) { if (link(q) ≡ p) { print_nl("LINK(");
        print_int(q);
        print_char(')');
      }
      if (info(q) ≡ p) { print_nl("INFO(");
        print_int(q);
        print_char(')');
      }
    }
    ⟨Search eqtb for equivalents equal to p 254⟩;
    ⟨Search save_stack for equivalents that point to p 284⟩;
    ⟨Search hyph_list for pointers to p 932⟩;
  }
#endif
```

172. 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⟩ +≡
```
static int font_in_short_display;   /* an internal font number */
```

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

```
static void short_display(int p)   /* prints highlights of list p */
{ int n;   /* for replacement counts */
  while (p > mem_min) { if (is_char_node(p)) { if (p ≤ mem_end) {
          if (font(p) ≠ font_in_short_display) { if ((font(p) < font_base) ∨ (font(p) > font_max))
            print_char('*');
          else ⟨Print the font identifier for font(p) 266⟩;
          print_char('␣');
          font_in_short_display = font(p);
        }
        print_ASCII(qo(character(p)));
      }
    }
    else ⟨Print a short indication of the contents of node p 174⟩;
    p = link(p);
  }
}
```

174. ⟨Print a short indication of the contents of node p 174⟩ ≡
```
switch (type(p)) {
case hlist_node: case vlist_node: case ins_node: case whatsit_node: case mark_node:
  case adjust_node: case unset_node: print("[]"); break;
case rule_node: print_char('|'); break;
case glue_node:
  if (glue_ptr(p) ≠ zero_glue) print_char('␣'); break;
case math_node: print_char('$'); break;
case ligature_node: short_display(lig_ptr(p)); break;
case disc_node:
  { short_display(pre_break(p));
    short_display(post_break(p));
    n = replace_count(p);
    while (n > 0) { if (link(p) ≠ null) p = link(p);
      decr(n);
    }
  } break;
default: do_nothing;
}
```
This code is used in section 173.

175. 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.

```
static void print_font_and_char(int p)   /* prints char_node data */
{ if (p > mem_end) print_esc("CLOBBERED.");
  else { if ((font(p) < font_base) ∨ (font(p) > font_max)) print_char('*');
    else ⟨Print the font identifier for font(p) 266⟩;
    print_char('␣');
    print_ASCII(qo(character(p)));
  }
}
static void print_mark(int p)   /* prints token list data in braces */
{ print_char('{');
  if ((p < hi_mem_min) ∨ (p > mem_end)) print_esc("CLOBBERED.");
  else show_token_list(link(p), null, max_print_line − 10);
  print_char('}');
}
static void print_rule_dimen(scaled d)   /* prints dimension in rule node */
{ if (is_running(d)) print_char('*');
  else print_scaled(d);
}
```

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

```
static void print_glue(scaled d, int order, char *s)   /* prints a glue component */
{ print_scaled(d);
  if ((order < normal) ∨ (order > filll)) print("foul");
  else if (order > normal) { print("fil");
    while (order > fil) { print_char('l');
      decr(order);
    }
  }
  else if (s ≠ 0) print(s);
}
```

177. The next subroutine prints a whole glue specification.

```
static void print_spec(int p, char *s)   /* prints a glue specification */
{ if ((p < mem_min) ∨ (p ≥ lo_mem_max)) print_char('*');
  else { print_scaled(width(p));
    if (s ≠ 0) print(s);
    if (stretch(p) ≠ 0) { print("␣plus␣");
      print_glue(stretch(p), stretch_order(p), s);
    }
    if (shrink(p) ≠ 0) { print("␣minus␣");
      print_glue(shrink(p), shrink_order(p), s);
    }
  }
}
```

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

⟨Declare procedures needed for displaying the elements of mlists 690⟩
⟨Declare the procedure called *print_skip_param* 224⟩

179. 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(A)
        { append_char('.');
          show_node_list(A);
          flush_char;
        }    /* str_room need not be checked; see show_box below */
```

180. 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⟩ +≡
  static int depth_threshold;    /* maximum nesting depth in box displays */
  static int breadth_max;    /* maximum number of items shown at the same list level */
```

181. 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.

```
  static void show_node_list(int p)    /* prints a node list symbolically */
  { int n;    /* the number of items already printed at this level */
    double g;    /* a glue ratio, as a floating point number */
    if (cur_length > depth_threshold) { if (p > null) print("␣[]");
          /* indicate that there's been some truncation */
      return;
    }
    n = 0;
    while (p > mem_min) { print_ln();
      print_current_string();    /* display the nesting history */
      if (p > mem_end)    /* pointer out of range */
      { print("Bad␣link,␣display␣aborted.");
        return;
      }
      incr(n);
      if (n > breadth_max)    /* time to stop */
      { print("etc.");
        return;
      }
      ⟨Display node p 182⟩;
      p = link(p);
    }
  }
```

182. ⟨ Display node *p* 182 ⟩ ≡
```
if (is_char_node(p)) print_font_and_char(p);
else
  switch (type(p)) {
  case hlist_node: case vlist_node: case unset_node: ⟨ Display box p 183 ⟩ break;
  case rule_node: ⟨ Display rule p 186 ⟩ break;
  case ins_node: ⟨ Display insertion p 187 ⟩ break;
  case whatsit_node: ⟨ Display the whatsit node p 1355 ⟩ break;
  case glue_node: ⟨ Display glue p 188 ⟩ break;
  case kern_node: ⟨ Display kern p 190 ⟩ break;
  case math_node: ⟨ Display math node p 191 ⟩ break;
  case ligature_node: ⟨ Display ligature p 192 ⟩ break;
  case penalty_node: ⟨ Display penalty p 193 ⟩ break;
  case disc_node: ⟨ Display discretionary p 194 ⟩ break;
  case mark_node: ⟨ Display mark p 195 ⟩ break;
  case adjust_node: ⟨ Display adjustment p 196 ⟩ break;
  ⟨ Cases of show_node_list that arise in mlists only 689 ⟩
  default: print("Unknown␣node␣type!");
  }
```
This code is used in section 181.

183. ⟨ Display box *p* 183 ⟩ ≡
```
{ if (type(p) ≡ hlist_node) print_esc("h");
  else if (type(p) ≡ vlist_node) 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) ⟨ Display special fields of the unset node p 184 ⟩
  else { ⟨ Display the value of glue_set(p) 185 ⟩;
    if (shift_amount(p) ≠ 0) { print(",␣shifted␣");
      print_scaled(shift_amount(p));
    }
  }
  node_list_display(list_ptr(p));     /* recursive call */
}
```
This code is used in section 182.

184. ⟨ Display special fields of the unset node *p* 184 ⟩ ≡

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

This code is used in section 183.

185. The code will have to change in this place if **glue_ratio** is a structured type instead of an ordinary **double**. 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 **double** 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*) 185 ⟩ ≡

```
  g = unfix(glue_set(p));
  if ((g ≠ float_constant(0)) ∧ (glue_sign(p) ≠ normal)) { print(",␣glue␣set␣");
    if (glue_sign(p) ≡ shrinking) print("-␣");
    if (abs(mem[p + glue_offset].i) < °4000000) print("?.?");
    else if (abs(g) > float_constant(20000)) { if (g > float_constant(0)) print_char('>');
      else print("<␣-");
      print_glue(20000 ∗ unity, glue_order(p), 0);
    }
    else print_glue(round(unity ∗ g), glue_order(p), 0);
  }
```

This code is used in section 183.

186. ⟨ Display rule *p* 186 ⟩ ≡

```
{ print_esc("rule(");
  print_rule_dimen(height(p));
  print_char('+');
  print_rule_dimen(depth(p));
  print(")x");
  print_rule_dimen(width(p));
}
```

This code is used in section 182.

187. ⟨Display insertion *p* 187⟩ ≡

```
{ 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 */
}
```

This code is used in section 182.

188. ⟨Display glue *p* 188⟩ ≡

```
if (subtype(p) ≥ a_leaders) ⟨Display leaders p 189⟩
else { print_esc("glue");
  if (subtype(p) ≠ normal) { print_char('(');
    if (subtype(p) < cond_math_glue) print_skip_param(subtype(p) − 1);
    else if (subtype(p) ≡ cond_math_glue) print_esc("nonscript");
    else print_esc("mskip");
    print_char(')');
  }
  if (subtype(p) ≠ cond_math_glue) { print_char('␣');
    if (subtype(p) < cond_math_glue) print_spec(glue_ptr(p), 0);
    else print_spec(glue_ptr(p), "mu");
  }
}
```

This code is used in section 182.

189. ⟨Display leaders *p* 189⟩ ≡

```
{ print_esc("");
  if (subtype(p) ≡ c_leaders) print_char('c');
  else if (subtype(p) ≡ x_leaders) print_char('x');
  print("leaders␣");
  print_spec(glue_ptr(p), 0);
  node_list_display(leader_ptr(p));      /* recursive call */
}
```

This code is used in section 188.

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

⟨ Display kern *p* 190 ⟩ ≡
```
if (subtype(p) ≠ mu_glue) { print_esc("kern");
    if (subtype(p) ≠ normal) print_char('␣');
    print_scaled(width(p));
    if (subtype(p) ≡ acc_kern) print("␣(for␣accent)");
}
else { print_esc("mkern");
    print_scaled(width(p));
    print("mu");
}
```
This code is used in section 182.

191. ⟨ Display math node *p* 191 ⟩ ≡
```
{ print_esc("math");
    if (subtype(p) ≡ before) print("on");
    else print("off");
    if (width(p) ≠ 0) { print(",␣surrounded␣");
        print_scaled(width(p));
    }
}
```
This code is used in section 182.

192. ⟨ Display ligature *p* 192 ⟩ ≡
```
{ print_font_and_char(lig_char(p));
    print("␣(ligature␣");
    if (subtype(p) > 1) print_char('|');
    font_in_short_display = font(lig_char(p));
    short_display(lig_ptr(p));
    if (odd(subtype(p))) print_char('|');
    print_char(')');
}
```
This code is used in section 182.

193. ⟨ Display penalty *p* 193 ⟩ ≡
```
{ print_esc("penalty␣");
    print_int(penalty(p));
}
```
This code is used in section 182.

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

⟨ Display discretionary *p* 194 ⟩ ≡

```
{ print_esc("discretionary");
  if (replace_count(p) > 0) { print("␣replacing␣");
    print_int(replace_count(p));
  }
  node_list_display(pre_break(p));    /* recursive call */
  append_char('|');
  show_node_list(post_break(p));
  flush_char;    /* recursive call */
}
```

This code is used in section 182.

195. ⟨ Display mark *p* 195 ⟩ ≡

```
{ print_esc("mark");
  if (mark_class(p) ≠ 0) { print_char('s');
    print_int(mark_class(p));
  }
  print_mark(mark_ptr(p));
}
```

This code is used in section 182.

196. ⟨ Display adjustment *p* 196 ⟩ ≡

```
{ print_esc("vadjust");
  node_list_display(adjust_ptr(p));    /* recursive call */
}
```

This code is used in section 182.

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

```
static void show_box(pointer p)
{ ⟨ Assign the values depth_threshold := show_box_depth and breadth_max := show_box_breadth 235 ⟩;
  if (breadth_max ≤ 0) breadth_max = 5;
  if (pool_ptr + depth_threshold ≥ pool_size) 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();
}
```

198. 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.

199. 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(A) info(A)  /* reference count preceding a token list */
  static void delete_token_ref(pointer p)
      /* p points to the reference count of a token list that is losing one reference */
  { if (token_ref_count(p) ≡ null) flush_list(p);
    else decr(token_ref_count(p));
  }
```

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

```
#define fast_delete_glue_ref(A)
        { if (glue_ref_count(A) ≡ null) free_node(A, glue_spec_size);
          else decr(glue_ref_count(A));
        }
  static void delete_glue_ref(pointer p)  /* p points to a glue specification */
  fast_delete_glue_ref(p)
```

201. 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.

```
static void flush_node_list(pointer p)   /* erase list of nodes starting at p */
{       /* go here when node p has been freed */
  pointer q;      /* successor to node p */
  while (p ≠ null) { q = link(p);
    if (is_char_node(p)) free_avail(p)
    else { switch (type(p)) {
      case hlist_node: case vlist_node: case unset_node:
        { flush_node_list(list_ptr(p));
          free_node(p, box_node_size);
          goto done;
        }
      case rule_node:
        { free_node(p, rule_node_size);
          goto done;
        }
      case ins_node:
        { flush_node_list(ins_ptr(p));
          delete_glue_ref(split_top_ptr(p));
          free_node(p, ins_node_size);
          goto done;
        }
      case whatsit_node: ⟨Wipe out the whatsit node p and goto done 1357⟩
      case glue_node:
        { fast_delete_glue_ref(glue_ptr(p));
          if (leader_ptr(p) ≠ null) flush_node_list(leader_ptr(p));
        } break;
      case kern_node: case math_node: case penalty_node: do_nothing; break;
      case ligature_node: flush_node_list(lig_ptr(p)); break;
      case mark_node: delete_token_ref(mark_ptr(p)); break;
      case disc_node:
        { flush_node_list(pre_break(p));
          flush_node_list(post_break(p));
        } break;
      case adjust_node: flush_node_list(adjust_ptr(p)); break;
      ⟨Cases of flush_node_list that arise in mlists only 697⟩
      default: confusion("flushing");
      }
      free_node(p, small_node_size);
    done: ;
    }
    p = q;
  }
}
```

202. 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(A) incr(token_ref_count(A))  /* new reference to a token list */
#define add_glue_ref(A) incr(glue_ref_count(A))  /* new reference to a glue spec */
```

203. 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.

```
static pointer copy_node_list(pointer p)
    /* makes a duplicate of the node list that starts at p and returns a pointer to the new list */
{ pointer h;  /* temporary head of copied list */
  pointer q;  /* previous position in new list */
  pointer r;  /* current node being fabricated for new list */
  int words;  /* number of words remaining to be copied */
  h = get_avail();
  q = h;
  while (p ≠ null) { ⟨Make a copy of node p in node r 204⟩;
    link(q) = r;
    q = r;
    p = link(p);
  }
  link(q) = null;
  q = link(h);
  free_avail(h);
  return q;
}
```

204. ⟨Make a copy of node p in node r 204⟩ ≡

```
words = 1;  /* this setting occurs in more branches than any other */
if (is_char_node(p)) r = get_avail();
else ⟨Case statement to copy different types and set words to the number of initial words not yet
      copied 205⟩;
while (words > 0) { decr(words);
  mem[r + words] = mem[p + words];
}
```

This code is used in section 203.

205. ⟨ Case statement to copy different types and set *words* to the number of initial words not yet copied 205 ⟩ ≡

switch (*type*(*p*)) {
case *hlist_node*: **case** *vlist_node*: **case** *unset_node*:
  { $r = get\_node(box\_node\_size)$;
    $mem[r+6] = mem[p+6]$;
    $mem[r+5] = mem[p+5]$;   /* copy the last two words */
    $list\_ptr(r) = copy\_node\_list(list\_ptr(p))$;   /* this affects $mem[r+5]$ */
    $words = 5$;
  } **break**;
case *rule_node*:
  { $r = get\_node(rule\_node\_size)$;
    $words = rule\_node\_size$;
  } **break**;
case *ins_node*:
  { $r = get\_node(ins\_node\_size)$;
    $mem[r+4] = mem[p+4]$;
    $add\_glue\_ref(split\_top\_ptr(p))$;
    $ins\_ptr(r) = copy\_node\_list(ins\_ptr(p))$;   /* this affects $mem[r+4]$ */
    $words = ins\_node\_size - 1$;
  } **break**;
case *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 1356 ⟩ **break**;
case *glue_node*:
  { $r = get\_node(small\_node\_size)$;
    $add\_glue\_ref(glue\_ptr(p))$;
    $glue\_ptr(r) = glue\_ptr(p)$;
    $leader\_ptr(r) = copy\_node\_list(leader\_ptr(p))$;
  } **break**;
case *kern_node*: **case** *math_node*: **case** *penalty_node*:
  { $r = get\_node(small\_node\_size)$;
    $words = small\_node\_size$;
  } **break**;
case *ligature_node*:
  { $r = get\_node(small\_node\_size)$;
    $mem[lig\_char(r)] = mem[lig\_char(p)]$;   /* copy *font* and *character* */
    $lig\_ptr(r) = copy\_node\_list(lig\_ptr(p))$;
  } **break**;
case *disc_node*:
  { $r = get\_node(small\_node\_size)$;
    $pre\_break(r) = copy\_node\_list(pre\_break(p))$;
    $post\_break(r) = copy\_node\_list(post\_break(p))$;
  } **break**;
case *mark_node*:
  { $r = get\_node(small\_node\_size)$;
    $add\_token\_ref(mark\_ptr(p))$;
    $words = small\_node\_size$;
  } **break**;
case *adjust_node*:
  { $r = get\_node(small\_node\_size)$;
    $adjust\_ptr(r) = copy\_node\_list(adjust\_ptr(p))$;
  } **break**;   /* $words \equiv 1 \equiv small\_node\_size - 1$ */

default: *confusion*("copying");
}

This code is used in section 204.

206. 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 ∗/

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

#define *char_num* 16 /∗ character specified numerically (`\char`) ∗/
#define *math_char_num* 17 /∗ explicit math code (`\mathchar`) ∗/
#define *mark* 18 /∗ mark definition (`\mark`) ∗/
#define *xray* 19 /∗ peek inside of TEX (`\show`, `\showbox`, etc.) ∗/
#define *make_box* 20 /∗ make a box (`\box`, `\copy`, `\hbox`, etc.) ∗/
#define *hmove* 21 /∗ horizontal motion (`\moveleft`, `\moveright`) ∗/
#define *vmove* 22 /∗ vertical motion (`\raise`, `\lower`) ∗/
#define *un_hbox* 23 /∗ unglue a box (`\unhbox`, `\unhcopy`) ∗/
#define *un_vbox* 24 /∗ unglue a box (`\unvbox`, `\unvcopy`) ∗/
/∗ (or `\pagediscards`, `\splitdiscards`) ∗/
#define *remove_item* 25 /∗ nullify last item (`\unpenalty`, `\unkern`, `\unskip`) ∗/
#define *hskip* 26 /∗ horizontal glue (`\hskip`, `\hfil`, etc.) ∗/
#define *vskip* 27 /∗ vertical glue (`\vskip`, `\vfil`, etc.) ∗/
#define *mskip* 28 /∗ math glue (`\mskip`) ∗/
#define *kern* 29 /∗ fixed space (`\kern`) ∗/
#define *mkern* 30 /∗ math kern (`\mkern`) ∗/
#define *leader_ship* 31 /∗ use a box (`\shipout`, `\leaders`, etc.) ∗/
#define *halign* 32 /∗ horizontal table alignment (`\halign`) ∗/
#define *valign* 33 /∗ vertical table alignment (`\valign`) ∗/
#define *no_align* 34 /∗ temporary escape from alignment (`\noalign`) ∗/
#define *vrule* 35 /∗ vertical rule (`\vrule`) ∗/
#define *hrule* 36 /∗ horizontal rule (`\hrule`) ∗/
#define *insert* 37 /∗ vlist inserted in box (`\insert`) ∗/
#define *vadjust* 38 /∗ vlist inserted in enclosing paragraph (`\vadjust`) ∗/
#define *ignore_spaces* 39 /∗ gobble *spacer* tokens (`\ignorespaces`) ∗/
#define *after_assignment* 40 /∗ save till assignment is done (`\afterassignment`) ∗/
#define *after_group* 41 /∗ save till group is done (`\aftergroup`) ∗/
#define *break_penalty* 42 /∗ additional badness (`\penalty`) ∗/
#define *start_par* 43 /∗ begin paragraph (`\indent`, `\noindent`) ∗/
#define *ital_corr* 44 /∗ italic correction (`\/`) ∗/
#define *accent* 45 /∗ attach accent in text (`\accent`) ∗/
#define *math_accent* 46 /∗ attach accent in math (`\mathaccent`) ∗/
#define *discretionary* 47 /∗ discretionary texts (`\-`, `\discretionary`) ∗/
#define *eq_no* 48 /∗ equation number (`\eqno`, `\leqno`) ∗/
#define *left_right* 49 /∗ variable delimiter (`\left`, `\right`) ∗/ /∗ (or `\middle`) ∗/
#define *math_comp* 50 /∗ component of formula (`\mathbin`, etc.) ∗/
#define *limit_switch* 51 /∗ diddle limit conventions (`\displaylimits`, etc.) ∗/
#define *above* 52 /∗ generalized fraction (`\above`, `\atop`, etc.) ∗/
#define *math_style* 53 /∗ style specification (`\displaystyle`, etc.) ∗/
#define *math_choice* 54 /∗ choice specification (`\mathchoice`) ∗/
#define *non_script* 55 /∗ conditional math glue (`\nonscript`) ∗/
#define *vcenter* 56 /∗ vertically center a vbox (`\vcenter`) ∗/
#define *case_shift* 57 /∗ force specific case (`\lowercase`, `\uppercase`) ∗/
#define *message* 58 /∗ send to user (`\message`, `\errmessage`) ∗/
#define *extension* 59 /∗ extensions to TEX (`\write`, `\special`, etc.) ∗/
#define *in_stream* 60 /∗ files for reading (`\openin`, `\closein`) ∗/
#define *begin_group* 61 /∗ begin local grouping (`\begingroup`) ∗/
#define *end_group* 62 /∗ end local grouping (`\endgroup`) ∗/
#define *omit* 63 /∗ omit alignment template (`\omit`) ∗/
#define *ex_space* 64 /∗ explicit space (`\␣`) ∗/

#define *no_boundary* 65 /* suppress boundary ligatures (`\noboundary`) */
#define *radical* 66 /* square root and similar signs (`\radical`) */
#define *end_cs_name* 67 /* end control sequence (`\endcsname`) */
#define *min_internal* 68 /* the smallest code that can follow `\the` */
#define *char_given* 68 /* character code defined by `\chardef` */
#define *math_given* 69 /* math code defined by `\mathchardef` */
#define *last_item* 70 /* most recent item (`\lastpenalty`, `\lastkern`, `\lastskip`) */
#define *max_non_prefixed_command* 70 /* largest command code that can't be `\global` */

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

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

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

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

210. 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 */

  static void print_mode(int m)   /* prints the mode represented by m */
  { if (m > 0)
      switch (m/(max_command + 1)) {
      case 0: print("vertical"); break;
      case 1: print("horizontal"); break;
      case 2: print("display␣math");
      }
    else if (m ≡ 0) print("no");
    else
      switch ((−m)/(max_command + 1)) {
      case 0: print("internal␣vertical"); break;
      case 1: print("restricted␣horizontal"); break;
      case 2: print("math");
      }
    print("␣mode");
  }
```

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

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

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

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

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

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

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

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

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

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

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

\#**define** *ignore_depth* −65536000 /∗ *prev_depth* value that is ignored ∗/

⟨ Types in the outer block 18 ⟩ +≡

```
typedef struct {
  int16_t mode_field; pointer head_field, tail_field;
  pointer eTeX_aux_field;
  int pg_field, ml_field; memory_word aux_field;
} list_state_record;
```

212. **#define** *mode* *cur_list.mode_field* /* current mode */
#define *head* *cur_list.head_field* /* header node of current list */
#define *tail* *cur_list.tail_field* /* final node on current list */
#define *eTeX_aux* *cur_list.eTeX_aux_field* /* auxiliary data for ε-TEX */
#define *LR_save* *eTeX_aux* /* LR stack when a paragraph is interrupted */
#define *LR_box* *eTeX_aux* /* prototype box for display */
#define *delim_ptr* *eTeX_aux* /* most recent left or right noad of a math left group */
#define *prev_graf* *cur_list.pg_field* /* number of paragraph lines accumulated */
#define *aux* *cur_list.aux_field* /* auxiliary data about the current list */
#define *prev_depth* *aux.sc* /* the name of *aux* in vertical mode */
#define *space_factor* *aux.hh.lh* /* part of *aux* in horizontal mode */
#define *clang* *aux.hh.rh* /* the other part of *aux* in horizontal mode */
#define *incompleat_noad* *aux.i* /* 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 ⟩ +≡
  static list_state_record *nest*[*nest_size* + 1];
  static int *nest_ptr*; /* first unused location of *nest* */
  static int *max_nest_stack*; /* maximum of *nest_ptr* when pushing */
  static list_state_record *cur_list*; /* the "top" semantic state */
  static int *shown_mode*; /* most recent mode shown by `\tracingcommands` */

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

#define *tail_append*(*A*)
  { *link*(*tail*) = *A*;
    tail = *link*(*tail*);
  }

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

⟨ Set initial values of key variables 21 ⟩ +≡
  nest_ptr = 0;
  max_nest_stack = 0;
  mode = *vmode*;
  head = *contrib_head*;
  tail = *contrib_head*;
  eTeX_aux = *null*;
  prev_depth = *ignore_depth*;
  mode_line = 0;
  prev_graf = 0;
  shown_mode = 0;
  ⟨ Start a new current page 990 ⟩;

215. 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*.

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

216. 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.

```
static void pop_nest(void)    /* leave a semantic level, re-enter the old */
{ free_avail(head);
  decr(nest_ptr);
  cur_list = nest[nest_ptr];
}
```

217. Here is a procedure that displays what TEX is working on, at all levels.

```
static void print_totals(void);

static void show_activities(void)
{ int p;  /* index into nest */
  int m;  /* mode */
  memory_word a;  /* auxiliary */
  pointer q, r;  /* for showing the current page */
  int t;  /* ditto */

  nest[nest_ptr] = cur_list;  /* put the top level into the array */
  print_nl("");
  print_ln();
  for (p = nest_ptr; p ≥ 0; p--) { 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)
      if (nest[p].pg_field ≠ °40600000) { print("␣(language");
        print_int(nest[p].pg_field % °200000);
        print(":hyphenmin");
        print_int(nest[p].pg_field/°20000000);
        print_char(',');
        print_int((nest[p].pg_field/°200000) % °100);
        print_char(')');
      }
    if (nest[p].ml_field < 0) print("␣(\\output␣routine)");
    if (p ≡ 0) { ⟨Show the status of the current page 985⟩;
      if (link(contrib_head) ≠ null) print_nl("###␣recent␣contributions:");
    }
    show_box(link(nest[p].head_field));
    ⟨Show the auxiliary field, a 218⟩;
  }
}
```

218. ⟨ Show the auxiliary field, *a* 218 ⟩ ≡

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

This code is used in section 217.

219. 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.

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

221. 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_primitive (frozen_control_sequence + 10) /* permanent '\primitive:' */
#define frozen_null_font (frozen_control_sequence + 11) /* 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) 163 ⟩ +≡

```
  eq_type(undefined_control_sequence) = undefined_cs;
  equiv(undefined_control_sequence) = null;
  eq_level(undefined_control_sequence) = level_zero;
  for (k = active_base; k ≤ undefined_control_sequence - 1; k++)
    eqtb[k] = eqtb[undefined_control_sequence];
```

222. 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 2 222 ⟩ ≡

```
  { sprint_cs(n);
    print_char('=');
    print_cmd_chr(eq_type(n), equiv(n));
    if (eq_type(n) ≥ call) { print_char(':');
      show_token_list(link(equiv(n)), null, 32);
    }
  }
```

This code is used in section 251.

223. 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*(*A*) *equiv*(*skip_base* + *A*) /* *mem* location of glue specification */
#define *mu_skip*(*A*) *equiv*(*mu_skip_base* + *A*) /* *mem* location of math glue spec */
#define *glue_par*(*A*) *equiv*(*glue_base* + *A*) /* *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* 223⟩ ≡
glue_par(*n*)

This code is used in sections 151 and 153.

224. 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* 224 ⟩ ≡

```
static void print_skip_param(int n)
{ switch (n) {
    case line_skip_code: print_esc("lineskip"); break;
    case baseline_skip_code: print_esc("baselineskip"); break;
    case par_skip_code: print_esc("parskip"); break;
    case above_display_skip_code: print_esc("abovedisplayskip"); break;
    case below_display_skip_code: print_esc("belowdisplayskip"); break;
    case above_display_short_skip_code: print_esc("abovedisplayshortskip"); break;
    case below_display_short_skip_code: print_esc("belowdisplayshortskip"); break;
    case left_skip_code: print_esc("leftskip"); break;
    case right_skip_code: print_esc("rightskip"); break;
    case top_skip_code: print_esc("topskip"); break;
    case split_top_skip_code: print_esc("splittopskip"); break;
    case tab_skip_code: print_esc("tabskip"); break;
    case space_skip_code: print_esc("spaceskip"); break;
    case xspace_skip_code: print_esc("xspaceskip"); break;
    case par_fill_skip_code: print_esc("parfillskip"); break;
    case thin_mu_skip_code: print_esc("thinmuskip"); break;
    case med_mu_skip_code: print_esc("medmuskip"); break;
    case thick_mu_skip_code: print_esc("thickmuskip"); break;
    default: print("[unknown␣glue␣parameter!]");
    }
}
```

This code is used in section 178.

225. 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 225 ⟩ ≡
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 229, 237, 247, 264, 333, 375, 383, 410, 415, 467, 486, 490, 552, 779, 982, 1051, 1057, 1070, 1087, 1106, 1113, 1140, 1155, 1168, 1177, 1187, 1207, 1218, 1221, 1229, 1249, 1253, 1261, 1271, 1276, 1285, 1290, 1343, and 1737.

This code is used in section 1335.

226. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ ≡
case *assign_glue*: **case** *assign_mu_glue*:
  if (*chr_code* < *skip_base*) *print_skip_param*(*chr_code* − *glue_base*);
  else if (*chr_code* < *mu_skip_base*) { *print_esc*("skip");
    print_int(*chr_code* − *skip_base*);
  }
  else { *print_esc*("muskip");
    print_int(*chr_code* − *mu_skip_base*);
  } **break**;

See also sections 230, 238, 248, 265, 334, 376, 384, 411, 416, 468, 487, 491, 780, 983, 1052, 1058, 1071, 1088, 1107, 1114, 1142, 1156, 1169, 1178, 1188, 1208, 1219, 1222, 1230, 1250, 1254, 1260, 1262, 1272, 1277, 1286, 1291, 1294, and 1345.

This code is used in section 297.

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

⟨ Initialize table entries (done by INITEX only) 163 ⟩ +≡
equiv(*glue_base*) = *zero_glue*;
eq_level(*glue_base*) = *level_one*;
eq_type(*glue_base*) = *glue_ref*;
for (k = *glue_base* + 1; $k \le$ *local_base* − 1; k++) *eqtb*[k] = *eqtb*[*glue_base*];
glue_ref_count(*zero_glue*) = *glue_ref_count*(*zero_glue*) + *local_base* − *glue_base*;

228. ⟨Show equivalent n, in region 3 228⟩ ≡

if $(n < skip\_base)$ { $print\_skip\_param(n - glue\_base)$;
  $print\_char$('=');
  if $(n < glue\_base + thin\_mu\_skip\_code)$ $print\_spec(equiv(n),$ "pt");
  else $print\_spec(equiv(n),$ "mu");
}
else if $(n < mu\_skip\_base)$ { $print\_esc$("skip");
  $print\_int(n - skip\_base)$;
  $print\_char$('=');
  $print\_spec(equiv(n),$ "pt");
}
else { $print\_esc$("muskip");
  $print\_int(n - mu\_skip\_base)$;
  $print\_char$('=');
  $print\_spec(equiv(n),$ "mu");
}

This code is used in section 251.

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

#**define** *par_shape_loc* *local_base* /* specifies paragraph shape */
#**define** *output_routine_loc* (*local_base* + 1) /* points to token list for `\output` */
#**define** *every_par_loc* (*local_base* + 2) /* points to token list for `\everypar` */
#**define** *every_math_loc* (*local_base* + 3) /* points to token list for `\everymath` */
#**define** *every_display_loc* (*local_base* + 4) /* points to token list for `\everydisplay` */
#**define** *every_hbox_loc* (*local_base* + 5) /* points to token list for `\everyhbox` */
#**define** *every_vbox_loc* (*local_base* + 6) /* points to token list for `\everyvbox` */
#**define** *every_job_loc* (*local_base* + 7) /* points to token list for `\everyjob` */
#**define** *every_cr_loc* (*local_base* + 8) /* points to token list for `\everycr` */
#**define** *err_help_loc* (*local_base* + 9) /* points to token list for `\errhelp` */
#**define** *tex_toks* (*local_base* + 10) /* end of TEX's token list parameters */

#**define** *etex_toks_base* *tex_toks* /* base for ε-TEX's token list parameters */
#**define** *every_eof_loc* *etex_toks_base* /* points to token list for `\everyeof` */
#**define** *etex_toks* (*etex_toks_base* + 1) /* end of ε-TEX's token list parameters */

#**define** *toks_base* *etex_toks* /* table of 256 token list registers */

#**define** *etex_pen_base* (*toks_base* + 256) /* start of table of ε-TEX's penalties */
#**define** *inter_line_penalties_loc* *etex_pen_base* /* additional penalties between lines */
#**define** *club_penalties_loc* (*etex_pen_base* + 1) /* penalties for creating club lines */
#**define** *widow_penalties_loc* (*etex_pen_base* + 2) /* penalties for creating widow lines */
#**define** *display_widow_penalties_loc* (*etex_pen_base* + 3) /* ditto, just before a display */
#**define** *etex_pens* (*etex_pen_base* + 4) /* end of table of ε-TEX's penalties */

#**define** *box_base* *etex_pens* /* table of 256 box registers */
#**define** *cur_font_loc* (*box_base* + 256) /* internal font number outside math mode */
#**define** *math_font_base* (*cur_font_loc* + 1) /* table of 48 math font numbers */
#**define** *cat_code_base* (*math_font_base* + 48) /* table of 256 command codes (the "catcodes") */
#**define** *lc_code_base* (*cat_code_base* + 256) /* table of 256 lowercase mappings */
#**define** *uc_code_base* (*lc_code_base* + 256) /* table of 256 uppercase mappings */
#**define** *sf_code_base* (*uc_code_base* + 256) /* table of 256 spacefactor mappings */
#**define** *math_code_base* (*sf_code_base* + 256) /* table of 256 math mode mappings */
#**define** *int_base* (*math_code_base* + 256) /* beginning of region 5 */

#**define** *par_shape_ptr* *equiv*(*par_shape_loc*)
#**define** *output_routine* *equiv*(*output_routine_loc*)
#**define** *every_par* *equiv*(*every_par_loc*)
#**define** *every_math* *equiv*(*every_math_loc*)
#**define** *every_display* *equiv*(*every_display_loc*)
#**define** *every_hbox* *equiv*(*every_hbox_loc*)
#**define** *every_vbox* *equiv*(*every_vbox_loc*)
#**define** *every_job* *equiv*(*every_job_loc*)
#**define** *every_cr* *equiv*(*every_cr_loc*)
#**define** *err_help* *equiv*(*err_help_loc*)
#**define** *toks*(X) *equiv*(*toks_base* + X)
#**define** *box*(A) *equiv*(*box_base* + A)
#**define** *cur_font* *equiv*(*cur_font_loc*)
#**define** *fam_fnt*(A) *equiv*(*math_font_base* + A)
#**define** *cat_code*(A) *equiv*(*cat_code_base* + A)
#**define** *lc_code*(A) *equiv*(*lc_code_base* + A)
#**define** *uc_code*(A) *equiv*(*uc_code_base* + A)

```
#define sf_code(A) equiv(sf_code_base + A)
#define math_code(A) equiv(math_code_base + A)
        /* Note: math_code(c) is the true math code plus min_halfword */
⟨Put each of TEX's primitives into the hash table 225⟩ +≡
  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);
```

230. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 226⟩ +≡

```
  case assign_toks:
  if (chr_code ≥ toks_base) { print_esc("toks");
    print_int(chr_code − toks_base);
  }
  else switch (chr_code) {
case output_routine_loc: print_esc("output"); break;
case every_par_loc: print_esc("everypar"); break;
case every_math_loc: print_esc("everymath"); break;
case every_display_loc: print_esc("everydisplay"); break;
case every_hbox_loc: print_esc("everyhbox"); break;
case every_vbox_loc: print_esc("everyvbox"); break;
case every_job_loc: print_esc("everyjob"); break;
case every_cr_loc: print_esc("everycr"); break;
  ⟨Cases of assign_toks for print_cmd_chr 1388⟩
default: print_esc("errhelp");
  } break;
```

231. 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) 163⟩ +≡
  par_shape_ptr = null;
  eq_type(par_shape_loc) = shape_ref;
  eq_level(par_shape_loc) = level_one;
  for (k = etex_pen_base; k ≤ etex_pens − 1; k++) eqtb[k] = eqtb[par_shape_loc];
  for (k = output_routine_loc; k ≤ toks_base + 255; k++) 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; k ≤ box_base + 255; k++) 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; k ≤ math_font_base + 47; k++) 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; k ≤ int_base − 1; k++) eqtb[k] = eqtb[cat_code_base];
  for (k = 0; k ≤ 255; k++) { cat_code(k) = other_char;
    math_code(k) = hi(k);
    sf_code(k) = 1000;
  }
  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'; k ≤ '9'; k++) math_code(k) = hi(k + var_code);
  for (k = 'A'; k ≤ 'Z'; k++) { 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;
  }
```

232. ⟨Show equivalent n, in region 4 232⟩ ≡

```
if ((n ≡ par_shape_loc) ∨ ((n ≥ etex_pen_base) ∧ (n < etex_pens))) { print_cmd_chr(set_shape, n);
  print_char('=');
  if (equiv(n) ≡ null) print_char('0');
  else if (n > par_shape_loc) { print_int(penalty(equiv(n)));
    print_char('␣');
    print_int(penalty(equiv(n) + 1));
    if (penalty(equiv(n)) > 1) print_esc("ETC.");
  }
  else print_int(info(par_shape_ptr));
}
else if (n < toks_base) { print_cmd_chr(assign_toks, n);
  print_char('=');
  if (equiv(n) ≠ null) show_token_list(link(equiv(n)), null, 32);
}
else if (n < box_base) { print_esc("toks");
  print_int(n − toks_base);
  print_char('=');
  if (equiv(n) ≠ null) show_token_list(link(equiv(n)), null, 32);
}
else if (n < cur_font_loc) { print_esc("box");
  print_int(n − box_base);
  print_char('=');
  if (equiv(n) ≡ null) print("void");
  else { depth_threshold = 0;
    breadth_max = 1;
    show_node_list(equiv(n));
  }
}
else if (n < cat_code_base) ⟨Show the font identifier in eqtb[n] 233⟩
else ⟨Show the halfword code in eqtb[n] 234⟩
```

This code is used in section 251.

233. ⟨Show the font identifier in $eqtb[n]$ 233⟩ ≡

```
{ if (n ≡ cur_font_loc) print("current␣font");
  else if (n < math_font_base + 16) { print_esc("textfont");
    print_int(n − math_font_base);
  }
  else if (n < math_font_base + 32) { print_esc("scriptfont");
    print_int(n − math_font_base − 16);
  }
  else { print_esc("scriptscriptfont");
    print_int(n − math_font_base − 32);
  }
  print_char('=');
  printn_esc(hash[font_id_base + equiv(n)].rh);    /* that's font_id_text(equiv(n)) */
}
```

This code is used in section 232.

234. ⟨Show the halfword code in *eqtb*[*n*] 234⟩ ≡

if ($n <$ *math_code_base*) { **if** ($n <$ *lc_code_base*) { *print_esc*("catcode");
    print_int($n -$ *cat_code_base*);
  }
  else if ($n <$ *uc_code_base*) { *print_esc*("lccode");
    print_int($n -$ *lc_code_base*);
  }
  else if ($n <$ *sf_code_base*) { *print_esc*("uccode");
    print_int($n -$ *uc_code_base*);
  }
  else { *print_esc*("sfcode");
    print_int($n -$ *sf_code_base*);
  }
  print_char('=');
  print_int(*equiv*(*n*));
}
else { *print_esc*("mathcode");
  print_int($n -$ *math_code_base*);
  print_char('=');
  print_int(*ho*(*equiv*(*n*)));
}

This code is used in section 232.

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

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

```
#define default_skew_char_code 47 /* value of \skewchar when a font is loaded */
#define end_line_char_code 48 /* character placed at the right end of the buffer */
#define new_line_char_code 49 /* character that prints as print_ln */
#define language_code 50 /* current hyphenation table */
#define left_hyphen_min_code 51 /* minimum left hyphenation fragment size */
#define right_hyphen_min_code 52 /* minimum right hyphenation fragment size */
#define holding_inserts_code 53 /* do not remove insertion nodes from \box255 */
#define error_context_lines_code 54 /* maximum intermediate line pairs shown */
#define tex_int_pars 55 /* total number of TeX's integer parameters */

#define etex_int_base tex_int_pars /* base for ε-TeX's integer parameters */
#define tracing_assigns_code etex_int_base /* show assignments */
#define tracing_groups_code (etex_int_base + 1) /* show save/restore groups */
#define tracing_ifs_code (etex_int_base + 2) /* show conditionals */
#define tracing_scan_tokens_code (etex_int_base + 3) /* show pseudo file open and close */
#define tracing_nesting_code (etex_int_base + 4) /* show incomplete groups and ifs within files */
#define saving_vdiscards_code (etex_int_base + 5) /* save items discarded from vlists */
#define saving_hyph_codes_code (etex_int_base + 6) /* save hyphenation codes for languages */
#define expand_depth_code (etex_int_base + 7) /* maximum depth for expansion—ε-TeX */
#define eTeX_state_code (etex_int_base + 8) /* ε-TeX state variables */
#define etex_int_pars (eTeX_state_code + eTeX_states)
          /* total number of ε-TeX's integer parameters */

#define int_pars etex_int_pars /* total number of integer parameters */
#define count_base (int_base + int_pars) /* 256 user \count registers */
#define del_code_base (count_base + 256) /* 256 delimiter code mappings */
#define dimen_base (del_code_base + 256) /* beginning of region 6 */

#define del_code(A) eqtb[del_code_base + A].i
#define count(A) eqtb[count_base + A].i
#define int_par(A) eqtb[int_base + A].i /* 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* $\equiv$ *int_par*(*year_code*)
#define *show_box_breadth* *int_par*(*show_box_breadth_code*)
#define *show_box_depth* *int_par*(*show_box_depth_code*)
#define *hbadness* *int_par*(*hbadness_code*)
#define *vbadness* *int_par*(*vbadness_code*)
#define *pausing* *int_par*(*pausing_code*)
#define *tracing_online* *int_par*(*tracing_online_code*)
#define *tracing_macros* *int_par*(*tracing_macros_code*)
#define *tracing_stats* *int_par*(*tracing_stats_code*)
#define *tracing_paragraphs* *int_par*(*tracing_paragraphs_code*)
#define *tracing_pages* *int_par*(*tracing_pages_code*)
#define *tracing_output* *int_par*(*tracing_output_code*)
#define *tracing_lost_chars* *int_par*(*tracing_lost_chars_code*)
#define *tracing_commands* *int_par*(*tracing_commands_code*)
#define *tracing_restores* *int_par*(*tracing_restores_code*)
#define *uc_hyph* *int_par*(*uc_hyph_code*)
#define *output_penalty* *int_par*(*output_penalty_code*)
#define *max_dead_cycles* *int_par*(*max_dead_cycles_code*)
#define *hang_after* *int_par*(*hang_after_code*)
#define *floating_penalty* *int_par*(*floating_penalty_code*)
#define *global_defs* *int_par*(*global_defs_code*)
#define *cur_fam* *int_par*(*cur_fam_code*)
#define *escape_char* *int_par*(*escape_char_code*)
#define *default_hyphen_char* *int_par*(*default_hyphen_char_code*)
#define *default_skew_char* *int_par*(*default_skew_char_code*)
#define *end_line_char* *int_par*(*end_line_char_code*)
#define *new_line_char* *int_par*(*new_line_char_code*)
#define *language* *int_par*(*language_code*)
#define *left_hyphen_min* *int_par*(*left_hyphen_min_code*)
#define *right_hyphen_min* *int_par*(*right_hyphen_min_code*)
#define *holding_inserts* *int_par*(*holding_inserts_code*)
#define *error_context_lines* *int_par*(*error_context_lines_code*)

#define *tracing_assigns* *int_par*(*tracing_assigns_code*)
#define *tracing_groups* *int_par*(*tracing_groups_code*)
#define *tracing_ifs* *int_par*(*tracing_ifs_code*)
#define *tracing_scan_tokens* *int_par*(*tracing_scan_tokens_code*)
#define *tracing_nesting* *int_par*(*tracing_nesting_code*)
#define *saving_vdiscards* *int_par*(*saving_vdiscards_code*)
#define *saving_hyph_codes* *int_par*(*saving_hyph_codes_code*)
#define *expand_depth* *int_par*(*expand_depth_code*)

⟨ Assign the values *depth_threshold*: = *show_box_depth* and *breadth_max*: = *show_box_breadth* 235 ⟩ ≡
depth_threshold = *show_box_depth*; *breadth_max* = *show_box_breadth*

This code is used in section 197.

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

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

case *language_code*: *print_esc*("language"); **break**;
case *left_hyphen_min_code*: *print_esc*("lefthyphenmin"); **break**;
case *right_hyphen_min_code*: *print_esc*("righthyphenmin"); **break**;
case *holding_inserts_code*: *print_esc*("holdinginserts"); **break**;
case *error_context_lines_code*: *print_esc*("errorcontextlines"); **break**;
⟨ Cases for *print_param* 1389 ⟩
default: *print*("[unknown␣integer␣parameter!]");
}
}

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

⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡

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", $assign\_int$, $int\_base + error\_context\_lines\_code$);

238. ⟨ Cases of $print\_cmd\_chr$ for symbolic printing of primitives 226 ⟩ +≡
case $assign\_int$:
  if ($chr\_code < count\_base$) $print\_param(chr\_code - int\_base)$;
  else { $print\_esc$("count");
    $print\_int(chr\_code - count\_base)$;
  } **break**;

239. 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) 163 ⟩ +≡
  for ($k = int\_base$; $k \le del\_code\_base - 1$; $k{+}{+}$) $eqtb[k].i = 0$;
  $mag = 1000$;
  $tolerance = 10000$;
  $hang\_after = 1$;
  $max\_dead\_cycles = 25$;
  $escape\_char$ = '\\';
  $end\_line\_char = carriage\_return$;
  for ($k = 0$; $k \le 255$; $k{+}{+}$) $del\_code(k) = -1$;
  del_code('.') $= 0$;   /* this null delimiter is used in error recovery */

240. The following procedure, which is called just before TEX initializes its input and output, establishes the initial values of the date and time. This does include too, for system integrators, the creation date and the reference moment for the timer—PRoTE extensions. If the system supports environment variables, if FORCE_SOURCE_DATE is set to 1 and SOURCE_DATE_EPOCH is set, the date related values: year, month, day and time, including creation date, will be taken relative from the value defined by SOURCE_DATE_EPOCH. TEX Live calls *tl_now* to obtain the current time as a *tm* structure.

  static void $fix\_date\_and\_time$(**void**)
  { **struct** tm $*t = tl\_now()$;
    $time = sys\_time = t \to tm\_hour * 60 + t \to tm\_min$;   /* minutes since midnight */
    $day = sys\_day = t \to tm\_mday$;   /* day of the month */
    $month = sys\_month = t \to tm\_mon + 1$;   /* month of the year */
    $year = sys\_year = t \to tm\_year + 1900$;   /* Anno Domini */
  }

241. ⟨Show equivalent n, in region 5 241⟩ ≡

```
{ if (n < count_base) print_param(n − int_base);
  else if (n < del_code_base) { print_esc("count");
    print_int(n − count_base);
  }
  else { print_esc("delcode");
    print_int(n − del_code_base);
  }
  print_char('=');
  print_int(eqtb[n].i);
}
```

This code is used in section 251.

242. ⟨Set variable c to the current escape character 242⟩ ≡

```
c = escape_char
```

This code is used in section 62.

243. ⟨Character s is the current new-line character 243⟩ ≡

```
s ≡ new_line_char
```

This code is used in sections 57 and 58.

244. 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:

```
static void begin_diagnostic(void)    /* prepare to do some tracing */
{ old_setting = selector;
  if ((tracing_online ≤ 0) ∧ (selector ≡ term_and_log)) { decr(selector);
    if (history ≡ spotless) history = warning_issued;
  }
}
static void end_diagnostic(bool blank_line)    /* restore proper conditions after tracing */
{ print_nl("");
  if (blank_line) print_ln();
  selector = old_setting;
}
```

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

⟨Global variables 13⟩ +≡

```
static int old_setting;
static int sys_time, sys_day, sys_month, sys_year;    /* date and time supplied by external system */
```

246. 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 *page_width_code* 21 /* current paper page width */
#define *page_height_code* 22 /* current paper page height */
#define *dimen_pars* 23 /* 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*(*A*) *eqtb*[*scaled_base* + *A*].*sc*
#define *dimen_par*(*A*) *eqtb*[*dimen_base* + *A*].*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*)
#define *page_height* *dimen_par*(*page_height_code*)

```
static void print_length_param(int n)
{ switch (n) {
  case par_indent_code: print_esc("parindent"); break;
  case math_surround_code: print_esc("mathsurround"); break;
  case line_skip_limit_code: print_esc("lineskiplimit"); break;
  case hsize_code: print_esc("hsize"); break;
  case vsize_code: print_esc("vsize"); break;
  case max_depth_code: print_esc("maxdepth"); break;
  case split_max_depth_code: print_esc("splitmaxdepth"); break;
  case box_max_depth_code: print_esc("boxmaxdepth"); break;
  case hfuzz_code: print_esc("hfuzz"); break;
  case vfuzz_code: print_esc("vfuzz"); break;
  case delimiter_shortfall_code: print_esc("delimitershortfall"); break;
  case null_delimiter_space_code: print_esc("nulldelimiterspace"); break;
  case script_space_code: print_esc("scriptspace"); break;
  case pre_display_size_code: print_esc("predisplaysize"); break;
  case display_width_code: print_esc("displaywidth"); break;
  case display_indent_code: print_esc("displayindent"); break;
  case overfull_rule_code: print_esc("overfullrule"); break;
  case hang_indent_code: print_esc("hangindent"); break;
  case h_offset_code: print_esc("hoffset"); break;
  case v_offset_code: print_esc("voffset"); break;
  case emergency_stretch_code: print_esc("emergencystretch"); break;
  case page_width_code: print_esc("pagewidth"); break;
  case page_height_code: print_esc("pageheight"); break;
  default: print("[unknown␣dimen␣parameter!]");
  }
}
```

247. ⟨Put each of TEX's primitives into the hash table 225⟩ +≡

```
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);
```

248. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

case *assign_dimen*:
  if (*chr_code* < *scaled_base*) *print_length_param*(*chr_code* − *dimen_base*);
  else { *print_esc*("dimen");
    print_int(*chr_code* − *scaled_base*);
  } **break**;

249. ⟨ Initialize table entries (done by INITEX only) 163 ⟩ +≡

  for (*k* = *dimen_base*; *k* ≤ *eqtb_size*; *k*++) *eqtb*[*k*].*sc* = 0;

250. ⟨ Show equivalent *n*, in region 6 250 ⟩ ≡

  { **if** (*n* < *scaled_base*) *print_length_param*(*n* − *dimen_base*);
    else { *print_esc*("dimen");
      print_int(*n* − *scaled_base*);
    }
    print_char('=');
    print_scaled(*eqtb*[*n*].*sc*);
    print("pt");
  }

This code is used in section 251.

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

⟨ Declare the procedure called *print_cmd_chr* 297 ⟩
#**ifdef** STAT
  static void *show_eqtb*(**pointer** *n*)
  { **if** (*n* < *active_base*) *print_char*('?'); /∗ this can't happen ∗/
    else if (*n* < *glue_base*) ⟨ Show equivalent *n*, in region 1 or 2 222 ⟩
    else if (*n* < *local_base*) ⟨ Show equivalent *n*, in region 3 228 ⟩
    else if (*n* < *int_base*) ⟨ Show equivalent *n*, in region 4 232 ⟩
    else if (*n* < *dimen_base*) ⟨ Show equivalent *n*, in region 5 241 ⟩
    else if (*n* ≤ *eqtb_size*) ⟨ Show equivalent *n*, in region 6 250 ⟩
    else *print_char*('?'); /∗ this can't happen either ∗/
  }
#**endif**

252. 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 ⟩ +≡
  static memory_word *eqtb0*[*eqtb_size* − *active_base* + 1], ∗**const** *eqtb* = *eqtb0* − *active_base*;
  static quarterword *xeq_level0*[*eqtb_size* − *int_base* + 1], ∗**const** *xeq_level* = *xeq_level0* − *int_base*;

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

  for (*k* = *int_base*; *k* ≤ *eqtb_size*; *k*++) *xeq_level*[*k*] = *level_one*;

254. 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 254 ⟩ ≡

```
for (q = active_base; q ≤ box_base + 255; q++) { if (equiv(q) ≡ p) { print_nl("EQUIV(");
      print_int(q);
      print_char(')');
    }
  }
```

This code is used in section 171.

255. 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 $text(p) \equiv 0$; if position p is either empty or the end of a coalesced hash list, we have $next(p) \equiv 0$. An auxiliary pointer variable called *hash_used* is maintained in such a way that all locations $p \ge hash\_used$ 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*(*A*) *hash*[*A*].*lh* /∗ link for coalesced lists ∗/
#**define** *text*(*A*) *hash*[*A*].*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*(*A*) *text*(*font_id_base* + *A*) /∗ a frozen font identifier's name ∗/

⟨ Global variables 13 ⟩ +≡
static two_halves *hash0*[*undefined_control_sequence* − *hash_base*], ∗**const** *hash* = *hash0* − *hash_base*;
/∗ the hash table ∗/
static pointer *hash_used*; /∗ allocation pointer for *hash* ∗/
static bool *no_new_control_sequence*; /∗ are new identifiers legal? ∗/
static int *cs_count*; /∗ total number of known identifiers ∗/

256. ⟨ 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; *k* ≤ *undefined_control_sequence* − 1; *k*++) *hash*[*k*] = *hash*[*hash_base*];

257. ⟨ Initialize table entries (done by `INITEX` only) 163 ⟩ +≡
hash_used = *frozen_control_sequence*; /∗ nothing is used ∗/
cs_count = 0;
eq_type(*frozen_dont_expand*) = *dont_expand*;
text(*frozen_dont_expand*) = *s_no*(`"notexpanded:"`);

258. 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.

```
static pointer id_lookup(int j, int l)   /* search the hash table */
{   /* go here if you found it */
  int h;   /* hash code */
  int d;   /* number of characters in incomplete current string */
  pointer p;   /* index in hash array */
  int k;   /* index in buffer array */
  ⟨Compute the hash code h 260⟩;
  p = h + hash_base;   /* we start searching here; note that 0 ≤ h < hash_prime */
  loop { if (text(p) > 0)
      if (length(text(p)) ≡ l)
        if (str_eq_buf(text(p), j)) goto found;
    if (next(p) ≡ 0) { if (no_new_control_sequence) p = undefined_control_sequence;
      else ⟨Insert a new control sequence after p, then make p point to it 259⟩;
      goto found;
    }
    p = next(p);
  }
found: return p;
}
```

259. ⟨Insert a new control sequence after p, then make p point to it 259⟩ ≡

```
{ if (text(p) > 0) { do {
      if (hash_is_full) overflow("hash␣size", hash_size);
      decr(hash_used);
    } while (¬(text(hash_used) ≡ 0));   /* search for an empty location in hash */
    next(p) = hash_used;
    p = hash_used;
  }
  str_room(l);
  d = cur_length;
  while (pool_ptr > str_start[str_ptr]) { decr(pool_ptr);
    str_pool[pool_ptr + l] = str_pool[pool_ptr];
  }   /* move current string up to make room for another */
  for (k = j; k ≤ j + l − 1; k++) append_char(buffer[k]);
  text(p) = make_string();
  pool_ptr = pool_ptr + d;
#ifdef STAT
  incr(cs_count);
#endif
}
```

This code is used in section 258.

260. 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 260⟩ ≡
```
h = buffer[j];
for (k = j + 1; k ≤ j + l − 1; k++) { h = h + h + buffer[k];
  while (h ≥ hash_prime) h = h − hash_prime;
}
```
This code is used in section 258.

261. 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 55⟩ +≡
```
static void print_cs(int p)    /* prints a purported control sequence */
{ if (p < hash_base)    /* single character */
    if (p ≥ single_base)
      if (p ≡ null_cs) { print_esc("csname");
        print_esc("endcsname");
        print_char('␣');
      }
      else { printn_esc(p − single_base);
        if (cat_code(p − single_base) ≡ letter) print_char('␣');
      }
    else if (p < active_base) print_esc("IMPOSSIBLE.");
    else printn(p − active_base);
  else if (p ≥ undefined_control_sequence) print_esc("IMPOSSIBLE.");
  else if ((text(p) < 0) ∨ (text(p) ≥ str_ptr)) print_esc("NONEXISTENT.");
  else { if (p ≡ frozen_primitive) print_esc("primitive");
    printn_esc(text(p));
    print_char('␣');
  }
}
```

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

⟨Basic printing procedures 55⟩ +≡
```
static void sprint_cs(pointer p)    /* prints a control sequence */
{ if (p < hash_base)
    if (p < single_base) printn(p − active_base);
    else if (p < null_cs) printn_esc(p − single_base);
    else { print_esc("csname");
      print_esc("endcsname");
    }
  else printn_esc(text(p));
}
```

263. 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.

```
#ifdef INIT
  static void primitive(char *str, quarterword c, halfword o)
  { str_number s = s_no(str);
    int k;    /* index into str_pool */
    int j;    /* index into buffer */
    small_number l;    /* length of the string */
    pointer p;    /* pointer in ROM */
    if (s < 256) cur_val = s + single_base;
    else { k = str_start[s];
      l = str_start[s + 1] - k;    /* we will move s into the (possibly non-empty) buffer */
      if (first + l > buf_size + 1) overflow("buffer size", buf_size);
      for (j = 0; j ≤ l - 1; j++) buffer[first + j] = so(str_pool[k + j]);
      cur_val = id_lookup(first, l);    /* no_new_control_sequence is false */
      flush_string;
      text(cur_val) = s;    /* we don't want to have the string twice */
    }
    eq_level(cur_val) = level_one;
    eq_type(cur_val) = c;
    equiv(cur_val) = o;
    ⟨Add primitive definition to the ROM array 1583⟩;
  }
#endif
```

264. 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 225⟩ +≡
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*) = *text*(*cur_val*);
eqtb[*frozen_end_group*] = *eqtb*[*cur_val*];
primitive("expandafter", *expand_after*, 0);
primitive("font", *def_font*, 0);
primitive("fontdimen", *assign_font_dimen*, 0);
primitive("halign", *halign*, 0);
primitive("hrule", *hrule*, 0);
primitive("ignorespaces", *ignore_spaces*, 0);
primitive("insert", *insert*, 0);
primitive("mark", *mark*, 0);
primitive("mathaccent", *math_accent*, 0);
primitive("mathchar", *math_char_num*, 0);
primitive("mathchoice", *math_choice*, 0);
primitive("multiply", *multiply*, 0);
primitive("noalign", *no_align*, 0);
primitive("noboundary", *no_boundary*, 0);
primitive("noexpand", *no_expand*, 0);
primitive("nonscript", *non_script*, 0);
primitive("omit", *omit*, 0);
primitive("parshape", *set_shape*, *par_shape_loc*);
primitive("penalty", *break_penalty*, 0);
primitive("prevgraf", *set_prev_graf*, 0);
primitive("radical", *radical*, 0);
primitive("read", *read_to_cs*, 0);
primitive("relax", *relax*, 256); /* cf. *scan_file_name* */
text(*frozen_relax*) = *text*(*cur_val*);
eqtb[*frozen_relax*] = *eqtb*[*cur_val*];
primitive("setbox", *set_box*, 0);
primitive("the", *the*, 0);
primitive("toks", *toks_register*, *mem_bot*);
primitive("vadjust", *vadjust*, 0);
primitive("valign", *valign*, 0);
primitive("vcenter", *vcenter*, 0);
primitive("vrule", *vrule*, 0);

265. 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 226 ⟩ +≡
case *accent*: *print_esc*("accent"); **break**;
case *advance*: *print_esc*("advance"); **break**;
case *after_assignment*: *print_esc*("afterassignment"); **break**;
case *after_group*: *print_esc*("aftergroup"); **break**;
case *assign_font_dimen*: *print_esc*("fontdimen"); **break**;
case *begin_group*: *print_esc*("begingroup"); **break**;
case *break_penalty*: *print_esc*("penalty"); **break**;
case *char_num*: *print_esc*("char"); **break**;
case *cs_name*: *print_esc*("csname"); **break**;
case *def_font*: *print_esc*("font"); **break**;
case *delim_num*: *print_esc*("delimiter"); **break**;
case *divide*: *print_esc*("divide"); **break**;
case *end_cs_name*: *print_esc*("endcsname"); **break**;
case *end_group*: *print_esc*("endgroup"); **break**;
case *ex_space*: *print_esc*("␣"); **break**;
case *expand_after*:
  switch (*chr_code*) {
  case 0: *print_esc*("expandafter"); **break**;
    ⟨ Cases of *expandafter* for *print_cmd_chr* 1445 ⟩
  } **break**; /* there are no other cases */
case *halign*: *print_esc*("halign"); **break**;
case *hrule*: *print_esc*("hrule"); **break**;
case *ignore_spaces*: *print_esc*("ignorespaces"); **break**;
case *insert*: *print_esc*("insert"); **break**;
case *ital_corr*: *print_esc*("/"); **break**;
case *mark*:
  { *print_esc*("mark");
    if (*chr_code* > 0) *print_char*('s');
  } **break**;
case *math_accent*: *print_esc*("mathaccent"); **break**;
case *math_char_num*: *print_esc*("mathchar"); **break**;
case *math_choice*: *print_esc*("mathchoice"); **break**;
case *multiply*: *print_esc*("multiply"); **break**;
case *no_align*: *print_esc*("noalign"); **break**;
case *no_boundary*: *print_esc*("noboundary"); **break**;
case *no_expand*: *print_esc*("noexpand"); **break**;
case *non_script*: *print_esc*("nonscript"); **break**;
case *omit*: *print_esc*("omit"); **break**;
case *radical*: *print_esc*("radical"); **break**; **case** *read_to_cs*: **if** (*chr_code* ≡ 0) *print_esc*("read") ⟨ Cases of *read* for *print_cmd_chr* 1442 ⟩; **break**;
case *relax*: *print_esc*("relax"); **break**;
case *set_box*: *print_esc*("setbox"); **break**;
case *set_prev_graf*: *print_esc*("prevgraf"); **break**;
case *set_shape*:
  switch (*chr_code*) {
  case *par_shape_loc*: *print_esc*("parshape"); **break**;
    ⟨ Cases of *set_shape* for *print_cmd_chr* 1535 ⟩
  } **break**; /* there are no other cases */

  case *the*: **if** $(chr\_code \equiv 0)$ *print_esc*(`"the"`) ⟨ Cases of *the* for *print_cmd_chr* 1417 ⟩; **break**;
case *toks_register*: ⟨ Cases of *toks_register* for *print_cmd_chr* 1515 ⟩ **break**;
case *vadjust*: *print_esc*(`"vadjust"`); **break**;
case *valign*: *print_esc*(`"valign"`); **break**;
case *vcenter*: *print_esc*(`"vcenter"`); **break**;
case *vrule*: *print_esc*(`"vrule"`); **break**;

266. 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*) 266 ⟩ ≡
  printn_esc(*font_id_text*(*font*(*p*)))

This code is used in sections 173 and 175.

267. 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 $p \equiv save\_ptr - 1$; it contains three fields called *save_type*, *save_level*, and *save_index*, and it is interpreted in one of five ways:

1) If $save\_type(p) \equiv restore\_old\_value$, 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 $save\_type(p) \equiv restore\_zero$, 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 $save\_type(p) \equiv insert\_token$, then $save\_index(p)$ is a token that should be inserted into TEX's input when the current group ends.

4) If $save\_type(p) \equiv level\_boundary$, then $save\_level(p)$ is a code explaining what kind of group we were previously in, and $save\_index(p)$ points to the level boundary word at the bottom of the entries for that group. Furthermore, in extended ε-TEX mode, $save\_stack[p-1]$ contains the source line number at which the current level of grouping was entered.

5) If $save\_type(p) \equiv restore\_sa$, then *sa_chain* points to a chain of sparse array entries to be restored at the end of the current group. Furthermore $save\_index(p)$ and $save\_level(p)$ should replace the values of *sa_chain* and *sa_level* respectively.

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

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩

268. 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 ⟩ +≡

```
typedef int8_t group_code; /* save_level for a level boundary */
```

269. 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*.

270. ⟨ Global variables 13 ⟩ +≡

```
static memory_word save_stack[save_size + 1];
static int save_ptr; /* first unused entry on save_stack */
static int max_save_stack; /* maximum usage of save stack */
static quarterword cur_level; /* current nesting level for groups */
static group_code cur_group; /* current group type */
static int cur_boundary; /* where the current level begins */
```

271. 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;
```

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

```
#define check_full_save_stack
        if (save_ptr > max_save_stack) { max_save_stack = save_ptr;
          if (max_save_stack > save_size − 7) overflow("save␣size", save_size);
        }
```

273. 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(A) save_stack[save_ptr + A].i

  static void new_save_level(group_code c)      /* begin a new level of grouping */
  { check_full_save_stack;
    if (eTeX_ex) { saved(0) = line;
      incr(save_ptr);
    }
    save_type(save_ptr) = level_boundary;
    save_level(save_ptr) = cur_group;
    save_index(save_ptr) = cur_boundary;
    if (cur_level ≡ max_quarterword)
      overflow("grouping␣levels", max_quarterword − min_quarterword);
        /* quit if (cur_level + 1) is too big to be stored in eqtb */
    cur_boundary = save_ptr;
    cur_group = c;
#ifdef STAT
    if (tracing_groups > 0) group_trace(false);
#endif
    incr(cur_level);
    incr(save_ptr);
  }
```

274. 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.

```
static void eq_destroy(memory_word w)    /* gets ready to forget w */
    { pointer q;    /* equiv field of w */
    switch (eq_type_field(w)) {
  case call: case long_call: case outer_call: case long_outer_call: delete_token_ref(equiv_field(w));
    break;
  case glue_ref: delete_glue_ref(equiv_field(w)); break;
  case shape_ref:
    { q = equiv_field(w);    /* we need to free a \parshape block */
      if (q ≠ null) free_node(q, info(q) + info(q) + 1);
    } break;    /* such a block is 2n + 1 words long, where n ≡ info(q) */
  case box_ref: flush_node_list(equiv_field(w)); break;
    ⟨Cases for eq_destroy 1516⟩
  default: do_nothing;
    }
    }
```

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

```
static void eq_save(pointer p, quarterword l)    /* saves eqtb[p] */
{ check_full_save_stack;
  if (l ≡ level_zero) save_type(save_ptr) = restore_zero;
  else { save_stack[save_ptr] = eqtb[p];
    incr(save_ptr);
    save_type(save_ptr) = restore_old_value;
  }
  save_level(save_ptr) = l;
  save_index(save_ptr) = p;
  incr(save_ptr);
}
```

276. 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.

```
#ifdef STAT
#define assign_trace (A, B)
  if (tracing_assigns > 0) restore_trace(A, B);
#else
#define assign_trace (A, B)
#endif
  static void eq_define(pointer p, quarterword t, halfword e)      /* new data for eqtb */
  { if (eTeX_ex ∧ (eq_type(p) ≡ t) ∧ (equiv(p) ≡ e)) { assign_trace(p, "reassigning")
        eq_destroy(eqtb[p]);
        return;
      }
      assign_trace(p, "changing")
      if (eq_level(p) ≡ cur_level) eq_destroy(eqtb[p]);
      else if (cur_level > level_one) eq_save(p, eq_level(p));
      eq_level(p) = cur_level;
      eq_type(p) = t;
      equiv(p) = e;
      assign_trace(p, "into")
  }
```

277. 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.

```
  static void eq_word_define(pointer p, int w)
  { if (eTeX_ex ∧ (eqtb[p].i ≡ w)) { assign_trace(p, "reassigning")
        return;
      }
      assign_trace(p, "changing")
      if (xeq_level[p] ≠ cur_level) { eq_save(p, xeq_level[p]);
        xeq_level[p] = cur_level;
      }
      eqtb[p].i = w;
      assign_trace(p, "into")
  }
```

278. 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*.

```
static void geq_define(pointer p, quarterword t, halfword e)    /* global eq_define */
{ assign_trace(p, "globally changing")
  { eq_destroy(eqtb[p]);
    eq_level(p) = level_one;
    eq_type(p) = t;
    equiv(p) = e;
  }
  assign_trace(p, "into");
}
static void geq_word_define(pointer p, int w)    /* global eq_word_define */
{ assign_trace(p, "globally changing")
  { eqtb[p].i = w;
    xeq_level[p] = level_one;
  }
  assign_trace(p, "into");
}
```

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

```
static void save_for_after(halfword t)
{ if (cur_level > level_one) { check_full_save_stack;
    save_type(save_ptr) = insert_token;
    save_level(save_ptr) = level_zero;
    save_index(save_ptr) = t;
    incr(save_ptr);
  }
}
```

280. 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.

```
static void back_input(void);

static void unsave(void)    /* pops the top level off the save stack */
{ pointer p;    /* position to be restored */
  quarterword l;    /* saved level, if in fullword regions of eqtb */
  halfword t;    /* saved value of cur_tok */
  bool a;    /* have we already processed an \aftergroup ? */

  a = false;
  if (cur_level > level_one) { decr(cur_level);
    ⟨Clear off top level from save_stack 281⟩;
  }
  else confusion("curlevel");    /* unsave is not used when cur_group ≡ bottom_level */
}
```

281. ⟨ Clear off top level from *save_stack* 281 ⟩ ≡

```
loop { decr(save_ptr);
  if (save_type(save_ptr) ≡ level_boundary) goto done;
  p = save_index(save_ptr);
  if (save_type(save_ptr) ≡ insert_token) ⟨ Insert token p into TeX's input 325 ⟩
  else if (save_type(save_ptr) ≡ restore_sa) { sa_restore();
    sa_chain = p;
    sa_level = save_level(save_ptr);
  }
  else { if (save_type(save_ptr) ≡ restore_old_value) { l = save_level(save_ptr);
      decr(save_ptr);
    }
    else save_stack[save_ptr] = eqtb[undefined_control_sequence];
    ⟨ Store save_stack[save_ptr] in eqtb[p], unless eqtb[p] holds a global value 282 ⟩;
  }
}
done:
#ifdef STAT
  if (tracing_groups > 0) group_trace(true);
#endif
  if (grp_stack[in_open] ≡ cur_boundary) group_warning();
    /* groups possibly not properly nested with files */
  cur_group = save_level(save_ptr);
  cur_boundary = save_index(save_ptr); if (eTeX_ex) decr(save_ptr)
```

This code is used in section 280.

282. 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 282⟩ ≡

```
  if (p < int_base)
    if (eq_level(p) ≡ level_one) { eq_destroy(save_stack[save_ptr]);    /* destroy the saved value */
#ifdef STAT
      if (tracing_restores > 0) restore_trace(p, "retaining");
#endif
    }
    else { eq_destroy(eqtb[p]);    /* destroy the current value */
      eqtb[p] = save_stack[save_ptr];    /* restore the saved value */
#ifdef STAT
      if (tracing_restores > 0) restore_trace(p, "restoring");
#endif
    }
  else if (xeq_level[p] ≠ level_one) { eqtb[p] = save_stack[save_ptr];
    xeq_level[p] = l;
#ifdef STAT
    if (tracing_restores > 0) restore_trace(p, "restoring");
#endif
  }
  else {
#ifdef STAT
    if (tracing_restores > 0) restore_trace(p, "retaining");
#endif
  }
```

This code is used in section 281.

283. ⟨Declare ε-TEX procedures for tracing and input 283⟩ ≡

```
#ifdef STAT
  static void restore_trace(pointer p, char *s)    /* eqtb[p] has just been restored or retained */
  { begin_diagnostic();
    print_char('{');
    print(s);
    print_char('␣');
    show_eqtb(p);
    print_char('}');
    end_diagnostic(false);
  }
#endif
```

See also sections 1391, 1392, 1438, 1439, 1456, 1458, 1459, 1503, 1505, 1519, 1520, 1521, 1522, and 1523.

This code is used in section 267.

284. 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* 284 ⟩ ≡

```
if (save_ptr > 0)
  for (q = 0; q ≤ save_ptr − 1; q++) { if (equiv_field(save_stack[q]) ≡ p) { print_nl("SAVE(");
        print_int(q);
        print_char(')');
      }
  }
```

This code is used in section 171.

285. 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 ⟩ +≡

```
static int mag_set;    /* if nonzero, this magnification should be used henceforth */
```

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

```
mag_set = 0;
```

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

```
static void prepare_mag(void)
{ if ((mag_set > 0) ∧ (mag ≠ mag_set)) { 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 */
  }
  if ((mag ≤ 0) ∨ (mag > 32768)) {
    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);
  }
  mag_set = mag;
}
```

288. 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 $cs\_token\_flag \equiv 2^{12} - 1$ is larger than $2^8m + c$, yet it is small enough that $cs\_token\_flag + p < max\_halfword$; thus, a token fits comfortably in a halfword.

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

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

289. ⟨ Check the "constant" values for consistency 14 ⟩ +≡
  if $(cs\_token\_flag + undefined\_control\_sequence > max\_halfword)$ $bad = 21$;

290. 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 $m \equiv match$, it means that TEX should scan a parameter for the current macro; when $m \equiv end\_match$, it means that parameter matching should end and TEX should start reading the macro text; and when $m \equiv out\_param$, 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.

291. 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* 291⟩ ≡
```
static void show_token_list(int p, int q, int l)
{ int m, c;     /* pieces of a token */
  ASCII_code match_chr;     /* character used in a 'match' */
  ASCII_code n;     /* the highest parameter number, as an ASCII digit */
  match_chr = '#';
  n = '0';
  tally = 0;
  while ((p ≠ null) ∧ (tally < l)) { if (p ≡ q) ⟨Do magic computation 319⟩;
    ⟨Display token p, and return if there are problems 292⟩;
    p = link(p);
  }
  if (p ≠ null) print_esc("ETC.");
}
```
This code is used in section 118.

292. ⟨Display token p, and **return** if there are problems 292⟩ ≡
```
if ((p < hi_mem_min) ∨ (p > mem_end)) { print_esc("CLOBBERED.");
  return;
}
if (info(p) ≥ cs_token_flag) print_cs(info(p) − cs_token_flag);
else { m = info(p)/°400;
  c = info(p) % °400;
  if (info(p) < 0) print_esc("BAD.");
  else ⟨Display the token (m, c) 293⟩;
}
```
This code is used in section 291.

293. 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) 293 ⟩ ≡

```
switch (m) {
case left_brace: case right_brace: case math_shift: case tab_mark: case sup_mark: case sub_mark:
  case spacer: case letter: case other_char: printn(c); break;
case mac_param:
  { printn(c);
    printn(c);
  } break;
case out_param:
  { printn(match_chr);
    if (c ≤ 9) print_char(c + '0');
    else { print_char('!');
      return;
    }
  } break;
case match:
  { match_chr = c;
    printn(c);
    incr(n);
    print_char(n);
    if (n > '9') return;
  } break;
case end_match:
  if (c ≡ 0) print("->"); break;
default: print_esc("BAD.");
}
```

This code is used in section 292.

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

```
static void token_show(pointer p)
{ if (p ≠ null) show_token_list(link(p), null, 10000000);
}
```

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

```
static void print_meaning(void)
{ print_cmd_chr(cur_cmd, cur_chr);
  if (cur_cmd ≥ call) { print_char(':');
    print_ln();
    token_show(cur_chr);
  }
  else if ((cur_cmd ≡ top_bot_mark) ∧ (cur_chr < marks_code)) { print_char(':');
    print_ln();
    token_show(cur_mark[cur_chr]);
  }
}
```

296. 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 ⟩ +≡
  static eight_bits *cur_cmd*; /∗ current command set by *get_next* ∗/
  static halfword *cur_chr*; /∗ operand of current command ∗/
  static pointer *cur_cs*; /∗ control sequence found here, zero if none found ∗/
  static halfword *cur_tok*; /∗ packed representative of *cur_cmd* and *cur_chr* ∗/

297. 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(A)
        { print(A);
          print_ASCII(chr_code);
        }
⟨Declare the procedure called print_cmd_chr 297⟩ ≡
  static void print_cmd_chr(quarterword cmd, halfword chr_code){ int n;     /* temp variable */
      switch (cmd) {
    case left_brace: chr_cmd("begin-group␣character␣") break;
    case right_brace: chr_cmd("end-group␣character␣") break;
    case math_shift: chr_cmd("math␣shift␣character␣") break;
    case mac_param: chr_cmd("macro␣parameter␣character␣") break;
    case sup_mark: chr_cmd("superscript␣character␣") break;
    case sub_mark: chr_cmd("subscript␣character␣") break;
    case endv: print("end␣of␣alignment␣template"); break;
    case spacer: chr_cmd("blank␣space␣") break;
    case letter: chr_cmd("the␣letter␣") break;
    case other_char: chr_cmd("the␣character␣")
      break;⟨Cases of print_cmd_chr for symbolic printing of primitives 226⟩
    default: print("[unknown␣command␣code!]");
      }
      }
```

This code is used in section 251.

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

```
static void show_cur_cmd_chr(void)
{ int n;     /* level of \if...\fi nesting */
  int l;     /* line where \if started */
  pointer p;
  begin_diagnostic();
  print_nl("{");
  if (mode ≠ shown_mode) { print_mode(mode);
    print(":␣");
    shown_mode = mode;
  }
  print_cmd_chr(cur_cmd, cur_chr);
  if (tracing_ifs > 0)
    if (cur_cmd ≥ if_test)
      if (cur_cmd ≤ fi_or_else) { print(":␣");
        if (cur_cmd ≡ fi_or_else) { print_cmd_chr(if_test, cur_if);
          print_char('␣');
          n = 0;
          l = if_line;
        }
        else { n = 1;
          l = line;
        }
        p = cond_ptr;
        while (p ≠ null) { incr(n);
          p = link(p);
        }
        print("(level␣");
        print_int(n);
        print_char(')');
        print_if_line(l);
      }
  print_char('}');
  end_diagnostic(false);
}
```

299. 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 $stack[0 \to (ptr - 1)]$. 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 $stack[0 \to (ptr - 1)]$. 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 ⟩ +≡
typedef struct {
  quarterword *state_field*, *index_field*;
  halfword *start_field*, *loc_field*, *limit_field*, *name_field*;
  halfword *depth_field*;
} **in_state_record**;

300. ⟨ Global variables 13 ⟩ +≡
static in_state_record *input_stack*[*stack_size* + 1];
static int *input_ptr*; /∗ first unused location of *input_stack* ∗/
static int *max_in_stack*; /∗ largest value of *input_ptr* when pushing ∗/
static in_state_record *cur_input*; /∗ the "top" input state, according to convention (1) ∗/

301. 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 ∗/
#define *cur_depth* *cur_input.depth_field* /∗ nesting depth of current macro ∗/

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

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

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

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

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

1) *state* ≡ *mid_line* is the normal state.
2) *state* ≡ *skip_blanks* is like *mid_line*, but blanks are ignored.
3) *state* ≡ *new_line* is the state at the beginning of a line.

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

#**define** *mid_line* 1 /∗ *state* code when scanning a line of characters ∗/
#**define** *skip_blanks* (2 + *max_char_code*) /∗ *state* code when ignoring blanks ∗/
#**define** *new_line* (3 + *max_char_code* + *max_char_code*) /∗ *state* code at start of line ∗/

303. 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 $index \equiv 0$ when reading from the terminal and prompting the user for each line; then if the user types, e.g., '`\input paper`', we will have $index \equiv 1$ 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 $in\_open \equiv index$ 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 $name \equiv 0$, 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*] **int**' by analogy with *line_stack*.

\#**define** *terminal_input* ($name \equiv 0$) /∗ are we reading from the terminal? ∗/
\#**define** *cur_file* *input_file*[*index*] /∗ the current **alpha_file** variable ∗/

⟨ Global variables 13 ⟩ +≡
  static int *in_open*; /∗ the number of lines in the buffer, less one ∗/
  static int *open_parens*; /∗ the number of open text files ∗/
  static alpha_file *input_file0* [*max_in_open*], ∗**const** $input\_file = input\_file0 - 1$;
  static int *line*; /∗ current line number in the current source file ∗/
  static int *line_stack0* [*max_in_open*], ∗**const** $line\_stack = line\_stack0 - 1$;

304. 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 ⟩ +≡
static int *scanner_status*; /* can a subfile end now? */
static pointer *warning_index*; /* identifier relevant to non-*normal* scanner status */
static pointer *def_ref*; /* reference count of token list being defined */

305. 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* 305⟩ ≡

```
static void runaway(void)
{ pointer p;     /* head of runaway list */
  if (scanner_status > skipping) { print_nl("Runaway␣");
    switch (scanner_status) {
    case defining:
      { print("definition");
        p = def_ref;
      } break;
    case matching:
      { print("argument");
        p = temp_head;
      } break;
    case aligning:
      { print("preamble");
        p = hold_head;
      } break;
    case absorbing:
      { print("text");
        p = def_ref;
      }
    }     /* there are no other cases */
    print_char('?');
    print_ln();
    show_token_list(link(p), null, error_line − 10);
  }
}
```

This code is used in section 118.

306. 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 $state \equiv token\_list$, 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 $loc \equiv null$, the token list has been fully read.

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

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

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

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

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

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

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

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

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

#define *every_job_text* 12 /* *token_type* code for `\everyjob` */
#define *every_cr_text* 13 /* *token_type* code for `\everycr` */
#define *mark_text* 14 /* *token_type* code for `\topmark`, etc. */
#define *eTeX_text_offset* (*output_routine_loc* − *output_text*)
#define *every_eof_text* (*every_eof_loc* − *eTeX_text_offset*) /* *token_type* code for `\everyeof` */
#define *write_text* (*toks_base* − *eTeX_text_offset*) /* *token_type* code for `\write` */

307. 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 ⟩ +≡
static pointer *param_stack*[*param_size* + 1]; /* token list pointers for parameters */
static int *param_ptr*; /* first unused entry in *param_stack* */
static int *max_param_stack*; /* largest value of *param_ptr*, will be $\le$ *param_size* + 9 */

308. 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.

The same principle applies when entering the definition of a control sequence between `\csname` and `\endcsname`.

⟨ Global variables 13 ⟩ +≡
static int *align_state*; /* group level with respect to current alignment */
static int *incsname_state*; /* group level with respect to in csname state */

309. 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 ⟩ +≡
static int *base_ptr*; /* shallowest level shown by *show_context* */

310. 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.

```
static void show_context(void)   /* prints where the scanner is */
{ int old_setting;   /* saved selector setting */
  int nn;   /* number of contexts shown so far, less one */
  bool bottom_line;   /* have we reached the final context to be shown? */
  ⟨Local variables for formatting calculations 314⟩
  base_ptr = input_ptr;
  input_stack[base_ptr] = cur_input;   /* store current state */
  nn = −1;
  bottom_line = false;
  loop { cur_input = input_stack[base_ptr];   /* enter into the context */
    if ((state ≠ token_list))
      if ((name > 19) ∨ (base_ptr ≡ 0)) bottom_line = true;
    if ((base_ptr ≡ input_ptr) ∨ bottom_line ∨ (nn < error_context_lines))
      ⟨Display the current context 311⟩
    else if (nn ≡ error_context_lines) { print_nl("...");
      incr(nn);   /* omitted if error_context_lines < 0 */
    }
    if (bottom_line) goto done;
    decr(base_ptr);
  }
done: cur_input = input_stack[input_ptr];   /* restore original state */
}
```

311. ⟨Display the current context 311⟩ ≡

```
{ if ((base_ptr ≡ input_ptr) ∨ (state ≠ token_list) ∨ (token_type ≠ backed_up) ∨ (loc ≠ null))
      /* we omit backed-up token lists that have already been read */
  { tally = 0;   /* get ready to count characters */
    old_setting = selector;
    if (state ≠ token_list) { ⟨Print location of current line 312⟩;
      ⟨Pseudoprint the line 317⟩;
    }
    else { ⟨Print type of token list 313⟩;
      ⟨Pseudoprint the token list 318⟩;
    }
    selector = old_setting;   /* stop pseudoprinting */
    ⟨Print two lines using the tricky pseudoprinted information 316⟩;
    incr(nn);
  }
}
```

This code is used in section 310.

312. 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 312⟩ ≡
  if (name ≤ 17)
    if (terminal_input)
      if (base_ptr ≡ 0) print_nl("<*>");
      else print_nl("<insert>␣");
    else { print_nl("<read␣");
      if (name ≡ 17) print_char('*'); else print_int(name − 1);
      print_char('>');
    }
  else { print_nl("l.");
    if (index ≡ in_open) print_int(line);
    else print_int(line_stack[index + 1]);    /* input from a pseudo file */
  }
  print_char('␣')
```

This code is used in section 311.

313.

```
⟨Print type of token list 313⟩ ≡
  switch (token_type) {
  case parameter: print_nl("<argument>␣"); break;
  case u_template: case v_template: print_nl("<template>␣"); break;
  case backed_up:
    if (loc ≡ null) print_nl("<recently␣read>␣");
    else print_nl("<to␣be␣read␣again>␣"); break;
  case inserted: print_nl("<inserted␣text>␣"); break;
  case macro:
    { print_ln();
      print_cs(name);
    } break;
  case output_text: print_nl("<output>␣"); break;
  case every_par_text: print_nl("<everypar>␣"); break;
  case every_math_text: print_nl("<everymath>␣"); break;
  case every_display_text: print_nl("<everydisplay>␣"); break;
  case every_hbox_text: print_nl("<everyhbox>␣"); break;
  case every_vbox_text: print_nl("<everyvbox>␣"); break;
  case every_job_text: print_nl("<everyjob>␣"); break;
  case every_cr_text: print_nl("<everycr>␣"); break;
  case mark_text: print_nl("<mark>␣"); break;
  case every_eof_text: print_nl("<everyeof>␣"); break;
  case write_text: print_nl("<write>␣"); break;
  default: print_nl("?");    /* this should never happen */
  }
```

This code is used in section 311.

314. 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 % *error_line*] if $k < trick\_count$, otherwise character k is dropped. Initially we set $tally = 0$ and $trick\_count = 1000000$; then when we reach the point where transition from line 1 to line 2 should occur, we set $first\_count = tally$ and $trick\_count = \max(error\_line, tally + 1 + error\_line - half\_error\_line)$. 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 $k \equiv first\_count$, and the length of the context information gathered for line 2 is $m = \min(tally, trick\_count) - k$. If $l + k \le h$, where $h \equiv half\_error\_line$, we print $trick\_buf[0 \mathinner{..} k-1]$ after the descriptive information on line 1, and set $n = l + k$; here n is the length of line 1. If $l + k > h$, some cropping is necessary, so we set $n = h$ and print '...' followed by

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

where subscripts of *trick_buf* are circular modulo *error_line*. The second line consists of n spaces followed by $trick\_buf[k \mathinner{..} (k + m - 1)]$, 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 314⟩ ≡
```
int i; /* index into buffer */
int j; /* end of current line in buffer */
int l; /* length of descriptive information on line 1 */
int m; /* context information gathered for line 2 */
int n; /* length of line 1 */
int p; /* starting or ending place in trick_buf */
int q; /* temporary index */
```
This code is used in section 310.

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

```
#define begin_pseudoprint
        { l = tally;
          tally = 0;
          selector = pseudo;
          trick_count = 1000000;
        }
#define set_trick_count
        { first_count = tally;
          trick_count = tally + 1 + error_line − half_error_line;
          if (trick_count < error_line) trick_count = error_line;
        }
```

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

⟨Print two lines using the tricky pseudoprinted information 316⟩ ≡

```
if (trick_count ≡ 1000000) set_trick_count;     /* set_trick_count must be performed */
if (tally < trick_count) m = tally − first_count;
else m = trick_count − first_count;     /* context on line 2 */
if (l + first_count ≤ half_error_line) { p = 0;
  n = l + first_count;
}
else { print("...");
  p = l + first_count − half_error_line + 3;
  n = half_error_line;
}
for (q = p; q ≤ first_count − 1; q++) print_char(trick_buf[q % error_line]);
print_ln();
for (q = 1; q ≤ n; q++) print_char('␣');     /* print n spaces to begin line 2 */
if (m + n ≤ error_line) p = first_count + m;
else p = first_count + (error_line − n − 3);
for (q = first_count; q ≤ p − 1; q++) print_char(trick_buf[q % error_line]);
if (m + n > error_line) print("...")
```

This code is used in section 311.

317. 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 317⟩ ≡

```
begin_pseudoprint;
if (buffer[limit] ≡ end_line_char) j = limit;
else j = limit + 1;     /* determine the effective end of the line */
if (j > 0)
  for (i = start; i ≤ j − 1; i++) { if (i ≡ loc) set_trick_count;
    printn(buffer[i]);
  }
```

This code is used in section 311.

318. ⟨Pseudoprint the token list 318⟩ ≡

```
begin_pseudoprint;
if (token_type < macro) show_token_list(start, loc, 100000);
else show_token_list(link(start), loc, 100000)     /* avoid reference count */
```

This code is used in section 311.

319. 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 319⟩ ≡

```
set_trick_count
```

This code is used in section 291.

320. 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 */
        { if (input_ptr > max_in_stack) { max_in_stack = input_ptr;
            if (input_ptr ≡ stack_size) overflow("input␣stack␣size", stack_size);
          }
          input_stack[input_ptr] = cur_input; /* stack the record */
          incr(input_ptr);
        }
```

321. And of course what goes up must come down.

```
#define pop_input /* leave an input level, re-enter the old */
        { decr(input_ptr);
          cur_input = input_stack[input_ptr];
        }
```

322. Here is a procedure that starts a new level of token-list input, given a token list p and its type t. If $t \equiv$ *macro*, the calling routine should set *name* and *loc*.

```
#define back_list(A) begin_token_list(A, backed_up) /* backs up a simple token list */
#define ins_list(A) begin_token_list(A, inserted) /* inserts a simple token list */

  static void begin_token_list(pointer p, quarterword t)
  { push_input;
    state = token_list;
    start = p;
    token_type = t;
    if (t ≥ macro) /* the token list starts with a reference count */
    { add_token_ref(p);
      if (t ≡ macro) param_start = param_ptr;
      else { ⟨additional local variables for begin_token_list 1773⟩loc = link(p);
        if (tracing_macros > 1) { begin_diagnostic();
          print_nl("");
          switch (t) {
          case mark_text: print_esc("mark"); break;
          case write_text: print_esc("write"); break;
          default: print_cmd_chr(assign_toks, t − output_text + output_routine_loc);
          }
          print("->");
          token_show(p);
          end_diagnostic(false);
        }
        ⟨update the macro stack 1771⟩
      }
    }
    else loc = p;
  }
```

323. 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.

```
static void end_token_list(void)    /* leave a token-list input level */
{ if (token_type ≥ backed_up)    /* token list to be deleted */
  { if (token_type ≤ inserted) flush_list(start);
    else { delete_token_ref(start);    /* update reference count */
      if (token_type ≡ macro)    /* parameters must be flushed */
      {
        while (param_ptr > param_start) { decr(param_ptr);
          flush_list(param_stack[param_ptr]);
        }
      }
    }
  }
  else if (token_type ≡ u_template)
    if (align_state > 500000) align_state = 0;
    else fatal_error("(interwoven␣alignment␣preambles␣are␣not␣allowed)");
  pop_input;
  check_interrupt;
}
```

324. 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.

We charge the backup token to the current file and line.

```
static void back_input(void)    /* undoes one token of input */
{ pointer p;    /* a token list of length one */
  while ((state ≡ token_list) ∧ (loc ≡ null) ∧ (token_type ≠ v_template)) end_token_list();
      /* conserve stack space */
  p = get_avail();
  info(p) = cur_tok;
  fl_mem[p] = prof_file_line;
  if (cur_tok < right_brace_limit)
    if (cur_tok < left_brace_limit) 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 */
}
```

325. ⟨Insert token p into TEX's input 325⟩ ≡

```
{ t = cur_tok;
  cur_tok = p;
  if (a) { p = get_avail();
    info(p) = cur_tok;
    link(p) = loc;
    loc = p;
    start = p;
    if (cur_tok < right_brace_limit)
      if (cur_tok < left_brace_limit) decr(align_state);
      else incr(align_state);
  }
  else { back_input();
    a = eTeX_ex;
  }
  cur_tok = t;
}
```

This code is used in section 281.

326. 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.

```
static void back_error(void)  /* back up one token and call error */
{ OK_to_interrupt = false;
  back_input();
  OK_to_interrupt = true;
  error ( ) ;
}
static void ins_error(void)  /* back up one inserted token and call error */
{ OK_to_interrupt = false;
  back_input();
  token_type = inserted;
  OK_to_interrupt = true;
  error ( ) ;
}
```

327. 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*.

```
static void begin_file_reading(void)
{ if (in_open ≡ max_in_open) overflow("text␣input␣levels", max_in_open);
  if (first ≡ buf_size) overflow("buffer␣size", buf_size);
  incr(in_open);
  push_input;
  index = in_open;
  source_filename_stack[index] = Λ;    /* TEX Live */
  full_source_filename_stack[index] = Λ;    /* TEX Live */
  eof_seen[index] = false;
  grp_stack[index] = cur_boundary;
  if_stack[index] = cond_ptr;
  line_stack[index] = line;
  start = first;
  state = mid_line;
  name = 0;    /* terminal_input is now true */
  cur_file_num = terminal_file;
}
```

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

```
static void end_file_reading(void)
{ first = start;
  line = line_stack[index];
  if ((name ≡ 18) ∨ (name ≡ 19)) pseudo_close();
  else if (name > 17) a_close(&cur_file);    /* forget it */
  if (full_source_filename_stack[in_open] ≠ Λ) { free(full_source_filename_stack[in_open]);
    full_source_filename_stack[in_open] = Λ;
  }
  pop_input;
  decr(in_open);
}
```

329. 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.

```
static void clear_for_error_prompt(void)
{ while ((state ≠ token_list) ∧ terminal_input ∧
        (input_ptr > 0) ∧ (loc > limit)) end_file_reading();
  print_ln();
  clear_terminal;
}
```

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

⟨ Initialize the input routines 330 ⟩ ≡
  { *input_ptr* = 0;
    max_in_stack = 0;
    in_open = 0;
    open_parens = 0;
    max_buf_stack = 0;
    grp_stack[0] = 0;
    if_stack[0] = *null*;
    param_ptr = 0;
    max_param_stack = 0;
    first = *buf_size*;
    do { *buffer*[*first*] = 0;
      decr(*first*);
    } **while** (¬(*first* ≡ 0));
    scanner_status = *normal*;
    warning_index = *null*;
    first = 1;
    state = *new_line*;
    start = 1;
    index = 0;
    line = 0;
    name = 0;
    cur_depth = 0;
    force_eof = *false*;
    align_state = 1000000;
    if (¬*init_terminal*( )) *exit*(0);
    limit = *last*;
    first = *last* + 1;   /∗ *init_terminal* has set *loc* and *last* ∗/
  }

This code is used in section 1336.

331. 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 $cur\_cmd \equiv cur\_chr \equiv cur\_cs \equiv 0$ 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.)

332. 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.

The same is true for the input, for the warning message, since input is expected by default before every scanning and hence setting of *cur_cs*.

⟨ Global variables 13 ⟩ +≡
  static pointer *par_loc*; /∗ location of '`\par`' in *eqtb* ∗/
  static halfword *par_token*; /∗ token representing '`\par`' ∗/
  static pointer *input_loc*; /∗ location of '`\input`' in *eqtb* ∗/
  static halfword *input_token*; /∗ token representing '`\input`' ∗/

333. ⟨ Put each of TeX's primitives into the hash table 225 ⟩ +≡
  primitive("par", *par_end*, 256); /∗ cf. *scan_file_name* ∗/
  $par\_loc = cur\_val$;
  $par\_token = cs\_token\_flag + par\_loc$;

334. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *par_end*: *print_esc*("par"); **break**;

335. 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.

```
static void check_outer_validity(void)
{ pointer p;    /* points to inserted token list */
  pointer q;    /* auxiliary pointer */
  if (scanner_status ≠ normal) { deletions_allowed = false;
    ⟨Back up an outer control sequence so that it can be reread 336⟩;
    if (scanner_status > skipping) ⟨Tell the user what has run away and try to recover 337⟩
    else { 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) 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();
    }
    deletions_allowed = true;
  }
}
```

336. 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 336⟩ ≡
  if (cur_cs ≠ 0) { if ((state ≡ token_list) ∨ (name < 1) ∨ (name > 17)) { p = get_avail();
      info(p) = cs_token_flag + cur_cs;
      back_list(p);    /* prepare to read the control sequence again */
    }
    cur_cmd = spacer;
    cur_chr = '␣';    /* replace it by a space */
  }
```

This code is used in section 335.

337. ⟨ Tell the user what has run away and try to recover 337 ⟩ ≡

```
{ runaway();    /* print a definition, argument, or preamble */
  if (cur_cs ≡ 0) print_err("File␣ended");
  else { cur_cs = 0;
    print_err("Forbidden␣control␣sequence␣found");
  }
  print("␣while␣scanning␣");
  ⟨ Print either 'definition' or 'use' or 'preamble' or 'text', and insert tokens that should lead to
      recovery 338 ⟩;
  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 ( ) ;
}
```

This code is used in section 335.

338. 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 338 ⟩ ≡

```
p = get_avail();
switch (scanner_status) {
case defining:
  { print("definition");
    info(p) = right_brace_token + '}';
  } break;
case matching:
  { print("use");
    info(p) = par_token;
    long_state = outer_call;
  } break;
case aligning:
  { 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;
  } break;
case absorbing:
  { print("text");
    info(p) = right_brace_token + '}';
  }
}    /* there are no other cases */
ins_list(p)
```

This code is used in section 337.

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

static void *firm_up_the_line*(**void**);

340. 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.

static void *get_next*(**void**) /∗ sets *cur_cmd*, *cur_chr*, *cur_cs* to next token ∗/
{ /∗ go here to get the next input token ∗/ /∗ go here to eat the next character from a file ∗/ /∗ go here to digest it again ∗/ /∗ go here to start looking for a control sequence ∗/ /∗ go here when a control sequence has been found ∗/ /∗ go here when the next input token has been got ∗/
  int *k*; /∗ an index into *buffer* ∗/
  halfword *t*; /∗ a token ∗/
  int *cat*; /∗ *cat_code*(*cur_chr*), usually ∗/
  ASCII_code *c*, *cc*; /∗ constituents of a possible expanded code ∗/
  int *d*; /∗ number of excess characters in an expanded code ∗/
restart: *cur_cs* = 0;
  if (*state* ≠ *token_list*) ⟨ Input from external file, **goto** *restart* if no input found 342 ⟩
  else ⟨ Input from token list, **goto** *restart* if end of list or if a parameter needs to be expanded 356 ⟩;
  ⟨ If an alignment entry has just ended, take appropriate action 341 ⟩;
}

341. 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 341 ⟩ ≡
  if (*cur_cmd* ≤ *car_ret*)
    if (*cur_cmd* ≥ *tab_mark*)
      if (*align_state* ≡ 0) ⟨ Insert the $\langle v_j\rangle$ template and **goto** *restart* 788 ⟩

This code is used in section 340.

342. ⟨ Input from external file, **goto** *restart* if no input found 342 ⟩ ≡
{ *get_cur_chr*:
  if (*loc* ≤ *limit*) /∗ current line not yet finished ∗/
  { *cur_chr* = *buffer*[*loc*];
    incr(*loc*);
    ⟨ Set *cur_file_line* based on the information in *cur_input* 1747 ⟩
  reswitch: *cur_cmd* = *cat_code*(*cur_chr*);
    ⟨ Change state if necessary, and **goto** *get_cur_chr* if the current character should be ignored, or **goto** *reswitch* if the current character changes to another 343 ⟩;
  }
  else { *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 359 ⟩;
    check_interrupt;
    goto *get_cur_chr*;
  }
}

This code is used in section 340.

343. 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*(*A*) **case** *mid_line* + *A*: **case** *skip_blanks* + *A*: **case** *new_line* + *A*

⟨ Change state if necessary, and **goto** *get_cur_chr* if the current character should be ignored, or **goto** *reswitch* if the current character changes to another 343 ⟩ ≡
  switch (*state* + *cur_cmd*) {
  ⟨ Cases where character is ignored 344 ⟩: **goto** *get_cur_chr*;
  any_state_plus(*escape*): ⟨ Scan a control sequence and set *state*: = *skip_blanks* or *mid_line* 353 ⟩ **break**;
  any_state_plus(*active_char*):
    ⟨ Process an active-character control sequence and set *state*: = *mid_line* 352 ⟩ **break**;
  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* 351 ⟩ **break**;
  any_state_plus(*invalid_char*): ⟨ Decry the invalid character and **goto** *restart* 345 ⟩
  ⟨ Handle situations involving spaces, braces, changes of state 346 ⟩
  default: *do_nothing*;
  }

This code is used in section 342.

344. ⟨ Cases where character is ignored 344 ⟩ ≡
  any_state_plus(*ignore*): **case** *skip_blanks* + *spacer*: **case** *new_line* + *spacer*

This code is used in section 343.

345. We go to *restart* instead of to *get_cur_chr*, 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* 345 ⟩ ≡
  { *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*;
  }

This code is used in section 343.

346. **#define** *add_delims_to*(*A*) *A* + *math_shift*: *A* + *tab_mark*: *A* + *mac_param*: *A* + *sub_mark*: *A* + *letter*: *A* + *other_char*

⟨Handle situations involving spaces, braces, changes of state 346⟩ ≡
case *mid_line* + *spacer*: ⟨Enter *skip_blanks* state, emit a space 348⟩ **break**;
case *mid_line* + *car_ret*: ⟨Finish line, emit a space 347⟩ **break**;
case *skip_blanks* + *car_ret*: *any_state_plus*(*comment*): ⟨Finish line, **goto switch** 349⟩
case *new_line* + *car_ret*: ⟨Finish line, emit a `\par` 350⟩ **break**;
case *mid_line* + *left_brace*: *incr*(*align_state*); **break**;
case *skip_blanks* + *left_brace*: **case** *new_line* + *left_brace*:
{ *state* = *mid_line*;
incr(*align_state*);
} **break**;
case *mid_line* + *right_brace*: *decr*(*align_state*); **break**;
case *skip_blanks* + *right_brace*: **case** *new_line* + *right_brace*:
{ *state* = *mid_line*;
decr(*align_state*);
} **break**;
add_delims_to(**case** *skip_blanks*): *add_delims_to*(**case** *new_line*): *state* = *mid_line*; **break**;

This code is used in section 343.

347. When a character of type *spacer* gets through, its character code is changed to "␣" = 040. 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 347⟩ ≡
{ *loc* = *limit* + 1;
cur_cmd = *spacer*;
cur_chr = '␣';
}

This code is used in section 346.

348. The following code is performed only when *cur_cmd* ≡ *spacer*.

⟨Enter *skip_blanks* state, emit a space 348⟩ ≡
{ *state* = *skip_blanks*;
cur_chr = '␣';
}

This code is used in section 346.

349. ⟨Finish line, **goto switch** 349⟩ ≡
{ *loc* = *limit* + 1;
goto *get_cur_chr*;
}

This code is used in section 346.

350. ⟨Finish line, emit a `\par` 350⟩ ≡

$\{$ $loc = limit + 1$;
  $cur\_cs = par\_loc$;
  $cur\_cmd = eq\_type(cur\_cs)$;
  $cur\_chr = equiv(cur\_cs)$;
  if $(cur\_cmd \geq outer\_call)$ $check\_outer\_validity()$;
$\}$

This code is used in section 346.

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

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

⟨If this sup_mark starts an expanded character like `^^A` or `^^df`, then **goto** $reswitch$, otherwise set $state: = mid\_line$ 351⟩ ≡

$\{$ **if** $(cur\_chr \equiv buffer[loc])$
  if $(loc < limit)$ $\{$ $c = buffer[loc + 1]$; **if** $(c < °200)$ /* yes we have an expanded char */
    $\{$ $loc = loc + 2$;
      if $(is\_hex(c))$
        if $(loc \leq limit)$ $\{$ $cc = buffer[loc]$; **if** $(is\_hex(cc))$ $\{$ $incr(loc)$;
            $hex\_to\_cur\_chr$;
            goto $reswitch$;
          $\}$
        $\}$
      if $(c < °100)$ $cur\_chr = c + °100$; **else** $cur\_chr = c - °100$;
      goto $reswitch$;
    $\}$
  $\}$
  $state = mid\_line$;
$\}$

This code is used in section 343.

352. ⟨Process an active-character control sequence and set $state: = mid\_line$ 352⟩ ≡

$\{$ $cur\_cs = cur\_chr + active\_base$;
  $cur\_cmd = eq\_type(cur\_cs)$;
  $cur\_chr = equiv(cur\_cs)$;
  $state = mid\_line$;
  if $(cur\_cmd \geq outer\_call)$ $check\_outer\_validity()$;
$\}$

This code is used in section 343.

353. 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* 353⟩ ≡
{ **if** (*loc* > *limit*) *cur_cs* = *null_cs*; /∗ *state* is irrelevant in this case ∗/
 else { *start_cs*: *k* = *loc*;
  cur_chr = *buffer*[*k*];
  cat = *cat_code*(*cur_chr*);
  incr(*k*);
  if (*cat* ≡ *letter*) *state* = *skip_blanks*;
  else if (*cat* ≡ *spacer*) *state* = *skip_blanks*;
  else *state* = *mid_line*;
  if ((*cat* ≡ *letter*) ∧ (*k* ≤ *limit*)) ⟨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* 355⟩
  else ⟨If an expanded code is present, reduce it and **goto** *start_cs* 354⟩;
  cur_cs = *single_base* + *buffer*[*loc*];
  incr(*loc*);
 }
found: *cur_cmd* = *eq_type*(*cur_cs*);
 cur_chr = *equiv*(*cur_cs*);
 if (*cur_cmd* ≥ *outer_call*) *check_outer_validity*( );
}

This code is used in section 343.

354. Whenever we reach the following piece of code, we will have $cur\_chr \equiv buffer[k-1]$ and $k \le limit+1$ and $cat \equiv cat\_code(cur\_chr)$. If an expanded code like `^^A` or `^^df` appears in $buffer[(k-1) \mathinner{..} (k+1)]$ or $buffer[(k-1) \mathinner{..} (k+2)]$, 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* 354⟩ ≡

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

This code is used in sections 353 and 355.

355. ⟨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* 355⟩ ≡

```
{ do { cur_chr = buffer[k];
    cat = cat_code(cur_chr);
    incr(k);
  } while (¬((cat ≠ letter) ∨ (k > limit)));
  ⟨If an expanded code is present, reduce it and goto start_cs 354⟩;
  if (cat ≠ letter) decr(k);    /* now k points to first nonletter */
  if (k > loc + 1)    /* multiletter control sequence has been scanned */
  { cur_cs = id_lookup(loc, k − loc);
    loc = k;
    goto found;
  }
}
```

This code is used in section 353.

356. Let's consider now what happens when *get_next* is looking at a token list. We restore the file and line information.

⟨Input from token list, **goto** *restart* if end of list or if a parameter needs to be expanded 356⟩ ≡

```
if (loc ≠ null)   /* list not exhausted */
{ t = info(loc);
  cur_file_line = fl_mem[loc];
  loc = link(loc);   /* move to next */
  if (t ≥ cs_token_flag)   /* a control sequence token */
  { cur_cs = t − cs_token_flag;
    cur_cmd = eq_type(cur_cs);
    cur_chr = equiv(cur_cs);
    if (cur_cmd ≥ outer_call)
      if (cur_cmd ≡ dont_expand) ⟨Get the next token, suppressing expansion 357⟩
      else check_outer_validity();
  }
  else { cur_cmd = t/°400;
    cur_chr = t % °400;
    switch (cur_cmd) {
    case left_brace: incr(align_state); break;
    case right_brace: decr(align_state); break;
    case out_param: ⟨Insert macro parameter and goto restart 358⟩
    default: do_nothing;
    }
  }
}
else {   /* we are done with this token list */
  end_token_list();
  goto restart;   /* resume previous level */
}
```

This code is used in section 340.

357. 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 357⟩ ≡

```
{ cur_cs = info(loc) − cs_token_flag;
  loc = null;
  cur_cmd = eq_type(cur_cs);
  cur_chr = equiv(cur_cs);
  if (cur_cmd > max_command) { cur_cmd = relax;
    cur_chr = no_expand_flag;
  }
}
```

This code is used in section 356.

358. ⟨Insert macro parameter and **goto** *restart* 358⟩ ≡

```
{ begin_token_list(param_stack[param_start + cur_chr − 1], parameter);
  goto restart;
}
```

This code is used in section 356.

359. 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* $< 0) \lor ($*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 359⟩ ≡

```
if (name > 17) ⟨Read next line of file into buffer, or goto restart if the file has ended 361⟩
else { if (¬terminal_input)    /* \read line has ended */
    { cur_cmd = 0;
      cur_chr = 0;
      ⟨Set cur_file_line when a \read line ends 1748⟩
      return;
    }
    if (input_ptr > 0)    /* text was inserted during error recovery */
    { end_file_reading();
      goto restart;    /* resume previous level */
    }
    if (selector < log_only) open_log_file();
    if (interaction > nonstop_mode) { if (end_line_char_inactive) incr(limit);
      if (limit ≡ start)    /* 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) decr(limit);
      else buffer[limit] = end_line_char;
      first = limit + 1;
      loc = start;
    }
    else fatal_error("***␣(job␣aborted,␣no␣legal␣\\end␣found)");
        /* nonstop mode, which is intended for overnight batch processing, never waits for on-line input */
}
```

This code is used in section 342.

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

⟨Global variables 13⟩ +≡

```
static bool force_eof;    /* should the next \input be aborted early? */
```

361. ⟨ Read next line of file into *buffer*, or **goto** *restart* if the file has ended 361 ⟩ ≡

```
{ incr(line);
  first = start;
  ⟨ check line for overflow 1742 ⟩
  if (¬force_eof)
    if (name ≤ 19) { if (pseudo_input())   /* not end of file */
        firm_up_the_line();   /* this sets limit */
      else if ((every_eof ≠ null) ∧ ¬eof_seen[index]) { limit = first − 1;
        eof_seen[index] = true;   /* fake one empty line */
        begin_token_list(every_eof, every_eof_text);
        goto restart;
      }
      else force_eof = true;
    }
    else { if (input_ln(&cur_file, true))   /* not end of file */
        firm_up_the_line();   /* this sets limit */
      else if ((every_eof ≠ null) ∧ ¬eof_seen[index]) { limit = first − 1;
        eof_seen[index] = true;   /* fake one empty line */
        begin_token_list(every_eof, every_eof_text);
        goto restart;
      }
      else force_eof = true;
    }
  if (force_eof) { if (tracing_nesting > 0)
      if ((grp_stack[in_open] ≠ cur_boundary) ∨
            (if_stack[in_open] ≠ cond_ptr)) file_warning();
          /* give warning for some unfinished groups and/or conditionals */
    if (name ≥ 19) { 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;
  }
  if (end_line_char_inactive) decr(limit);
  else buffer[limit] = end_line_char;
  first = limit + 1;
  loc = start;   /* ready to read */
}
```

This code is used in section 359.

362. 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.

```
static void firm_up_the_line(void)
{ int k;      /* an index into buffer */
  limit = last;
  if (pausing > 0)
    if (interaction > nonstop_mode) { wake_up_terminal;
      print_ln();
      if (start < limit)
        for (k = start; k ≤ limit − 1; k++) printn(buffer[k]);
      first = limit;
      prompt_input("=>");      /* wait for user response */
      if (last > first) { for (k = first; k ≤ last − 1; k++)      /* move line down in buffer */
          buffer[k + start − first] = buffer[k];
        limit = start + last − first;
      }
    }
}
```

363. 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*.

364. 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.

```
static void get_token(void)      /* sets cur_cmd, cur_chr, cur_tok */
{ no_new_control_sequence = false;
  get_next();
  no_new_control_sequence = true;
  if (cur_cs ≡ 0) cur_tok = (cur_cmd * °400) + cur_chr;
  else cur_tok = cs_token_flag + cur_cs;
}
```

365. 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 388⟩
⟨Declare the procedure called insert_relax 378⟩
⟨Declare ε-TeX procedures for expanding 1434⟩
  static void pass_text(void);
  static void start_input(void);
  static void conditional(void);
  static void get_x_token(void);
  static void conv_toks(void);
  static void ins_the_toks(void);

  static void expand(void)
  { halfword t;  /* token that is being "expanded after" */
    pointer p, q, r;  /* for list manipulation */
    int j;  /* index into buffer */
    int cv_backup;  /* to save the global quantity cur_val */
    small_number cvl_backup, radix_backup, co_backup;  /* to save cur_val_level, etc. */
    pointer backup_backup;  /* to save link(backup_head) */
    small_number save_scanner_status;  /* temporary storage of scanner_status */

    cv_backup = cur_val;
    cvl_backup = cur_val_level;
    radix_backup = radix;
    co_backup = cur_order;
    backup_backup = link(backup_head);
  reswitch:
    if (cur_cmd < call) ⟨Expand a nonmacro 366⟩
    else if (cur_cmd < end_template) macro_call();
    else ⟨Insert a token containing frozen_endv 374⟩;
    cur_val = cv_backup;
    cur_val_level = cvl_backup;
    radix = radix_backup;
    cur_order = co_backup;
    link(backup_head) = backup_backup;
  }
```

366. ⟨ Expand a nonmacro 366 ⟩ ≡

```
{ if (tracing_commands > 1) show_cur_cmd_chr();
  switch (cur_cmd) {
  case top_bot_mark: ⟨ Insert the appropriate mark text into the scanner 385 ⟩ break;
  case expand_after:
    switch (cur_chr) {
    case 0: ⟨ Expand the token after the next token 367 ⟩ break;
    case 1: ⟨ Negate a boolean conditional and goto reswitch 1447 ⟩ break;
      ⟨ Cases for expandafter 1586 ⟩
    } break;    /* there are no other cases */
  case no_expand: ⟨ Suppress expansion of the next token 368 ⟩ break;
  case cs_name: ⟨ Manufacture a control sequence name 371 ⟩ break;
  case convert: conv_toks(); break;    /* this procedure is discussed in Part 27 below */
  case the: ins_the_toks(); break;    /* this procedure is discussed in Part 27 below */
  case if_test: conditional(); break;    /* this procedure is discussed in Part 28 below */
  case fi_or_else: ⟨ Terminate the current conditional and skip to \fi 509 ⟩ break;
  case input: ⟨ Initiate or terminate input from a file 377 ⟩; break;
  default: ⟨ Complain about an undefined macro 369 ⟩
  }
}
```

This code is used in section 365.

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

⟨ Expand the token after the next token 367 ⟩ ≡

```
{ get_token();
  t = cur_tok;
  get_token();
  if (cur_cmd > max_command) expand(); else back_input();
  cur_tok = t;
  back_input();
}
```

This code is used in section 366.

368. 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 368 ⟩ ≡

```
{ save_scanner_status = scanner_status;
  scanner_status = normal;
  get_token();
  scanner_status = save_scanner_status;
  t = cur_tok;
  back_input();    /* now start and loc point to the backed-up token t */
  if (t ≥ cs_token_flag) { p = get_avail();
    info(p) = cs_token_flag + frozen_dont_expand;
    link(p) = loc;
    start = p;
    loc = p;
  }
}
```

This code is used in section 366.

369. ⟨ Complain about an undefined macro 369 ⟩ ≡

```
{ 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 ( ) ;
}
```

This code is used in section 366.

370. 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. Here we add code to store file and line information for each token.

```
#define store_new_token(A)
        { q = get_avail();
          link(p) = q;
          info(q) = A;
          fl_mem[q] = cur_file_line;
          p = q;    /* link(p) is null */
        }
#define fast_store_new_token(A)
        { fast_get_avail(q);
          link(p) = q;
          info(q) = A;
          fl_mem[q] = cur_file_line;
          p = q;    /* link(p) is null */
        }
```

371. ⟨ Manufacture a control sequence name 371 ⟩ ≡

```
{ r = get_avail();
  p = r;    /* head of the list of characters */
  incr(incsname_state);
  do { get_x_token();
    if (cur_cs ≡ 0) store_new_token(cur_tok);
  } while (¬(cur_cs ≠ 0));
  if (cur_cmd ≠ end_cs_name) ⟨ Complain about missing \endcsname 372 ⟩;
  decr(incsname_state);
  ⟨ Look up the characters of list r in the hash table, and set cur_cs 373 ⟩;
  flush_list(r);
  if (eq_type(cur_cs) ≡ undefined_cs) { eq_define(cur_cs, relax, 256);
      /* N.B.: The save_stack might change */
  }    /* the control sequence will now match '\relax' */
  cur_tok = cur_cs + cs_token_flag;
  back_input();
}
```

This code is used in section 366.

372. ⟨ Complain about missing \endcsname 372 ⟩ ≡

```
{ 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();
}
```

This code is used in sections 371 and 1449.

373. ⟨ Look up the characters of list r in the hash table, and set *cur_cs* 373 ⟩ ≡

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

This code is used in section 371.

374. 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* 374 ⟩ ≡
  { $cur\_tok = cs\_token\_flag + frozen\_endv$;
    back_input( );
  }

This code is used in section 365.

375. 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 225 ⟩ +≡
  primitive(`"input"`, *input*, 0);
  $input\_loc = cur\_val$;
  $input\_token = cs\_token\_flag + input\_loc$;
  primitive(`"endinput"`, *input*, 1);

376. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
  case *input*: **if** ($chr\_code \equiv 0$) *print_esc*(`"input"`)
  ⟨ Cases of *input* for *print_cmd_chr* 1430 ⟩;
  else *print_esc*(`"endinput"`); **break**;

377. ⟨ Initiate or terminate input from a file 377 ⟩ ≡
  if ($cur\_chr \equiv 1$) $force\_eof = true$
  ⟨ Cases for *input* 1431 ⟩;
  else
  if (*name_in_progress*) *insert_relax*( );
  else *start_input*( )

This code is used in section 366.

378. 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* 378 ⟩ ≡
  static void *insert_relax*(**void**)
  { $cur\_tok = cs\_token\_flag + cur\_cs$;
    back_input( );
    $cur\_tok = cs\_token\_flag + frozen\_relax$;
    back_input( );
    $token\_type = inserted$;
  }

This code is used in section 365.

379. 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.

```
static void get_x_token(void)    /* sets cur_cmd, cur_chr, cur_tok, and expands macros */
{ restart: get_next();
  if (cur_cmd ≤ max_command) goto done;
  if (cur_cmd ≥ call)
    if (cur_cmd < end_template) macro_call();
    else { cur_cs = frozen_endv;
      cur_cmd = endv;
      goto done;    /* cur_chr ≡ null_list */
    }
  else expand();
  goto restart;
done:
  if (cur_cs ≡ 0) cur_tok = (cur_cmd * °400) + cur_chr;
  else cur_tok = cs_token_flag + cur_cs;
}
```

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

```
static void x_token(void)    /* get_x_token without the initial get_next */
{ while (cur_cmd > max_command) { expand();
    get_next();
  }
  if (cur_cs ≡ 0) cur_tok = (cur_cmd * °400) + cur_chr;
  else cur_tok = cs_token_flag + cur_cs;
}
```

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

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

```
#define marks_code 5    /* add this for \topmarks etc. */

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

⟨ Global variables 13 ⟩ +≡

```
static pointer cur_mark0[split_bot_mark_code − top_mark_code + 1],
    *const cur_mark = cur_mark0 − top_mark_code;    /* token lists for marks */
```

382. ⟨ Set initial values of key variables 21 ⟩ +≡
$top\_mark = null$;
$first\_mark = null$;
$bot\_mark = null$;
$split\_first\_mark = null$;
$split\_bot\_mark = null$;

383. ⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
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*);

384. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *top_bot_mark*:
{ **switch** ((*chr_code* % *marks_code*)) {
case *first_mark_code*: *print_esc*(`"firstmark"`); **break**;
case *bot_mark_code*: *print_esc*(`"botmark"`); **break**;
case *split_first_mark_code*: *print_esc*(`"splitfirstmark"`); **break**;
case *split_bot_mark_code*: *print_esc*(`"splitbotmark"`); **break**;
default: *print_esc*(`"topmark"`);
}
if ($chr\_code \geq marks\_code$) *print_char*(`'s'`);
} **break**;

385. The following code is activated when $cur\_cmd \equiv top\_bot\_mark$ and when *cur_chr* is a code like *top_mark_code*.

⟨ Insert the appropriate mark text into the scanner 385 ⟩ ≡
{ $t = cur\_chr \,\%\, marks\_code$;
if ($cur\_chr \geq marks\_code$) *scan_register_num*( ); **else** $cur\_val = 0$;
if ($cur\_val \equiv 0$) $cur\_ptr = cur\_mark[t]$;
else ⟨ Compute the mark pointer for mark type t and class *cur_val* 1506 ⟩;
if ($cur\_ptr \neq null$) *begin_token_list*(*cur_ptr*, *mark_text*);
}

This code is used in section 366.

386. 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 ⟩ +≡
static int *long_state*; /* governs the acceptance of `\par` */

387. 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 ⟩ +≡
static pointer *pstack*[9]; /∗ arguments supplied to a macro ∗/

388. 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* 388 ⟩ ≡
static void *macro_call*(**void**) /∗ invokes a user-defined control sequence ∗/
{ **pointer** *r*; /∗ current node in the macro's token list ∗/
  pointer *p*; /∗ current node in parameter token list being built ∗/
  pointer *q*; /∗ new node being put into the token list ∗/
  pointer *s*; /∗ backup pointer for parameter matching ∗/
  pointer *t*; /∗ cycle pointer for backup recovery ∗/
  pointer *u*, *v*; /∗ auxiliary pointers for backup recovery ∗/
  pointer *rbrace_ptr*; /∗ one step before the last *right_brace* token ∗/
  small_number *n*; /∗ the number of parameters scanned ∗/
  halfword *unbalance*; /∗ unmatched left braces in current parameter ∗/
  int *m*; /∗ the number of tokens or groups (usually) ∗/
  pointer *ref_count*; /∗ start of the token list ∗/
  small_number *save_scanner_status*; /∗ *scanner_status* upon entry ∗/
  pointer *save_warning_index*; /∗ *warning_index* upon entry ∗/
  ASCII_code *match_chr*; /∗ character used in parameter ∗/
  ⟨ additional local variables for *macro_call* 1770 ⟩
  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) ⟨ Show the text of the macro being expanded 400 ⟩;
  if (*info*(*r*) ≡ *protected_token*) *r* = *link*(*r*);
  if (*info*(*r*) ≠ *end_match_token*) ⟨ Scan the parameters and make *link*(*r*) point to the macro body; but **goto** *end* if an illegal `\par` is detected 390 ⟩;
  ⟨ Feed the macro body and its parameters to the scanner 389 ⟩;
end: *scanner_status* = *save_scanner_status*;
  warning_index = *save_warning_index*;
  ⟨ update the macro stack 1771 ⟩
}

This code is used in section 365.

389. 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 389 ⟩ ≡

```
while ((state ≡ token_list) ∧ (loc ≡ null) ∧ (token_type ≠ v_template)) end_token_list();
      /* conserve stack space */
begin_token_list(ref_count, macro);
name = warning_index;
loc = link(r);
if (n > 0) { if (param_ptr + n > max_param_stack) { max_param_stack = param_ptr + n;
      if (max_param_stack > param_size) overflow("parameter␣stack␣size", param_size);
   }
   for (m = 0; m ≤ n − 1; m++) param_stack[param_ptr + m] = pstack[m];
   param_ptr = param_ptr + n;
}
```

This code is used in section 388.

390. 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 $s \equiv null$ 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 **goto** *end* if an illegal `\par` is detected 390 ⟩ ≡

```
{ scanner_status = matching;
   unbalance = 0;
   long_state = eq_type(cur_cs);
   if (long_state ≥ outer_call) long_state = long_state − 2;
   do { link(temp_head) = null;
      if ((info(r) > match_token + 255) ∨ (info(r) < match_token)) s = null;
      else { match_chr = info(r) − match_token;
         s = link(r);
         r = s;
         p = temp_head;
         m = 0;
      }
      ⟨ Scan a parameter until its delimiter string has been found; or, if s = null, simply scan the delimiter
            string 391 ⟩;     /* now info(r) is a token whose command code is either match or end_match */
   } while (¬(info(r) ≡ end_match_token));
}
```

This code is used in section 388.

391. 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* $\equiv$ *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 391⟩ ≡
resume: *get_token*( ); /∗ set *cur_tok* to the next token of input ∗/
  if (*cur_tok* $\equiv$ *info*(r)) ⟨Advance r; **goto** *found* if the parameter delimiter has been fully matched, otherwise **goto** *resume* 393⟩;
  ⟨Contribute the recently matched tokens to the current parameter, and **goto** *resume* if a partial match is still in effect; but abort if $s = null$ 396⟩;
  if (*cur_tok* $\equiv$ *par_token*)
    if (*long_state* $\neq$ *long_call*) ⟨Report a runaway argument and abort 395⟩;
  if (*cur_tok* < *right_brace_limit*)
    if (*cur_tok* < *left_brace_limit*) ⟨Contribute an entire group to the current parameter 398⟩
    else ⟨Report an extra right brace and **goto** *resume* 394⟩
  else ⟨Store the current token, but **goto** *resume* if it is a blank space that would become an undelimited parameter 392⟩;
  incr(m);
  if (*info*(r) > *end_match_token*) **goto** *resume*;
  if (*info*(r) < *match_token*) **goto** *resume*;
found:
  if ($s \neq null$) ⟨Tidy up the parameter just scanned, and tuck it away 399⟩

This code is used in section 390.

392. ⟨Store the current token, but **goto** *resume* if it is a blank space that would become an undelimited parameter 392⟩ ≡
  { **if** (*cur_tok* $\equiv$ *space_token*)
      if (*info*(r) $\leq$ *end_match_token*)
        if (*info*(r) $\geq$ *match_token*) **goto** *resume*;
    store_new_token(*cur_tok*);
  }

This code is used in section 391.

393. 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** *resume* 393⟩ ≡
  { $r = link(r)$;
    if ((*info*(r) $\geq$ *match_token*) $\land$ (*info*(r) $\leq$ *end_match_token*)) { **if** (*cur_tok* < *left_brace_limit*)
        decr(*align_state*);
      goto *found*;
    }
    else goto *resume*;
  }

This code is used in section 391.

394. ⟨Report an extra right brace and **goto** *resume* 394⟩ ≡

```
{ 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 resume;
}   /* a white lie; the \par won't always trigger a runaway */
```

This code is used in section 391.

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

⟨Report a runaway argument and abort 395⟩ ≡

```
{ if (long_state ≡ call) { 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();
  }
  pstack[n] = link(temp_head);
  align_state = align_state − unbalance;
  for (m = 0; m ≤ n; m++) flush_list(pstack[m]);
  goto end;
}
```

This code is used in sections 391 and 398.

396. 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 $s \equiv r$ 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** *resume* if a partial match is still in effect; but abort if $s = null$ 396⟩ ≡

```
if (s ≠ r)
  if (s ≡ null) ⟨Report an improper use of the macro and abort 397⟩
  else { t = s;
    do { store_new_token(info(t));
      incr(m);
      u = link(t);
      v = s;
      loop { if (u ≡ r)
          if (cur_tok ≠ info(v)) goto done;
          else { r = link(v);
            goto resume;
          }
        if (info(u) ≠ info(v)) goto done;
        u = link(u);
        v = link(v);
      }
    done: t = link(t);
    } while (¬(t ≡ r));
    r = s;    /* at this point, no tokens are recently matched */
  }
```

This code is used in section 391.

397. ⟨Report an improper use of the macro and abort 397⟩ ≡

```
{ 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();
  goto end;
}
```

This code is used in section 396.

398. ⟨Contribute an entire group to the current parameter 398⟩ ≡
{ *unbalance* = 1;
  loop { *fast_store_new_token*(*cur_tok*);
    get_token( );
    if (*cur_tok* ≡ *par_token*)
      if (*long_state* ≠ *long_call*) ⟨Report a runaway argument and abort 395⟩;
    if (*cur_tok* < *right_brace_limit*)
      if (*cur_tok* < *left_brace_limit*) *incr*(*unbalance*);
      else { *decr*(*unbalance*);
        if (*unbalance* ≡ 0) **goto** *done1*;
      }
  }
done1: *rbrace_ptr* = *p*;
  store_new_token(*cur_tok*);
}

This code is used in section 391.

399. 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 399⟩ ≡
{ **if** ((*m* ≡ 1) ∧ (*info*(*p*) < *right_brace_limit*)) { *link*(*rbrace_ptr*) = *null*;
    free_avail(*p*);
    p = *link*(*temp_head*);
    pstack[*n*] = *link*(*p*);
    free_avail(*p*);
  }
  else *pstack*[*n*] = *link*(*temp_head*);
  incr(*n*);
  if (*tracing_macros* > 0) { *begin_diagnostic*( );
    print_nl("");
    printn(*match_chr*);
    print_int(*n*);
    print("<-");
    show_token_list(*pstack*[*n* − 1], *null*, 1000);
    end_diagnostic(*false*);
  }
}

This code is used in section 391.

400. ⟨Show the text of the macro being expanded 400⟩ ≡
{ *begin_diagnostic*( );
  print_ln( );
  print_cs(*warning_index*);
  token_show(*ref_count*);
  end_diagnostic(*false*);
}

This code is used in section 388.

401. 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.

402. 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*.

```
static void scan_left_brace(void)      /* reads a mandatory left_brace */
{ ⟨Get the next non-blank non-relax non-call token 403⟩;
  if (cur_cmd ≠ left_brace) { 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);
  }
}
```

403. ⟨Get the next non-blank non-relax non-call token 403⟩ ≡

```
do { get_x_token();
} while (¬((cur_cmd ≠ spacer) ∧ (cur_cmd ≠ relax)))
```

This code is used in sections 402, 525, 1077, 1083, 1150, 1159, 1210, 1225, and 1269.

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

```
static void scan_optional_equals(void)
{ ⟨Get the next non-blank non-call token 405⟩;
  if (cur_tok ≠ other_token + '=') back_input();
}
```

405. ⟨Get the next non-blank non-call token 405⟩ ≡

```
do { get_x_token();
} while (¬(cur_cmd ≠ spacer))
```

This code is used in sections 404, 440, 454, 502, 576, 1044, 1466, and 1467.

406. 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.).

```
static bool scan_keyword(char *s)    /* look for a given string */
{ pointer p;    /* tail of the backup list */
  pointer q;    /* new node being added to the token list via store_new_token */
  p = backup_head;
  link(p) = null;
  while (*s ≠ 0) { get_x_token();    /* recursion is possible here */
    if ((cur_cs ≡ 0) ∧
          ((cur_chr ≡ so(*s)) ∨ (cur_chr ≡ so(*s) − 'a' + 'A'))) { store_new_token(cur_tok);
      incr(s);
    }
    else if ((cur_cmd ≠ spacer) ∨ (p ≠ backup_head)) { back_input();
      if (p ≠ backup_head) back_list(link(backup_head));
      return false;
    }
  }
  flush_list(link(backup_head));
  return true;
}
```

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

```
static void mu_error(void)
{ print_err("Incompatible␣glue␣units");
  help1("I'm␣going␣to␣assume␣that␣1mu=1pt␣when␣they're␣mixed.");
  error ( ) ;
}
```

408. 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.

```
static void scan_int(void);    /* scans an integer value */
⟨ Declare procedures that scan restricted classes of integers 432 ⟩
⟨ Declare ε-TEX procedures for scanning 1412 ⟩
⟨ Declare procedures that scan font-related stuff 576 ⟩
```

409. 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 $level \equiv tok\_val$.

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 ⟩ +≡
static int *cur_val*; /* value returned by numeric scanners */
static int *cur_val_level*; /* the "level" of this value */

410. The hash table is initialized with '`\count`', '`\dimen`', '`\skip`', and '`\muskip`' all having *internal_register* as their command code; they are distinguished by the *chr_code*, which is either *int_val*, *dimen_val*, *glue_val*, or *mu_val* more than *mem_bot* (dynamic variable-size nodes cannot have these values)

⟨ Put each of TeX's primitives into the hash table 225 ⟩ +≡
primitive("count", *internal_register*, *mem_bot* + *int_val*);
primitive("dimen", *internal_register*, *mem_bot* + *dimen_val*);
primitive("skip", *internal_register*, *mem_bot* + *glue_val*);
primitive("muskip", *internal_register*, *mem_bot* + *mu_val*);

411. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *internal_register*: ⟨ Cases of **register** for *print_cmd_chr* 1514 ⟩ **break**;

412. 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(A, B) { cur_val = A;
            cur_val_level = B; }

  static void scan_something_internal(small_number level, bool negative)
      /* fetch an internal parameter */
  { halfword m;    /* chr_code part of the operand token */
    pointer q, r;    /* general purpose indices */
    pointer tx;    /* effective tail node */
    four_quarters i;    /* character info */
    int p;    /* index into nest */

    m = cur_chr;
    switch (cur_cmd) {
    case def_code: ⟨Fetch a character code from some table 413⟩ break;
    case toks_register: case assign_toks: case def_family: case set_font: case def_font:
      ⟨Fetch a token list or font identifier, provided that level = tok_val 414⟩ break;
    case assign_int: scanned_result(eqtb[m].i, int_val) break;
    case assign_dimen: scanned_result(eqtb[m].sc, dimen_val) break;
    case assign_glue: scanned_result(equiv(m), glue_val) break;
    case assign_mu_glue: scanned_result(equiv(m), mu_val) break;
    case set_aux: ⟨Fetch the space_factor or the prev_depth 417⟩ break;
    case set_prev_graf: ⟨Fetch the prev_graf 421⟩ break;
    case set_page_int: ⟨Fetch the dead_cycles or the insert_penalties 418⟩ break;
    case set_page_dimen: ⟨Fetch something on the page_so_far 420⟩ break;
    case set_shape: ⟨Fetch the par_shape size 422⟩ break;
    case set_box_dimen: ⟨Fetch a box dimension 419⟩ break;
    case char_given: case math_given: scanned_result(cur_chr, int_val) break;
    case assign_font_dimen: ⟨Fetch a font dimension 424⟩ break;
    case assign_font_int: ⟨Fetch a font integer 425⟩ break;
    case internal_register: ⟨Fetch a register 426⟩ break;
    case last_item: ⟨Fetch an item in the current node, if appropriate 423⟩ break;
    default: ⟨Complain that \the can't do this; give zero result 427⟩
    }
    while (cur_val_level > level) ⟨Convert cur_val to a lower level 428⟩;
    ⟨Fix the reference count, if any, and negate cur_val if negative 429⟩;
  }
```

413. ⟨Fetch a character code from some table 413⟩ ≡

```
  { scan_char_num();
    if (m ≡ math_code_base) scanned_result(ho(math_code(cur_val)), int_val)
    else if (m < math_code_base) scanned_result(equiv(m + cur_val), int_val)
    else scanned_result(eqtb[m + cur_val].i, int_val);
  }
```

This code is used in section 412.

414. ⟨ Fetch a token list or font identifier, provided that *level* = *tok_val* 414 ⟩ ≡

```
if (level ≠ tok_val) { 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);
}
else if (cur_cmd ≤ assign_toks) { if (cur_cmd < assign_toks)     /* cur_cmd ≡ toks_register */
    if (m ≡ mem_bot) { scan_register_num();
      if (cur_val < 256) cur_val = equiv(toks_base + cur_val);
      else { find_sa_element(tok_val, cur_val, false);
        if (cur_ptr ≡ null) cur_val = null;
        else cur_val = sa_ptr(cur_ptr);
      }
    }
    else cur_val = sa_ptr(m);
  else cur_val = equiv(m);
  cur_val_level = tok_val;
}
else { back_input();
  scan_font_ident();
  scanned_result(font_id_base + cur_val, ident_val);
}
```

This code is used in section 412.

415. Users refer to '\the\spacefactor' only in horizontal mode, and to '\the\prevdepth' only in vertical mode; so we put the associated mode in the modifier part of the *set_aux* command. The *set_page_int* command has modifier 0 or 1, for '\deadcycles' and '\insertpenalties', respectively. The *set_box_dimen* command is modified by either *width_offset*, *height_offset*, or *depth_offset*. And the *last_item* command is modified by either *int_val*, *dimen_val*, *glue_val*, *input_line_no_code*, or *badness_code*. ε-TEX inserts *last_node_type_code* after *glue_val* and adds the codes for its extensions: *eTeX_version_code*, ...

#define *last_node_type_code* $(glue\_val + 1)$ /* code for \lastnodetype */
#define *input_line_no_code* $(glue\_val + 2)$ /* code for \inputlineno */
#define *badness_code* $(input\_line\_no\_code + 1)$ /* code for \badness */

#define *eTeX_int* $(badness\_code + 1)$ /* first of ε-TEX codes for integers */
#define *eTeX_dim* $(eTeX\_int + 8)$ /* first of ε-TEX codes for dimensions */
#define *eTeX_glue* $(eTeX\_dim + 9)$ /* first of ε-TEX codes for glue */
#define *eTeX_mu* $(eTeX\_glue + 1)$ /* first of ε-TEX codes for muglue */
#define *eTeX_expr* $(eTeX\_mu + 1)$ /* first of ε-TEX codes for expressions */
#define *eTeX_last_last_item_cmd_mod* $(eTeX\_expr - int\_val + mu\_val)$ /* \muexpr */

⟨Put each of TEX's primitives into the hash table 225⟩ +≡
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*);

416. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 226⟩ +≡
case *set_aux*:
if $(chr\_code \equiv vmode)$ *print_esc*("prevdepth"); **else** *print_esc*("spacefactor");
break; **case** *set_page_int*: **if** $(chr\_code \equiv 0)$ *print_esc*("deadcycles")
⟨Cases of *set_page_int* for *print_cmd_chr* 1423⟩; **else** *print_esc*("insertpenalties"); **break**;
case *set_box_dimen*:
if $(chr\_code \equiv width\_offset)$ *print_esc*("wd");
else if $(chr\_code \equiv height\_offset)$ *print_esc*("ht");
else *print_esc*("dp"); **break**; **case** *last_item*: **switch** (*chr_code*) {
case *int_val*: *print_esc*("lastpenalty"); **break**;
case *dimen_val*: *print_esc*("lastkern"); **break**;
case *glue_val*: *print_esc*("lastskip"); **break**;
case *input_line_no_code*: *print_esc*("inputlineno"); **break**;
⟨Cases of *last_item* for *print_cmd_chr* 1380⟩
default: *print_esc*("badness");
} **break**;

417. ⟨ Fetch the *space_factor* or the *prev_depth* 417 ⟩ ≡

```
if (abs(mode) ≠ m) { 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) scanned_result(0, dimen_val)
    else scanned_result(0, int_val);
}
else if (m ≡ vmode) scanned_result(prev_depth, dimen_val)
else scanned_result(space_factor, int_val)
```

This code is used in section 412.

418. ⟨ Fetch the *dead_cycles* or the *insert_penalties* 418 ⟩ ≡

```
{ if (m ≡ 0) cur_val = dead_cycles
⟨ Cases for 'Fetch the dead_cycles or the insert_penalties' 1424 ⟩;
else cur_val = insert_penalties;
cur_val_level = int_val;
}
```

This code is used in section 412.

419. ⟨ Fetch a box dimension 419 ⟩ ≡

```
{ scan_register_num( );
    fetch_box(q);
    if (q ≡ null) cur_val = 0; else cur_val = mem[q + m].sc;
    cur_val_level = dimen_val;
}
```

This code is used in section 412.

420. 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* 420 ⟩ ≡

```
{ if ((page_contents ≡ empty) ∧ (¬output_active))
        if (m ≡ 0) cur_val = max_dimen; else cur_val = 0;
    else cur_val = page_so_far[m];
    cur_val_level = dimen_val;
}
```

This code is used in section 412.

421. ⟨ Fetch the *prev_graf* 421 ⟩ ≡

```
if (mode ≡ 0) scanned_result(0, int_val)   /* prev_graf ≡ 0 within \write */
else { nest[nest_ptr] = cur_list;
    p = nest_ptr;
    while (abs(nest[p].mode_field) ≠ vmode) decr(p);
    scanned_result(nest[p].pg_field, int_val);
}
```

This code is used in section 412.

422. ⟨ Fetch the *par_shape* size 422 ⟩ ≡
{ **if** $(m > par\_shape\_loc)$ ⟨ Fetch a penalties array element 1536 ⟩
else if $(par\_shape\_ptr \equiv null)$ $cur\_val = 0$;
else $cur\_val = info(par\_shape\_ptr)$;
$cur\_val\_level = int\_val$;
}

This code is used in section 412.

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

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

⟨ Fetch an item in the current node, if appropriate 423 ⟩ ≡
  if ($m > eTeX\_last\_last\_item\_cmd\_mod$)
    ⟨ Fetch a PRoTE item 1549 ⟩
  else if ($m \ge input\_line\_no\_code$)
    if ($m \ge eTeX\_glue$) ⟨ Process an expression and **return** 1462 ⟩
    else if ($m \ge eTeX\_dim$) { **switch** (m) {
      ⟨ Cases for fetching a dimension value 1401 ⟩
      } /∗ there are no other cases ∗/
      $cur\_val\_level = dimen\_val$;
    }
    else { **switch** (m) {
      case $input\_line\_no\_code$: $cur\_val = line$; **break**;
      case $badness\_code$: $cur\_val = last\_badness$; **break**;
        ⟨ Cases for fetching an integer value 1381 ⟩
      } /∗ there are no other cases ∗/
      $cur\_val\_level = int\_val$;
    }
  else { **if** ($cur\_chr \equiv glue\_val$) $cur\_val = zero\_glue$; **else** $cur\_val = 0$;
    $tx = tail$;
    if ($cur\_chr \equiv last\_node\_type\_code$) { $cur\_val\_level = int\_val$;
      if (($tx \equiv head$) ∨ ($mode \equiv 0$)) $cur\_val = -1$;
    }
    else $cur\_val\_level = cur\_chr$;
    if ($\neg is\_char\_node(tx) \wedge (mode \neq 0)$)
      switch (cur_chr) {
      case int_val:
        if ($type(tx) \equiv penalty\_node$) $cur\_val = penalty(tx)$; **break**;
      case $dimen\_val$:
        if ($type(tx) \equiv kern\_node$) $cur\_val = width(tx)$; **break**;
      case $glue\_val$:
        if ($type(tx) \equiv glue\_node$) { $cur\_val = glue\_ptr(tx)$;
          if ($subtype(tx) \equiv mu\_glue$) $cur\_val\_level = mu\_val$;
        } **break**;
      case $last\_node\_type\_code$:
        if ($type(tx) \le unset\_node$) $cur\_val = type(tx) + 1$;
        else $cur\_val = unset\_node + 2$;
      } /∗ there are no other cases ∗/
    else if (($mode \equiv vmode$) ∧ ($tx \equiv head$))
      switch (cur_chr) {
      case int_val: $cur\_val = last\_penalty$; **break**;
      case $dimen\_val$: $cur\_val = last\_kern$; **break**;
      case $glue\_val$:
        if ($last\_glue \neq max\_halfword$) $cur\_val = last\_glue$; **break**;
      case $last\_node\_type\_code$: $cur\_val = last\_node\_type$;
      } /∗ there are no other cases ∗/
  }

This code is used in section 412.

424. ⟨Fetch a font dimension 424⟩ ≡
{ *find_font_dimen*(*false*);
  font_info[*fmem_ptr*].*sc* = 0;
  scanned_result(*font_info*[*cur_val*].*sc*, *dimen_val*);
}
This code is used in section 412.

425. ⟨Fetch a font integer 425⟩ ≡
{ *scan_font_ident*( );
  if ($m \equiv 0$) *scanned_result*(*hyphen_char*[*cur_val*], *int_val*)
  else *scanned_result*(*skew_char*[*cur_val*], *int_val*);
}
This code is used in section 412.

426. ⟨Fetch a register 426⟩ ≡
{ **if** ((m < *mem_bot*) ∨ (m > *lo_mem_stat_max*)) { *cur_val_level* = *sa_type*(m);
    if (*cur_val_level* < *glue_val*) *cur_val* = *sa_int*(m);
    else *cur_val* = *sa_ptr*(m);
  }
  else { *scan_register_num*( );
    cur_val_level = m − *mem_bot*;
    if (*cur_val* > 255) { *find_sa_element*(*cur_val_level*, *cur_val*, *false*);
      if (*cur_ptr* ≡ *null*)
        if (*cur_val_level* < *glue_val*) *cur_val* = 0;
        else *cur_val* = *zero_glue*;
      else if (*cur_val_level* < *glue_val*) *cur_val* = *sa_int*(*cur_ptr*);
      else *cur_val* = *sa_ptr*(*cur_ptr*);
    }
    else
      switch (*cur_val_level*) {
      case *int_val*: *cur_val* = *count*(*cur_val*); **break**;
      case *dimen_val*: *cur_val* = *dimen*(*cur_val*); **break**;
      case *glue_val*: *cur_val* = *skip*(*cur_val*); **break**;
      case *mu_val*: *cur_val* = *mu_skip*(*cur_val*);
      } /∗ there are no other cases ∗/
  }
}
This code is used in section 412.

427. ⟨Complain that `\the` can't do this; give zero result 427⟩ ≡
{ *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*) *scanned_result*(0, *dimen_val*)
  else *scanned_result*(0, *int_val*);
}
This code is used in section 412.

428. 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 428⟩ ≡
  { **if** (*cur_val_level* ≡ *glue_val*) *cur_val* = *width*(*cur_val*);
    else if (*cur_val_level* ≡ *mu_val*) *mu_error*( );
    decr(*cur_val_level*);
  }

This code is used in section 412.

429. 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* 429⟩ ≡
  if (*negative*)
    if (*cur_val_level* ≥ *glue_val*) { *cur_val* = *new_spec*(*cur_val*);
      ⟨Negate all three glue components of *cur_val* 430⟩;
    }
    else *negate*(*cur_val*);
  else if ((*cur_val_level* ≥ *glue_val*) ∧ (*cur_val_level* ≤ *mu_val*)) *add_glue_ref*(*cur_val*)

This code is used in section 412.

430. ⟨Negate all three glue components of *cur_val* 430⟩ ≡
  { *negate*(*width*(*cur_val*));
    negate(*stretch*(*cur_val*));
    negate(*shrink*(*cur_val*));
  }

This code is used in sections 429 and 1462.

431. 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*.

432. ⟨Declare procedures that scan restricted classes of integers 432⟩ ≡
  static void *scan_eight_bit_int*(**void**)
  { *scan_int*( );
    if ((*cur_val* < 0) ∨ (*cur_val* > 255)) { *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;
    }
  }

See also sections 433, 434, 435, 436, and 1493.

This code is used in section 408.

433. ⟨ Declare procedures that scan restricted classes of integers 432 ⟩ +≡

```
static void scan_char_num(void)
{ scan_int();
  if ((cur_val < 0) ∨ (cur_val > 255)) { 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;
  }
}
```

434. 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 432 ⟩ +≡

```
static void scan_four_bit_int(void)
{ scan_int();
  if ((cur_val < 0) ∨ (cur_val > 15)) { 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;
  }
}
```

435. ⟨ Declare procedures that scan restricted classes of integers 432 ⟩ +≡

```
static void scan_fifteen_bit_int(void)
{ scan_int();
  if ((cur_val < 0) ∨ (cur_val > °77777)) { 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;
  }
}
```

436. ⟨ Declare procedures that scan restricted classes of integers 432 ⟩ +≡

```
static void scan_twenty_seven_bit_int(void)
{ scan_int();
  if ((cur_val < 0) ∨ (cur_val > °777777777)) { 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;
  }
}
```

437. 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⟩ +≡
static small_number *radix*; /* *scan_int* sets this to 8, 10, 16, or zero */

438. 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*;

439. 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.

static void *scan_int*(**void**) /* sets *cur_val* to an integer */
{ **bool** *negative*; /* should the answer be negated? */
  int *m*; /* 2^{31}/ *radix*, the threshold of danger */
  small_number *d*; /* the digit just scanned */
  bool *vacuous*; /* have no digits appeared? */
  bool *OK_so_far*; /* has an error message been issued? */
  radix = 0;
  OK_so_far = *true*;
  ⟨Get the next non-blank non-sign token; set *negative* appropriately 440⟩;
  if (*cur_tok* ≡ *alpha_token*) ⟨Scan an alphabetic character code into *cur_val* 441⟩
  else if ((*cur_cmd* ≥ *min_internal*) ∧ (*cur_cmd* ≤ *max_internal*))
    scan_something_internal(*int_val*, *false*);
  else ⟨Scan a numeric constant 443⟩;
  if (*negative*) *negate*(*cur_val*);
}

440. ⟨Get the next non-blank non-sign token; set *negative* appropriately 440⟩ ≡
negative = *false*;
do { ⟨Get the next non-blank non-call token 405⟩;
  if (*cur_tok* ≡ *other_token* + '-') { *negative* = ¬*negative*;
    cur_tok = *other_token* + '+';
  }
} **while** (¬(*cur_tok* ≠ *other_token* + '+'))

This code is used in sections 439, 447, and 460.

441. 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* 441⟩ ≡

```
{ get_token( );    /* suppress macro expansion */
  if (cur_tok < cs_token_flag) { cur_val = cur_chr;
    if (cur_cmd ≤ right_brace)
      if (cur_cmd ≡ right_brace) incr(align_state);
      else decr(align_state);
  }
  else if (cur_tok < cs_token_flag + single_base) cur_val = cur_tok − cs_token_flag − active_base;
  else cur_val = cur_tok − cs_token_flag − single_base;
  if (cur_val > 255) { 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( );
  }
  else ⟨Scan an optional space 442⟩;
}
```

This code is used in section 439.

442. ⟨Scan an optional space 442⟩ ≡

```
{ get_x_token( );
  if (cur_cmd ≠ spacer) back_input( );
}
```

This code is used in sections 441, 447, 454, and 1199.

443. ⟨Scan a numeric constant 443⟩ ≡

```
{ radix = 10;
  m = 214748364;
  if (cur_tok ≡ octal_token) { radix = 8;
    m = °2000000000;
    get_x_token( );
  }
  else if (cur_tok ≡ hex_token) { radix = 16;
    m = °1000000000;
    get_x_token( );
  }
  vacuous = true;
  cur_val = 0;
  ⟨Accumulate the constant until cur_tok is not a suitable digit 444⟩;
  if (vacuous) ⟨Express astonishment that no number was here 445⟩
  else if (cur_cmd ≠ spacer) back_input( );
}
```

This code is used in section 439.

444. **#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 444⟩ ≡

```
loop { if ((cur_tok < zero_token + radix) ∧ (cur_tok ≥ zero_token) ∧ (cur_tok ≤ zero_token + 9))
      d = cur_tok − zero_token;
    else if (radix ≡ 16)
      if ((cur_tok ≤ A_token + 5) ∧ (cur_tok ≥ A_token)) d = cur_tok − A_token + 10;
      else if ((cur_tok ≤ other_A_token + 5) ∧ (cur_tok ≥ other_A_token))
        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))) { if (OK_so_far) {
        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;
      }
    }
    else cur_val = cur_val ∗ radix + d;
    get_x_token( );
  }
  done:
```

This code is used in section 443.

445. ⟨Express astonishment that no number was here 445⟩ ≡

```
  { 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( );
  }
```

This code is used in section 443.

446. 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⟩ +≡

```
  static glue_ord cur_order; /* order of infinity found by scan_dimen */
```

447. 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 **int** to **scaled**.

```
#define scan_normal_dimen scan_dimen(false, false, false)

  static void scan_dimen(bool mu, bool inf, bool shortcut)    /* sets cur_val to a dimension */
  { bool negative;    /* should the answer be negated? */
    int f;    /* numerator of a fraction whose denominator is 2^16 */
    ⟨Local variables for dimension calculations 449⟩
    f = 0;
    arith_error = false;
    cur_order = normal;
    negative = false;
    if (¬shortcut) { ⟨Get the next non-blank non-sign token; set negative appropriately 440⟩;
      if ((cur_cmd ≥ min_internal) ∧ (cur_cmd ≤ max_internal))
        ⟨Fetch an internal dimension and goto attach_sign, or fetch an internal integer 448⟩
      else { back_input();
        if (cur_tok ≡ continental_point_token) cur_tok = point_token;
        if (cur_tok ≠ point_token) scan_int();
        else { radix = 10;
          cur_val = 0;
        }
        if (cur_tok ≡ continental_point_token) cur_tok = point_token;
        if ((radix ≡ 10) ∧ (cur_tok ≡ point_token)) ⟨Scan decimal fraction 451⟩;
      }
    }
    if (cur_val < 0)    /* in this case f ≡ 0 */
    { negative = ¬negative;
      negate(cur_val);
    }
    ⟨Scan units and set cur_val to x · (cur_val + f/2^16), where there are x sp per unit; goto attach_sign
        if the units are internal 452⟩;
    ⟨Scan an optional space 442⟩;
  attach_sign:
    if (arith_error ∨ (abs(cur_val) ≥ °10000000000)) ⟨Report that this dimension is out of range 459⟩;
    if (negative) negate(cur_val);
  }
```

448. ⟨Fetch an internal dimension and **goto** *attach_sign*, or fetch an internal integer 448⟩ ≡

```
  if (mu) { scan_something_internal(mu_val, false);
    ⟨Coerce glue to a dimension 450⟩;
    if (cur_val_level ≡ mu_val) goto attach_sign;
    if (cur_val_level ≠ int_val) mu_error();
  }
  else { scan_something_internal(dimen_val, false);
    if (cur_val_level ≡ dimen_val) goto attach_sign;
  }
```

This code is used in section 447.

449. ⟨ Local variables for dimension calculations 449 ⟩ ≡

```
int num, denom;   /* conversion ratio for the scanned units */
int k, kk;   /* number of digits in a decimal fraction */
pointer p, q;   /* top of decimal digit stack */
scaled v;   /* an internal dimension */
int save_cur_val;   /* temporary storage of cur_val */
```

This code is used in section 447.

450. 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 450 ⟩ ≡

```
if (cur_val_level ≥ glue_val) { v = width(cur_val);
  delete_glue_ref(cur_val);
  cur_val = v;
}
```

This code is used in sections 448 and 454.

451. 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 451 ⟩ ≡

```
{ k = 0;
  p = null;
  get_token();   /* point_token is being re-scanned */
  loop { get_x_token();
    if ((cur_tok > zero_token + 9) ∨ (cur_tok < zero_token)) goto done1;
    if (k < 17)   /* digits for k ≥ 17 cannot affect the result */
    { q = get_avail();
      link(q) = p;
      info(q) = cur_tok − zero_token;
      p = q;
      incr(k);
    }
  }
done1:
  for (kk = k; kk ≥ 1; kk−−) { dig[kk − 1] = info(p);
    q = p;
    p = link(p);
    free_avail(q);
  }
  f = round_decimals(k);
  if (cur_cmd ≠ spacer) back_input();
}
```

This code is used in section 447.

452. 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 452 ⟩ ≡
  if (*inf*) ⟨ Scan for `fil` units; **goto** *attach_fraction* if found 453 ⟩;
  ⟨ Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 454 ⟩;
  if (*mu*) ⟨ Scan for `mu` units and **goto** *attach_fraction* 455 ⟩;
  if (*scan_keyword*(`"true"`)) ⟨ Adjust for the magnification ratio 456 ⟩;
  if (*scan_keyword*(`"pt"`)) **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 457 ⟩;
attach_fraction:
  if ($cur\_val \geq °40000$) *arith_error* = *true*;
  else *cur_val* = *cur_val* ∗ *unity* + *f*;
  done:

This code is used in section 447.

453. 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 453 ⟩ ≡
  if (*scan_keyword*(`"fil"`)) { *cur_order* = *fil*;
    while (*scan_keyword*(`"l"`)) { **if** (*cur_order* ≡ *filll*) { *print_err*(`"Illegal␣unit␣of␣measure␣("`);
        print(`"replaced␣by␣filll)"`);
        help1(`"I␣dddon't␣go␣any␣higher␣than␣filll."`);
        error ( ) ;
      }
      else *incr*(*cur_order*);
    }
    goto *attach_fraction*;
  }

This code is used in section 452.

454. ⟨Scan for units that are internal dimensions; **goto** *attach_sign* with *cur_val* set if found 454⟩ ≡

```
save_cur_val = cur_val;
⟨Get the next non-blank non-call token 405⟩;
if ((cur_cmd < min_internal) ∨ (cur_cmd > max_internal)) back_input();
else { if (mu) { scan_something_internal(mu_val, false);
      ⟨Coerce glue to a dimension 450⟩;
      if (cur_val_level ≠ mu_val) mu_error();
    }
    else scan_something_internal(dimen_val, false);
    v = cur_val;
    goto found;
  }
  if (mu) goto not_found;
  if (scan_keyword("em")) v = (⟨The em width for cur_font 557⟩);
  else if (scan_keyword("ex")) v = (⟨The x-height for cur_font 558⟩);
  else goto not_found;
  ⟨Scan an optional space 442⟩;
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 452.

455. ⟨Scan for mu units and **goto** *attach_fraction* 455⟩ ≡

```
if (scan_keyword("mu")) goto attach_fraction;
else { 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;
}
```

This code is used in section 452.

456. ⟨Adjust for the magnification ratio 456⟩ ≡

```
{ prepare_mag();
  if (mag ≠ 1000) { cur_val = xn_over_d(cur_val, 1000, mag);
    f = (1000 * f + °200000 * rem)/mag;
    cur_val = cur_val + (f/°200000);
    f = f % °200000;
  }
}
```

This code is used in section 452.

457. 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(A, B)  { num = A;
            denom = B;
          }
```

⟨Scan for all other units and adjust *cur_val* and *f* accordingly; **goto** *done* in the case of scaled points 457⟩ ≡

```
if (scan_keyword("in")) set_conversion(7227, 100)
else if (scan_keyword("pc")) set_conversion(12, 1)
else if (scan_keyword("cm")) set_conversion(7227, 254)
else if (scan_keyword("mm")) set_conversion(7227, 2540)
else if (scan_keyword("bp")) set_conversion(7227, 7200)
else if (scan_keyword("dd")) set_conversion(1238, 1157)
else if (scan_keyword("cc")) set_conversion(14856, 1157)
else if (scan_keyword("sp")) goto done;
else ⟨Complain about unknown unit and goto done2 458⟩;
cur_val = xn_over_d(cur_val, num, denom);
f = (num * f + °200000 * rem)/denom;
cur_val = cur_val + (f/°200000);
f = f % °200000; done2:
```

This code is used in section 452.

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

```
{ 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;
}
```

This code is used in section 457.

459. ⟨Report that this dimension is out of range 459⟩ ≡

```
{ 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;
}
```

This code is used in section 447.

460. 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.

```
static void scan_glue(small_number level)   /* sets cur_val to a glue spec pointer */
{ bool negative;   /* should the answer be negated? */
  pointer q;   /* new glue specification */
  bool mu;   /* does level ≡ mu_val? */
  mu = (level ≡ mu_val);
  ⟨Get the next non-blank non-sign token; set negative appropriately 440⟩;
  if ((cur_cmd ≥ min_internal) ∧ (cur_cmd ≤ max_internal)) {
    scan_something_internal(level, negative);
    if (cur_val_level ≥ glue_val) { if (cur_val_level ≠ level) mu_error();
      return;
    }
    if (cur_val_level ≡ int_val) scan_dimen(mu, false, true);
    else if (level ≡ mu_val) mu_error();
  }
  else { back_input();
    scan_dimen(mu, false, false);
    if (negative) negate(cur_val);
  }
  ⟨Create a new glue specification whose width is cur_val; scan for its stretch and shrink
      components 461⟩;
}
⟨Declare procedures needed for expressions 1464⟩
```

461. ⟨Create a new glue specification whose width is *cur_val*; scan for its stretch and shrink components 461⟩ ≡

```
q = new_spec(zero_glue);
width(q) = cur_val;
if (scan_keyword("plus")) { scan_dimen(mu, true, false);
  stretch(q) = cur_val;
  stretch_order(q) = cur_order;
}
if (scan_keyword("minus")) { scan_dimen(mu, true, false);
  shrink(q) = cur_val;
  shrink_order(q) = cur_order;
}
cur_val = q
```

This code is used in section 460.

462. 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 */
  static pointer scan_rule_spec(void)
  { pointer q;    /* the rule node being created */
    q = new_rule();    /* width, depth, and height all equal null_flag now */
    if (cur_cmd ≡ vrule) width(q) = default_rule;
    else { height(q) = default_rule;
      depth(q) = 0;
    }
  reswitch:
    if (scan_keyword("width")) { scan_normal_dimen;
      width(q) = cur_val;
      goto reswitch;
    }
    if (scan_keyword("height")) { scan_normal_dimen;
      height(q) = cur_val;
      goto reswitch;
    }
    if (scan_keyword("depth")) { scan_normal_dimen;
      depth(q) = cur_val;
      goto reswitch;
    }
    return q;
  }
```

463. 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 $p \equiv temp\_head$, the list is empty.)

⟨ Declare ε-TEX procedures for token lists 1413 ⟩

```
static pointer str_toks(pool_pointer b)    /* converts str_pool[b .. pool_ptr − 1] to a token list */
{ pointer p;    /* tail of the token list */
  pointer q;    /* new node being added to the token list via store_new_token */
  halfword t;    /* token being appended */
  pool_pointer k;    /* index into str_pool */

  str_room(1);
  p = temp_head;
  link(p) = null;
  k = b;
  while (k < pool_ptr) { t = so(str_pool[k]);
    if (t ≡ '␣') t = space_token;
    else t = other_token + t;
    fast_store_new_token(t);
    incr(k);
  }
  pool_ptr = b;
  return p;
}
```

464. 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'.

```
static pointer the_toks(void)
{ int old_setting;      /* holds selector setting */
  pointer p, q, r;      /* used for copying a token list */
  pool_pointer b;       /* base of temporary string */
  small_number c;       /* value of cur_chr */
  ⟨Handle \unexpanded or \detokenize and return 1418⟩;
  get_x_token();
  scan_something_internal(tok_val, false);
  if (cur_val_level ≥ ident_val) ⟨Copy the token list 465⟩
  else { old_setting = selector;
    selector = new_string;
    b = pool_ptr;
    switch (cur_val_level) {
    case int_val: print_int(cur_val); break;
    case dimen_val:
      { print_scaled(cur_val);
        print("pt");
      } break;
    case glue_val:
      { print_spec(cur_val, "pt");
        delete_glue_ref(cur_val);
      } break;
    case mu_val:
      { print_spec(cur_val, "mu");
        delete_glue_ref(cur_val);
      }
    }     /* there are no other cases */
    selector = old_setting;
    return str_toks(b);
  }
}
```

465. ⟨Copy the token list 465⟩ ≡

```
{ p = temp_head;
  link(p) = null;
  if (cur_val_level ≡ ident_val) store_new_token(cs_token_flag + cur_val)
  else if (cur_val ≠ null) { r = link(cur_val);     /* do not copy the reference count */
    while (r ≠ null) { fast_store_new_token(info(r));
      r = link(r);
    }
  }
  return p;
}
```

This code is used in section 464.

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

```
static void ins_the_toks(void)
{ link(garbage) = the_toks();
  ins_list(link(temp_head));
}
```

467. 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 */
#define etex_convert_base (job_name_code + 1)  /* base for ε-TeX's command codes */
#define eTeX_revision_code etex_convert_base  /* command code for \eTeXrevision */
#define etex_convert_codes (etex_convert_base + 1)  /* end of ε-TeX's command codes */
#define eTeX_last_convert_cmd_mod etex_convert_codes
```

⟨ Put each of TeX's primitives into the hash table 225 ⟩ +≡

```
  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);
```

468. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case convert:
  switch (chr_code) {
  case number_code: print_esc("number"); break;
  case roman_numeral_code: print_esc("romannumeral"); break;
  case string_code: print_esc("string"); break;
  case meaning_code: print_esc("meaning"); break;
  case font_name_code: print_esc("fontname"); break;
  case job_name_code: print_esc("jobname"); break;
  case eTeX_revision_code: print_esc("eTeXrevision"); break;
    ⟨ Cases of convert for print_cmd_chr 1556 ⟩
  } break;
```

469. 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`'.

```
static void conv_toks(void)
{ int old_setting;   /* holds selector setting */
  int c;   /* desired type of conversion */
  small_number save_scanner_status;   /* scanner_status upon entry */
  pool_pointer b;   /* base of temporary string */
  int i, k, l;   /* general purpose index */
  pool_pointer m, n;   /* general purpose pool pointer */
  bool r;   /* general purpose refraction i.e. changing the way */
  str_number s, t;   /* general purpose; de dicto */
  c = cur_chr;
  ⟨Scan the argument for command c 470⟩;
  old_setting = selector;
  selector = new_string;
  b = pool_ptr;
  ⟨Print the result of command c 471⟩;
  selector = old_setting;
  link(garbage) = str_toks(b);
  ins_list(link(temp_head));
}
```

470. ⟨Scan the argument for command *c* 470⟩ ≡

```
switch (c) {
case number_code: case roman_numeral_code: scan_int(); break;
case string_code: case meaning_code:
  { save_scanner_status = scanner_status;
    scanner_status = normal;
    get_token();
    scanner_status = save_scanner_status;
  } break;
case font_name_code: scan_font_ident(); break;
case job_name_code:
  if (job_name ≡ 0) open_log_file(); break;
case eTeX_revision_code: do_nothing; break;
  ⟨Cases of 'Scan the argument for command c' 1557⟩
}   /* there are no other cases */
```

This code is used in section 469.

471. ⟨ Print the result of command c 471 ⟩ ≡

```
switch (c) {
case number_code: print_int(cur_val); break;
case roman_numeral_code: print_roman_int(cur_val); break;
case string_code:
  if (cur_cs ≠ 0) sprint_cs(cur_cs);
  else print_char(cur_chr); break;
case meaning_code: print_meaning( ); break;
case font_name_code:
  { printn(font_name[cur_val]);
    if (font_size[cur_val] ≠ font_dsize[cur_val]) { print("␣at␣");
      print_scaled(font_size[cur_val]);
      print("pt");
    }
  } break;
case eTeX_revision_code: print(eTeX_revision); break;
case job_name_code: printn(job_name); break;
  ⟨ Cases of 'Print the result of command c' 1558 ⟩
}    /* there are no other cases */
```

This code is used in section 469.

472. 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.

```
static pointer scan_toks(bool macro_def, bool xpand)
{ halfword t;   /* token representing the highest parameter number */
  halfword s;   /* saved token */
  pointer p;   /* tail of the token list being built */
  pointer q;   /* new node being added to the token list via store_new_token */
  halfword unbalance;   /* number of unmatched left braces */
  halfword hash_brace;   /* possible '#{' token */

  if (macro_def) 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) ⟨Scan and build the parameter part of the macro definition 473⟩
  else scan_left_brace();   /* remove the compulsory left brace */
  ⟨Scan and build the body of the token list; goto found when finished 476⟩;
found: scanner_status = normal;
  if (hash_brace ≠ 0) store_new_token(hash_brace);
  return p;
}
⟨Declare PRoTE procedures for token lists 1561⟩
```

473.
```
⟨Scan and build the parameter part of the macro definition 473⟩ ≡
{ loop { resume: get_token();   /* set cur_cmd, cur_chr, cur_tok */
    if (cur_tok < right_brace_limit) goto done1;
    if (cur_cmd ≡ mac_param)
      ⟨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 475⟩;
    store_new_token(cur_tok);
  }
done1: store_new_token(end_match_token);
  if (cur_cmd ≡ right_brace) ⟨Express shock at the missing left brace; goto found 474⟩;
done: ;
}
```
This code is used in section 472.

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

```
{ 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;
}
```

This code is used in section 473.

475. ⟨ 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* 475 ⟩ ≡

```
{ s = match_token + cur_chr;
  get_token();
  if (cur_tok < left_brace_limit) { hash_brace = cur_tok;
    store_new_token(cur_tok);
    store_new_token(end_match_token);
    goto done;
  }
  if (t ≡ zero_token + 9) { 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 resume;
  }
  else { incr(t);
    if (cur_tok ≠ t) { 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();
    }
    cur_tok = s;
  }
}
```

This code is used in section 473.

476. ⟨ Scan and build the body of the token list; **goto** *found* when finished 476 ⟩ ≡

```
unbalance = 1;
loop { if (xpand) ⟨ Expand the next part of the input 477 ⟩
  else get_token();
  if (cur_tok < right_brace_limit)
    if (cur_cmd < right_brace) incr(unbalance);
    else { decr(unbalance);
      if (unbalance ≡ 0) goto found;
    }
  else if (cur_cmd ≡ mac_param)
    if (macro_def) ⟨ Look for parameter number or ## 478 ⟩;
  store_new_token(cur_tok);
}
```

This code is used in section 472.

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

⟨Expand the next part of the input 477⟩ ≡
{ **loop** { *get_next*();
  if (*cur_cmd* ≥ *call*)
    if (*info*(*link*(*cur_chr*)) ≡ *protected_token*) { *cur_cmd* = *relax*;
      cur_chr = *no_expand_flag*;
    }
  if (*cur_cmd* ≤ *max_command*) **goto** *done2*;
  if (*cur_cmd* ≠ *the*) *expand*();
  else { *q* = *the_toks*();
    if (*link*(*temp_head*) ≠ *null*) { *link*(*p*) = *link*(*temp_head*);
      p = *q*;
    }
  }
}
done2: *x_token*();
}

This code is used in section 476.

478. ⟨Look for parameter number or `##` 478⟩ ≡
{ *s* = *cur_tok*;
  if (*xpand*) *get_x_token*();
  else *get_token*();
  if (*cur_cmd* ≠ *mac_param*)
    if ((*cur_tok* ≤ *zero_token*) ∨ (*cur_tok* > *t*)) {
      print_err("Illegal␣parameter␣number␣in␣definition␣of␣");
      sprint_cs(*warning_index*);
      help3("You␣meant␣to␣type␣##␣instead␣of␣#,␣right?",
      "Or␣maybe␣a␣}␣was␣forgotten␣somewhere␣earlier,␣and␣things",
      "are␣all␣screwed␣up?␣I'm␣going␣to␣assume␣that␣you␣meant␣##.");
      back_error();
      cur_tok = *s*;
    }
  else *cur_tok* = *out_param_token* − '0' + *cur_chr*;
}

This code is used in section 476.

479. 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⟩ +≡
  static alpha_file *read_file*[16]; /∗ used for `\read` ∗/
  static int8_t *read_open*[17]; /∗ state of *read_file*[*n*] ∗/

480. ⟨Set initial values of key variables 21⟩ +≡
  for (*k* = 0; *k* ≤ 16; *k*++) *read_open*[*k*] = *closed*;

481. 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.

```
static void read_toks(int n, pointer r, halfword j)
{ pointer p;      /* tail of the token list */
  pointer q;      /* new node being added to the token list via store_new_token */
  int s;      /* saved value of align_state */
  small_number m;      /* stream number */
  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)) m = 16; else m = n;
  s = align_state;
  align_state = 1000000;      /* disable tab marks, etc. */
  do { ⟨Input and store tokens from the next line of the file 482⟩;
  } while (¬(align_state ≡ 1000000));
  cur_val = def_ref;
  scanner_status = normal;
  align_state = s;
}
```

482. ⟨Input and store tokens from the next line of the file 482⟩ ≡

```
begin_file_reading();
name = m + 1;
if (read_open[m] ≡ closed) ⟨Input for \read from the terminal 483⟩;
else if (read_open[m] ≡ just_open) ⟨Input the first line of read_file[m] 484⟩
else ⟨Input the next line of read_file[m] 485⟩;
limit = last;
if (end_line_char_inactive) decr(limit);
else buffer[limit] = end_line_char;
first = limit + 1;
loc = start;
state = new_line;
⟨Handle \readline and goto done 1443⟩;
loop { get_token();
  if (cur_tok ≡ 0) goto done;      /* cur_cmd ≡ cur_chr ≡ 0 will occur at the end of the line */
  if (align_state < 1000000)      /* unmatched '}' aborts the line */
  { do { get_token();
    } while (¬(cur_tok ≡ 0));
    align_state = 1000000;
    goto done;
  }
  store_new_token(cur_tok);
}
done: end_file_reading()
```

This code is used in section 481.

483. 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 483⟩ ≡
```
if (interaction > nonstop_mode)
  if (n < 0) prompt_input("")
  else { wake_up_terminal;
    print_ln();
    sprint_cs(r);
    prompt_input("=");
    n = −1;
  }
else fatal_error("***␣(cannot␣\\read␣from␣terminal␣in␣nonstop␣modes)")
```
This code is used in section 482.

484. 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*] 484⟩ ≡
```
if (input_ln(&read_file[m], false)) read_open[m] = normal;
else { a_close(&read_file[m]);
  read_open[m] = closed;
}
```
This code is used in section 482.

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

⟨Input the next line of *read_file*[*m*] 485⟩ ≡
```
{ if (¬input_ln(&read_file[m], true)) { a_close(&read_file[m]);
    read_open[m] = closed;
    if (align_state ≠ 1000000) { runaway();
      print_err("File␣ended␣within␣");
      print_esc("read");
      help1("This␣\\read␣has␣unbalanced␣braces.");
      align_state = 1000000;
      limit = 0;
      error();
    }
  }
}
```
This code is used in section 482.

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

```
#define unless_code 32   /* amount added for '\unless' prefix */
#define if_char_code 0   /* '\if' */
#define if_cat_code 1   /* '\ifcat' */
#define if_int_code 2   /* '\ifnum' */
#define if_dim_code 3   /* '\ifdim' */
#define if_odd_code 4   /* '\ifodd' */
#define if_vmode_code 5   /* '\ifvmode' */
#define if_hmode_code 6   /* '\ifhmode' */
#define if_mmode_code 7   /* '\ifmmode' */
#define if_inner_code 8   /* '\ifinner' */
#define if_void_code 9   /* '\ifvoid' */
#define if_hbox_code 10   /* '\ifhbox' */
#define if_vbox_code 11   /* '\ifvbox' */
#define ifx_code 12   /* '\ifx' */
#define if_eof_code 13   /* '\ifeof' */
#define if_true_code 14   /* '\iftrue' */
#define if_false_code 15   /* '\iffalse' */
#define if_case_code 16   /* '\ifcase' */
```

⟨Put each of TEX's primitives into the hash table 225⟩ +≡

```
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);
```

487. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *if_test*: { **if** (*chr_code* ≥ *unless_code*) *print_esc*("unless");
switch (*chr_code* % *unless_code*) {
case *if_cat_code*: *print_esc*("ifcat"); **break**;
case *if_int_code*: *print_esc*("ifnum"); **break**;
case *if_dim_code*: *print_esc*("ifdim"); **break**;
case *if_odd_code*: *print_esc*("ifodd"); **break**;
case *if_vmode_code*: *print_esc*("ifvmode"); **break**;
case *if_hmode_code*: *print_esc*("ifhmode"); **break**;
case *if_mmode_code*: *print_esc*("ifmmode"); **break**;
case *if_inner_code*: *print_esc*("ifinner"); **break**;
case *if_void_code*: *print_esc*("ifvoid"); **break**;
case *if_hbox_code*: *print_esc*("ifhbox"); **break**;
case *if_vbox_code*: *print_esc*("ifvbox"); **break**;
case *ifx_code*: *print_esc*("ifx"); **break**;
case *if_eof_code*: *print_esc*("ifeof"); **break**;
case *if_true_code*: *print_esc*("iftrue"); **break**;
case *if_false_code*: *print_esc*("iffalse"); **break**;
case *if_case_code*: *print_esc*("ifcase"); **break**;
⟨ Cases of *if_test* for *print_cmd_chr* 1446 ⟩
default: *print_esc*("if");
}
} **break**;

488. 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*(*A*) *mem*[*A* + 1].*i*
#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 ⟩ +≡
static pointer *cond_ptr*; /∗ top of the condition stack ∗/
static int *if_limit*; /∗ upper bound on *fi_or_else* codes ∗/
static small_number *cur_if*; /∗ type of conditional being worked on ∗/
static int *if_line*; /∗ line where that conditional began ∗/

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

490. ⟨ Put each of TeX's primitives into the hash table 225 ⟩ +≡

```
primitive("fi", fi_or_else, fi_code);
text(frozen_fi) = text(cur_val);
eqtb[frozen_fi] = eqtb[cur_val];
primitive("or", fi_or_else, or_code);
primitive("else", fi_or_else, else_code);
```

491. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case fi_or_else:
  if (chr_code ≡ fi_code) print_esc("fi");
  else if (chr_code ≡ or_code) print_esc("or");
  else print_esc("else"); break;
```

492. When we skip conditional text, we keep track of the line number where skipping began, for use in error messages.

⟨ Global variables 13 ⟩ +≡

```
  static int skip_line;    /* skipping began here */
```

493. 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).

```
  static void pass_text(void)
  { int l;    /* level of \if ... \fi nesting */
    small_number save_scanner_status;    /* scanner_status upon entry */

    save_scanner_status = scanner_status;
    scanner_status = skipping;
    l = 0;
    skip_line = line;
    loop { get_next();
      if (cur_cmd ≡ fi_or_else) { if (l ≡ 0) goto done;
        if (cur_chr ≡ fi_code) decr(l);
      }
      else if (cur_cmd ≡ if_test) incr(l);
    }
  done: scanner_status = save_scanner_status;
    if (tracing_ifs > 0) show_cur_cmd_chr();
  }
```

494. 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 494⟩ ≡
```
{ 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;
}
```
This code is used in section 497.

495. ⟨Pop the condition stack 495⟩ ≡
```
{ if (if_stack[in_open] ≡ cond_ptr) if_warning();
      /* conditionals possibly not properly nested with files */
  p = cond_ptr;
  if_line = if_line_field(p);
  cur_if = subtype(p);
  if_limit = type(p);
  cond_ptr = link(p);
  free_node(p, if_node_size);
}
```
This code is used in sections 497, 499, 508, and 509.

496. Here's a procedure that changes the *if_limit* code corresponding to a given value of *cond_ptr*.

```
static void change_if_limit(small_number l, pointer p)
{ pointer q;
  if (p ≡ cond_ptr) if_limit = l;    /* that's the easy case */
  else { q = cond_ptr;
    loop { if (q ≡ null) confusion("if");
      if (link(q) ≡ p) { type(q) = l;
        return;
      }
      q = link(q);
    }
  }
}
```

497. 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.

```
static void conditional(void)
{ bool b;   /* is the condition true? */
  int r;   /* relation to be evaluated */
  int m, n;   /* to be tested against the second operand */
  pointer p, q;   /* for traversing token lists in \ifx tests */
  small_number save_scanner_status;   /* scanner_status upon entry */
  pointer save_cond_ptr;   /* cond_ptr corresponding to this conditional */
  small_number this_if;   /* type of this conditional */
  bool is_unless;   /* was this if preceded by '\unless' ? */
  if (tracing_ifs > 0)
    if (tracing_commands ≤ 1) show_cur_cmd_chr();
  ⟨Push the condition stack 494⟩; save_cond_ptr = cond_ptr;
  is_unless = (cur_chr ≥ unless_code);
  this_if = cur_chr % unless_code;
  ⟨Either process \ifcase or set b to the value of a boolean condition 500⟩;
  if (is_unless) b = ¬b;
  if (tracing_commands > 1) ⟨Display the value of b 501⟩;
  if (b) { change_if_limit(else_code, save_cond_ptr);
    return;   /* wait for \else or \fi */
  }
  ⟨Skip to \else or \fi, then goto common_ending 499⟩;
common_ending:
  if (cur_chr ≡ fi_code) ⟨Pop the condition stack 495⟩
  else if_limit = fi_code;   /* wait for \fi */
}
```

498. 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.

499. ⟨Skip to \else or \fi, then **goto** *common_ending* 499⟩ ≡

```
loop { pass_text();
  if (cond_ptr ≡ save_cond_ptr) { if (cur_chr ≠ or_code) goto common_ending;
    print_err("Extra␣");
    print_esc("or");
    help1("I'm␣ignoring␣this;␣it␣doesn't␣match␣any␣\\if.");
    error( );
  }
  else if (cur_chr ≡ fi_code) ⟨Pop the condition stack 495⟩;
}
```

This code is used in section 497.

500. ⟨ Either process `\ifcase` or set b to the value of a boolean condition 500 ⟩ ≡
switch (*this_if*) {
case *if_char_code*: **case** *if_cat_code*: ⟨ Test if two characters match 505 ⟩ **break**;
case *if_int_code*: **case** *if_dim_code*: ⟨ Test relation between integers or dimensions 502 ⟩ **break**;
case *if_odd_code*: ⟨ Test if an integer is odd 503 ⟩ **break**;
case *if_vmode_code*: $b = (abs(mode) \equiv vmode)$; **break**;
case *if_hmode_code*: $b = (abs(mode) \equiv hmode)$; **break**;
case *if_mmode_code*: $b = (abs(mode) \equiv mmode)$; **break**;
case *if_inner_code*: $b = (mode < 0)$; **break**;
case *if_void_code*: **case** *if_hbox_code*: **case** *if_vbox_code*: ⟨ Test box register status 504 ⟩ **break**;
case *ifx_code*: ⟨ Test if two tokens match 506 ⟩ **break**;
case *if_eof_code*:
{ *scan_four_bit_int*( );
$b = (read\_open[cur\_val] \equiv closed)$;
} **break**;
case *if_true_code*: $b = true$; **break**;
case *if_false_code*: $b = false$; **break**;
⟨ Cases for *conditional* 1448 ⟩
case *if_case_code*: ⟨ Select the appropriate case and **return** or **goto** *common_ending* 508 ⟩;
} /∗ there are no other cases ∗/

This code is used in section 497.

501. ⟨ Display the value of b 501 ⟩ ≡
{ *begin_diagnostic*( );
if (b) *print*(`"{true}"`); **else** *print*(`"{false}"`);
end_diagnostic(*false*);
}

This code is used in section 497.

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

⟨ Test relation between integers or dimensions 502 ⟩ ≡
{ **if** (*this_if* ≡ *if_int_code*) *scan_int*( ); **else** *scan_normal_dimen*;
$n = cur\_val$;
⟨ Get the next non-blank non-call token 405 ⟩;
if $((cur\_tok \ge other\_token + \texttt{'<'}) \land (cur\_tok \le other\_token + \texttt{'>'}))$ $r = cur\_tok - other\_token$;
else { *print_err*(`"Missing␣=␣inserted␣for␣"`);
print_cmd_chr(*if_test*, *this_if*);
help1(`"I␣was␣expecting␣to␣see␣'<',␣'=',␣or␣'>'.␣Didn't."`);
back_error( );
$r = \texttt{'='}$;
}
if (*this_if* ≡ *if_int_code*) *scan_int*( ); **else** *scan_normal_dimen*;
switch (r) {
case `'<'`: $b = (n < cur\_val)$; **break**;
case `'='`: $b = (n \equiv cur\_val)$; **break**;
case `'>'`: $b = (n > cur\_val)$;
}
}

This code is used in section 500.

503. ⟨ Test if an integer is odd 503 ⟩ ≡

```
{ scan_int();
  b = odd(cur_val);
}
```

This code is used in section 500.

504. ⟨ Test box register status 504 ⟩ ≡

```
{ scan_register_num();
  fetch_box(p);
  if (this_if ≡ if_void_code) b = (p ≡ null);
  else if (p ≡ null) b = false;
  else if (this_if ≡ if_hbox_code) b = (type(p) ≡ hlist_node);
  else b = (type(p) ≡ vlist_node);
}
```

This code is used in section 500.

505. 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
        { get_x_token();
          if (cur_cmd ≡ relax)
            if (cur_chr ≡ no_expand_flag) { cur_cmd = active_char;
              cur_chr = cur_tok − cs_token_flag − active_base;
            }
        }
```

⟨ Test if two characters match 505 ⟩ ≡

```
{ get_x_token_or_active_char;
  if ((cur_cmd > active_char) ∨ (cur_chr > 255))    /* not a character */
  { m = relax;
    n = 256;
  }
  else { m = cur_cmd;
    n = cur_chr;
  }
  get_x_token_or_active_char;
  if ((cur_cmd > active_char) ∨ (cur_chr > 255)) { cur_cmd = relax;
    cur_chr = 256;
  }
  if (this_if ≡ if_char_code) b = (n ≡ cur_chr); else b = (m ≡ cur_cmd);
}
```

This code is used in section 500.

506. 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 506 ⟩ ≡
  { *save_scanner_status* = *scanner_status*;
    scanner_status = *normal*;
    get_next( );
    n = *cur_cs*;
    p = *cur_cmd*;
    q = *cur_chr*;
    get_next( );
    if (*cur_cmd* $\neq p$) b = *false*;
    else if (*cur_cmd* < *call*) b = (*cur_chr* $\equiv q$);
    else ⟨ Test if two macro texts match 507 ⟩;
    scanner_status = *save_scanner_status*;
  }

This code is used in section 500.

507. 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 507 ⟩ ≡
  { p = *link*(*cur_chr*);
    q = *link*(*equiv*(n));   /* omit reference counts */
    if ($p \equiv q$) b = *true*;
    else { **while** (($p \neq$ *null*) ∧ ($q \neq$ *null*))
        if (*info*(p) $\neq$ *info*(q)) p = *null*;
        else { p = *link*(p);
          q = *link*(q);
        }
      b = (($p \equiv$ *null*) ∧ ($q \equiv$ *null*));
    }
  }

This code is used in section 506.

508. ⟨ Select the appropriate case and **return** or **goto** *common_ending* 508 ⟩ ≡

```
{ scan_int();
  n = cur_val;    /* n is the number of cases to pass */
  if (tracing_commands > 1) { begin_diagnostic();
    print("{case␣");
    print_int(n);
    print_char('}');
    end_diagnostic(false);
  }
  while (n ≠ 0) { pass_text();
    if (cond_ptr ≡ save_cond_ptr)
      if (cur_chr ≡ or_code) decr(n);
      else goto common_ending;
    else if (cur_chr ≡ fi_code) ⟨ Pop the condition stack 495 ⟩;
  }
  change_if_limit(or_code, save_cond_ptr);
  return;    /* wait for \or, \else, or \fi */
}
```

This code is used in section 500.

509. 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` 509 ⟩ ≡

```
{ if (tracing_ifs > 0)
    if (tracing_commands ≤ 1) show_cur_cmd_chr();
  if (cur_chr > if_limit)
    if (if_limit ≡ if_code) insert_relax();    /* condition not yet evaluated */
    else { print_err("Extra␣");
      print_cmd_chr(fi_or_else, cur_chr);
      help1("I'm␣ignoring␣this;␣it␣doesn't␣match␣any␣\\if.");
      error ( ) ;
    }
  else { while (cur_chr ≠ fi_code) pass_text();    /* skip to \fi */
    ⟨ Pop the condition stack 495 ⟩;
  }
}
```

This code is used in section 366.

510. 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.

511. 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

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

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⟩ +≡
static str_number *cur_name*; /∗ name of file just scanned ∗/
static str_number *cur_area*; /∗ file area just scanned, or `""` ∗/
static str_number *cur_ext*; /∗ file extension just scanned, or `""` ∗/

512. 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⟩ +≡
  static pool_pointer area_delimiter;   /* the most recent '>' or ':', if any */
  static pool_pointer ext_delimiter;    /* the relevant '.', if any */
```

513. 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/"
```

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

```
  static bool quoted_filename;

  static void begin_name(void)
  { area_delimiter = 0;
    ext_delimiter = 0;
    quoted_filename = false;
  }
```

515. 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.

```
  static bool more_name(ASCII_code c)
  { if (c ≡ '␣' ∧ ¬quoted_filename) return false;
    else if (c ≡ '"') { quoted_filename = ¬quoted_filename;
      return true;
    }
    else { str_room(1);
      append_char(c);    /* contribute c to the current string */
      if (IS_DIR_SEP(c)) { area_delimiter = cur_length;
        ext_delimiter = 0;
      }
      else if (c ≡ '.') ext_delimiter = cur_length;
      return true;
    }
  }
```

516. The third.

```
static void end_name(void)
{ if (str_ptr + 3 > max_strings) overflow("number␣of␣strings", max_strings − init_str_ptr);
  if (area_delimiter ≡ 0) cur_area = empty_string;
  else { cur_area = str_ptr;
    str_start[str_ptr + 1] = str_start[str_ptr] + area_delimiter;
    incr(str_ptr);
  }
  if (ext_delimiter ≡ 0) { cur_ext = empty_string;
    cur_name = make_string();
  }
  else { 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();
  }
}
```

517. 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 55 ⟩ +≡

```
static void print_file_name(int n, int a, int e)
{ slow_print(a);
  slow_print(n);
  slow_print(e);
}
```

518. 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(A)
        { c = A;
          incr(k);
          if (k ≤ file_name_size) name_of_file[k] = xchr[c];
        }

static void pack_file_name(str_number n, str_number a, str_number e, char *f)
{ int k;    /* number of positions filled in name_of_file */
  ASCII_code c;    /* character being packed */
  int j;    /* index into str_pool */
  k = 0;
  for (j = str_start[a]; j ≤ str_start[a + 1] − 1; j++) append_to_name(so(str_pool[j]))
  for (j = str_start[n]; j ≤ str_start[n + 1] − 1; j++) append_to_name(so(str_pool[j]))
  if (f ≡ Λ)
    for (j = str_start[e]; j ≤ str_start[e + 1] − 1; j++) append_to_name(so(str_pool[j]))
  else
    while (*f ≠ 0) append_to_name(so(*f++))
  if (k ≤ file_name_size) name_length = k; else name_length = file_name_size;
  name_of_file[name_length + 1] = 0;
}
```

519. TEX Live does not use the global variable *TEX_format_default*. It is no longer needed to supply the text for default system areas and extensions related to format files.

520. Consequently TEX Live does not need the initialization of *TEX_format_default* either.

521. And TEX Live does not check the length of *TEX_format_default*.

522. The *format_extension*, however, is needed by TEX Live to create the format name from the job name.

#**define** *format_extension* `".fmt"`

523. 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 $buffer[loc \mathrel{..} (last - 1)]$, where $loc < last$ and $buffer[loc] \neq$ '␣'.

TEX Live uses the `kpathsearch` library to implement access to files. *open_fmt_file* is declared here and the actual implementation is in the section on TEX Live Integration.

⟨ Declare the function called *open_fmt_file* 523 ⟩ ≡

static bool *open_fmt_file*(**void**);

This code is used in section 1302.

524. 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*'.

```
static str_number make_name_string(void)
{ int k;    /* index into name_of_file */
  if ((pool_ptr + name_length > pool_size) ∨ (str_ptr ≡ max_strings) ∨ (cur_length > 0)) return '?';
  else { for (k = 1; k ≤ name_length; k++) append_char(xord[name_of_file[k]]);
    return make_string();
  }
}
static str_number a_make_name_string(alpha_file *f)
{ return make_name_string();
}
static str_number b_make_name_string(byte_file *f)
{ return make_name_string();
}
#ifdef INIT
static str_number w_make_name_string(word_file *f)
{ return make_name_string();
}
#endif
```

525. 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. There are two ways to specify the file name: as a general text argument or as a token (after expansion). The traditional token delimiter is the space. For a file name, however, a double quote is used as the token delimiter if the token starts with a double quote.

Once the *area_delimiter* and the *ext_delimiter* are defined, the final processing is shared for all variants.

When starting, `\relax` is skipped as well as blanks and non-calls. Then a test for the *left_brace* will branch to the code for scanning a general text.

```
static void scan_file_name(void)
{ pool_pointer j, k;    /* index into str_pool */
  int old_setting;    /* holds selector setting */
  name_in_progress = true;
  begin_name();
  ⟨Get the next non-blank non-relax non-call token 403⟩;
  if (cur_cmd ≡ left_brace) ⟨Define a general text file name and goto done 1702⟩
  loop { if ((cur_cmd > other_char) ∨ (cur_chr > 255))    /* not a character */
    { back_input();
      goto done;
    }
#if 0    /* This is from pdftex-final.ch. I don't know these 'some cases', and I am not sure whether the
            name should end even if quoting is on. */
         /* If cur_chr is a space and we're not scanning a token list, check whether we're at the end of the
            buffer. Otherwise we end up adding spurious spaces to file names in some cases. */
    if (cur_chr ≡ '␣' ∧ state ≠ token_list ∧ loc > limit) goto done;
#endif
    if (¬more_name(cur_chr)) goto done;
    get_x_token();
  }
done: end_name();
  name_in_progress = false;
}
```

526. 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⟩ +≡
  static bool name_in_progress;    /* is a file name being scanned? */
  static str_number job_name;    /* principal file name */
  static bool log_opened;    /* has the transcript file been opened? */
```

527. Initially $job\_name \equiv 0$; it becomes nonzero as soon as the true name is known. We have $job\_name \equiv 0$ 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 54⟩ +≡
  job_name = 0;
  name_in_progress = false;
  log_opened = false;
```

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

```
#define pack_cur_name(A)
        if (cur_ext ≡ empty_string) pack_file_name(cur_name, cur_area, cur_ext, A);
        else pack_file_name(cur_name, cur_area, cur_ext, Λ)

  static void pack_job_name(char *s)    /* s ≡ ".log", ".dvi", or format_extension */
  { cur_area = empty_string;
    cur_ext = empty_string;
    cur_name = job_name;
    pack_cur_name(s);
  }
```

529. 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. We handle the specification of a file name with possibly spaces in double quotes (the last one is optional if this is the end of line i.e. the end of the buffer). Upon exit from the routine, variables *cur_name*, *cur_area*, *cur_ext*, and *name_of_file* are ready for another attempt at file opening.

```
  static void prompt_file_name(char *s, char *e)
  { int k;    /* index into buffer */
    if (interaction ≡ scroll_mode) wake_up_terminal;
    if (strcmp(s, "input␣file␣name") ≡ 0) 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 (strcmp(e, ".tex") ≡ 0) show_context();
    print_nl("Please␣type␣another␣");
    print(s);
    if (interaction < scroll_mode) fatal_error("***␣(job␣aborted,␣file␣error␣in␣nonstop␣mode)");
    clear_terminal;
    prompt_input(":␣");
    ⟨Scan file name in the buffer 530⟩;
    pack_cur_name(e);
  }
```

530. ⟨Scan file name in the buffer 530⟩ ≡

```
  { begin_name();
    k = first;
    while ((buffer[k] ≡ '␣') ∧ (k < last)) incr(k);
    loop { if (k ≡ last) goto done;
      if (¬more_name(buffer[k])) goto done;
      incr(k);
    }
  done: end_name();
  }
```

This code is used in section 529.

531. 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) { if (job_name ≡ 0) open_log_file( );
          pack_job_name(".dvi");
          while (¬b_open_out(&dvi_file)) prompt_file_name("file␣name␣for␣output", ".dvi");
          output_file_name = b_make_name_string(&dvi_file);
        }
```

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

532. ⟨Initialize the output routines 54⟩ +≡

```
  output_file_name = 0;
```

533. 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.

```
  static void open_log_file(void)
  { int old_setting;    /* previous selector setting */
    int k;    /* index into months and buffer */
    int l;    /* end of first input line */
    char months[] = "␣JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC";
      /* abbreviations of month names */
    old_setting = selector;
    if (job_name ≡ 0) job_name = s_no(c_job_name ? c_job_name : "texput");    /* TEX Live */
    pack_job_name(".fls");
    recorder_change_filename((char *) name_of_file + 1);
    pack_job_name(".log");
    while (¬a_open_out(&log_file)) ⟨Try to get a different log file name 534⟩;
    log_name = a_make_name_string(&log_file);
    selector = log_only;
    log_opened = true;
    ⟨Print the banner line, including the date and time 535⟩;
    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) decr(l);
    for (k = 1; k ≤ l; k++) printn(buffer[k]);
    print_ln( );    /* now the transcript file contains the first line of input */
    selector = old_setting + 2;    /* log_only or term_and_log */
  }
```

534. 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 534 ⟩ ≡

```
{ selector = term_only;
  prompt_file_name("transcript␣file␣name", ".log");
}
```

This code is used in section 533.

535. ⟨ Print the banner line, including the date and time 535 ⟩ ≡

```
{ wlog("%s", banner);
  slow_print(format_ident);
  print("␣␣");
  print_int(sys_day);
  print_char('␣');
  for (k = 3 * sys_month − 2; k ≤ 3 * sys_month; k++) wlog("%c", months[k]);
  print_char('␣');
  print_int(sys_year);
  print_char('␣');
  print_two(sys_time/60);
  print_char(':');
  print_two(sys_time % 60);
  if (eTeX_ex) { ;
    wlog_cr;
    wlog("entering␣extended␣mode");
  }
  if (Prote_ex) { ;
    wlog_cr;
    wlog("entering␣Prote␣mode");
  }
}
```

This code is used in section 533.

536. 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.

```
static void start_input(void)    /* TeX will \input something */
  { scan_file_name();    /* set cur_name to desired file name */
  pack_cur_name("");
  loop { begin_file_reading();    /* set up cur_file and new level of input */
    if (kpse_in_name_ok((char *) name_of_file + 1) ∧ a_open_in(&cur_file)) goto done;
    end_file_reading();    /* remove the level that didn't work */
    prompt_file_name("input␣file␣name", ".tex");
  }
done: name = a_make_name_string(&cur_file);
  if (source_filename_stack[in_open] ≠ Λ) free(source_filename_stack[in_open]);
  source_filename_stack[in_open] = strdup((char *) name_of_file + 1);    /* TeX Live */
  if (full_source_filename_stack[in_open] ≠ Λ) free(full_source_filename_stack[in_open]);
  full_source_filename_stack[in_open] = strdup(full_name_of_file);
  ⟨Set new cur_file_num 1744⟩    /* new entry on the macro stack */
  { ⟨additional local variables for start_input 1772⟩⟨update the macro stack 1771⟩
  }
  if (job_name ≡ 0) { if (c_job_name ≡ Λ) job_name = cur_name;
    else job_name = s_no(c_job_name);
    open_log_file();    /* TeX Live */
  }    /* open_log_file doesn't show_context, so limit and loc needn't be set to meaningful values
         yet */
  if (term_offset + strlen(full_source_filename_stack[in_open]) > max_print_line − 2) print_ln();
  else if ((term_offset > 0) ∨ (file_offset > 0)) print_char('␣');
  print_char('(');
  incr(open_parens);
  print(full_source_filename_stack[in_open]);
  update_terminal;
  state = new_line;
  if (name ≡ str_ptr − 1)    /* conserve string pool space (but see note above) */
  { flush_string;
    name = cur_name;
  }
  ⟨Read the first line of the new file 537⟩;
  }
```

537. 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 537⟩ ≡
  { line = 1;
    if (input_ln(&cur_file, false)) do_nothing;
    firm_up_the_line();
    if (end_line_char_inactive) decr(limit);
    else buffer[limit] = end_line_char;
    first = limit + 1;
    loc = start;
  }
```

This code is used in section 536.

538. 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 ⟩ +≡
static byte_file *tfm_file*;

539. 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

$$lf \equiv 6 + lh + (ec - bc + 1) + nw + nh + nd + ni + nl + nk + ne + np.$$

Note that a font may contain as many as 256 characters (if $bc \equiv 0$ and $ec \equiv 255$), and as few as 0 characters (if $bc \equiv ec + 1$).

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.

540. 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 $+16$.

541. 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.

542. 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: *rem* (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*.

543. The *tag* field in a *char_info_word* has four values that explain how to interpret the *rem* field.

tag $\equiv$ 0 (*no_tag*) means that *rem* is unused.

tag $\equiv$ 1 (*lig_tag*) means that this character has a ligature/kerning program starting at position *rem* in the *lig_kern* array.

tag $\equiv$ 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 *rem* field gives the character code of the next larger character.

tag $\equiv$ 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*[*rem*].

Characters with *tag* $\equiv$ 2 and *tag* $\equiv$ 3 are treated as characters with *tag* $\equiv$ 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 */

544. 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: *rem*.

In a kern step, an additional space equal to $kern[256*(op\_byte-128)+rem]$ 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 *rem* 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 $skip\_byte \equiv 255$, 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 $skip\_byte \equiv 255$, there is a special ligature/kerning program for a boundary character at the left, beginning at location $256*op\_byte+rem$. 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 $256*op\_byte+rem$. 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

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

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

```
#define stop_flag qi(128) /* value indicating 'STOP' in a lig/kern program */
#define kern_flag qi(128) /* op code for a kern step */
#define skip_byte(A) A.b0
#define next_char(A) A.b1
#define op_byte(A) A.b2
#define rem_byte(A) A.b3
```

545. 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(A) A.b0 /* top piece in a recipe */
#define ext_mid(A) A.b1 /* mid piece in a recipe */
#define ext_bot(A) A.b2 /* bot piece in a recipe */
#define ext_rep(A) A.b3 /* rep piece in a recipe */
```

546. 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
```

547. 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 ⟩ +≡

```
typedef uint8_t internal_font_number; /* font in a char_node */
typedef int32_t font_index; /* index into font_info */
```

548. 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 ⟩ +≡
static memory_word *font_info*[*font_mem_size* + 1]; /* the big collection of font data */
static font_index *fmem_ptr*; /* first unused word of *font_info* */
static internal_font_number *font_ptr*; /* largest internal font number in use */
static four_quarters *font_check0*[*font_max* − *font_base* + 1], ∗**const** *font_check* = *font_check0* − *font_base*; /* check sum */
static scaled *font_size0*[*font_max* − *font_base* + 1], ∗**const** *font_size* = *font_size0* − *font_base*; /* "at" size */
static scaled *font_dsize0*[*font_max* − *font_base* + 1], ∗**const** *font_dsize* = *font_dsize0* − *font_base*; /* "design" size */
static font_index *font_params0*[*font_max* − *font_base* + 1], ∗**const** *font_params* = *font_params0* − *font_base*; /* how many font parameters are present */
static str_number *font_name0*[*font_max* − *font_base* + 1], ∗**const** *font_name* = *font_name0* − *font_base*; /* name of the font */
static str_number *font_area0*[*font_max* − *font_base* + 1], ∗**const** *font_area* = *font_area0* − *font_base*; /* area of the font */
static eight_bits *font_bc0*[*font_max* − *font_base* + 1], ∗**const** *font_bc* = *font_bc0* − *font_base*; /* beginning (smallest) character code */
static eight_bits *font_ec0*[*font_max* − *font_base* + 1], ∗**const** *font_ec* = *font_ec0* − *font_base*; /* ending (largest) character code */
static pointer *font_glue0*[*font_max* − *font_base* + 1], ∗**const** *font_glue* = *font_glue0* − *font_base*; /* glue specification for interword space, *null* if not allocated */
static bool *font_used0*[*font_max* − *font_base* + 1], ∗**const** *font_used* = *font_used0* − *font_base*; /* has a character from this font actually appeared in the output? */
static int *hyphen_char0*[*font_max* − *font_base* + 1], ∗**const** *hyphen_char* = *hyphen_char0* − *font_base*; /* current `\hyphenchar` values */
static int *skew_char0*[*font_max* − *font_base* + 1], ∗**const** *skew_char* = *skew_char0* − *font_base*; /* current `\skewchar` values */
static font_index *bchar_label0*[*font_max* − *font_base* + 1], ∗**const** *bchar_label* = *bchar_label0* − *font_base*; /* start of *lig_kern* program for left boundary character, *non_address* if there is none */
static int16_t *font_bchar0*[*font_max* − *font_base* + 1], ∗**const** *font_bchar* = *font_bchar0* − *font_base*; /* boundary character, *non_char* if there is none */
static int16_t *font_false_bchar0*[*font_max* − *font_base* + 1], ∗**const** *font_false_bchar* = *font_false_bchar0* − *font_base*; /* *font_bchar* if it doesn't exist in the font, otherwise *non_char* */

549. 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 ⟩ +≡
static int *char_base0* [*font_max* − *font_base* + 1], ∗**const** *char_base* = *char_base0* − *font_base*;
/∗ base addresses for *char_info* ∗/
static int *width_base0* [*font_max* − *font_base* + 1], ∗**const** *width_base* = *width_base0* − *font_base*;
/∗ base addresses for widths ∗/
static int *height_base0* [*font_max* − *font_base* + 1], ∗**const** *height_base* = *height_base0* − *font_base*;
/∗ base addresses for heights ∗/
static int *depth_base0* [*font_max* − *font_base* + 1], ∗**const** *depth_base* = *depth_base0* − *font_base*;
/∗ base addresses for depths ∗/
static int *italic_base0* [*font_max* − *font_base* + 1], ∗**const** *italic_base* = *italic_base0* − *font_base*;
/∗ base addresses for italic corrections ∗/
static int *lig_kern_base0* [*font_max* − *font_base* + 1], ∗**const** *lig_kern_base* = *lig_kern_base0* − *font_base*;
/∗ base addresses for ligature/kerning programs ∗/
static int *kern_base0* [*font_max* − *font_base* + 1], ∗**const** *kern_base* = *kern_base0* − *font_base*;
/∗ base addresses for kerns ∗/
static int *exten_base0* [*font_max* − *font_base* + 1], ∗**const** *exten_base* = *exten_base0* − *font_base*;
/∗ base addresses for extensible recipes ∗/
static int *param_base0* [*font_max* − *font_base* + 1], ∗**const** *param_base* = *param_base0* − *font_base*;
/∗ base addresses for font parameters ∗/

550. ⟨ Set initial values of key variables 21 ⟩ +≡
for (k = *font_base*; $k \le$ *font_max*; k++) *font_used*[k] = *false*;

551. 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) 163 ⟩ +≡
$font\_ptr = null\_font$;
$fmem\_ptr = 7$;
$font\_name[null\_font] = s\_no($`"nullfont"`$)$;
$font\_area[null\_font] = empty\_string$;
$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 $(k = 0;\ k \le 6;\ k{+}{+})\ font\_info[k].sc = 0$;

552. ⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
$primitive($`"nullfont"`$, set\_font, null\_font)$;
$text(frozen\_null\_font) = text(cur\_val)$;
$eqtb[frozen\_null\_font] = eqtb[cur\_val]$;

553. 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 $q \equiv char\_info(f)(c)$. 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** $char\_info(A,B)$ $font\_info[char\_base[A] + B].qqqq$
#**define** $char\_width(A,B)$ $font\_info[width\_base[A] + B.b0].sc$
#**define** $char\_exists(A)$ $(A.b0 > min\_quarterword)$
#**define** $char\_italic(A,B)$ $font\_info[italic\_base[A] + (qo(B.b2))/4].sc$
#**define** $height\_depth(A)$ $qo(A.b1)$
#**define** $char\_height(A,B)$ $font\_info[height\_base[A] + (B)/16].sc$
#**define** $char\_depth(A,B)$ $font\_info[depth\_base[A] + (B) \% 16].sc$
#**define** $char\_tag(A)$ $((qo(A.b2)) \% 4)$

554. 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 ⟩ +≡
  static four_quarters *null_character*; /∗ nonexistent character information ∗/

555. ⟨ Set initial values of key variables 21 ⟩ +≡
  $null\_character.b0 = min\_quarterword;$
  $null\_character.b1 = min\_quarterword;$
  $null\_character.b2 = min\_quarterword;$
  $null\_character.b3 = min\_quarterword;$

556. 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 $i \equiv$ *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*(A, B) *font_info*[*kern_base*$[A] + 256 *$ *op_byte*$(B) +$ *rem_byte*(B)].*sc*
#define *kern_base_offset* $256 * (128 +$ *min_quarterword*$)$
#define *lig_kern_start*(A, B) *lig_kern_base*$[A] + B.b3$ /∗ beginning of lig/kern program ∗/
#define *lig_kern_restart*(A, B)
lig_kern_base$[A] + 256 *$ *op_byte*$(B) +$ *rem_byte*$(B) + 32768 -$ *kern_base_offset*

557. Font parameters are referred to as *slant*(f), *space*(f), etc.

#define *param_end*(A) *param_base*$[A]$] . *sc*
#define *param*(A) *font_info* [$A +$ *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* 557 ⟩ ≡
quad(*cur_font*)

This code is used in section 454.

558. ⟨ The x-height for *cur_font* 558 ⟩ ≡
x_height(*cur_font*)

This code is used in section 454.

559. 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; $s \equiv -1000$ 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 abort goto bad_tfm   /* do this when the TFM data is wrong */

  static internal_font_number read_font_info(pointer u, str_number nom, str_number aire, scaled
        s)   /* input a TFM file */
  { int k;   /* index into font_info */
    bool file_opened;   /* was tfm_file successfully opened? */
    halfword lf, lh, bc, ec, nw, nh, nd, ni, nl, nk, ne, np;   /* sizes of subfiles */
    internal_font_number f;   /* the new font's number */
    internal_font_number g;   /* the number to return */
    eight_bits a, b, c, d;   /* byte variables */
    four_quarters qw;
    scaled sw;   /* accumulators */
    int bch_label;   /* left boundary start location, or infinity */
    int bchar;   /* boundary character, or 256 */
    scaled z;   /* the design size or the "at" size */
    int alpha;
    int beta;   /* auxiliary quantities used in fixed-point multiplication */

    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 561⟩;
  bad_tfm: ⟨Report that the font won't be loaded 560⟩;
  done:
    if (file_opened) b_close(&tfm_file);
    return g;
  }
```

560. 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, empty_string);
    if (s ≥ 0) { print("␣at␣");
      print_scaled(s);
      print("pt");
    }
    else if (s ≠ −1000) { print("␣scaled␣");
      print_int(−s);
    }
```

```
⟨Report that the font won't be loaded 560⟩ ≡
  start_font_error_message;
  if (file_opened) 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 559.

561. ⟨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* 561⟩ ≡

```
  ⟨Open tfm_file for input 562⟩;
  ⟨Read the TFM size fields 564⟩;
  ⟨Use size fields to allocate font information 565⟩;
  ⟨Read the TFM header 567⟩;
  ⟨Read character data 568⟩;
  ⟨Read box dimensions 570⟩;
  ⟨Read ligature/kern program 572⟩;
  ⟨Read extensible character recipes 573⟩;
  ⟨Read font parameters 574⟩;
  ⟨Make final adjustments and goto done 575⟩
```

This code is used in section 559.

562. ⟨Open *tfm_file* for input 562⟩ ≡

```
  file_opened = false;
  pack_file_name(nom, empty_string, empty_string, ".tfm");    /* TEX Live */
  if (¬b_open_in(&tfm_file)) abort;
  file_opened = true
```

This code is used in section 561.

563. 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.d* 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 '{ *get*(*tfm_file*); **if** (*eof*(*tfm_file*)) *abort*; }'.

#**define** *fget* *get*(*tfm_file*)
#**define** *fbyte* *tfm_file.d*
#**define** *read_sixteen*(*A*)
{ $A = fbyte$;
if $(A > 127)$ *abort*;
fget;
$A = A * {}^\circ 400 + fbyte$;
}
#**define** *store_four_quarters*(*A*)
{ *fget*;
$a = fbyte$;
$qw.b0 = qi(a)$;
fget;
$b = fbyte$;
$qw.b1 = qi(b)$;
fget;
$c = fbyte$;
$qw.b2 = qi(c)$;
fget;
$d = fbyte$;
$qw.b3 = qi(d)$;
$A = qw$;
}

564. ⟨ Read the TFM size fields 564 ⟩ ≡

```
{ read_sixteen(lf);
  fget;
  read_sixteen(lh);
  fget;
  read_sixteen(bc);
  fget;
  read_sixteen(ec);
  if ((bc > ec + 1) ∨ (ec > 255)) abort;
  if (bc > 255)     /* bc ≡ 256 and ec ≡ 255 */
  { bc = 1;
    ec = 0;
  }
  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) abort;
  if ((nw ≡ 0) ∨ (nh ≡ 0) ∨ (nd ≡ 0) ∨ (ni ≡ 0)) abort;
}
```

This code is used in section 561.

565. 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 565 ⟩ ≡

```
lf = lf − 6 − lh;     /* lf words should be loaded into font_info */
if (np < 7) lf = lf + 7 − np;     /* at least seven parameters will appear */
if ((font_ptr ≡ font_max) ∨ (fmem_ptr + lf > font_mem_size))
  ⟨ Apologize for not loading the font, goto done 566 ⟩;
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 561.

566. ⟨ Apologize for not loading the font, **goto** *done* 566 ⟩ ≡

```
{ 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;
}
```

This code is used in section 565.

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

⟨ Read the TFM header 567 ⟩ ≡

```
{ if (lh < 2) 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/°20);
  if (z < unity) abort;
  while (lh > 2) { fget;
    fget;
    fget;
    fget;
    decr(lh);    /* ignore the rest of the header */
  }
  font_dsize[f] = z;
  if (s ≠ −1000)
    if (s ≥ 0) z = s;
    else z = xn_over_d(z, −s, 1000);
  font_size[f] = z;
}
```

This code is used in section 561.

568. ⟨ Read character data 568 ⟩ ≡

```
for (k = fmem_ptr; k ≤ width_base[f] − 1; k++) { store_four_quarters(font_info[k].qqqq);
  if ((a ≥ nw) ∨ (b/°20 ≥ nh) ∨ (b % °20 ≥ nd) ∨ (c/4 ≥ ni)) abort;
  switch (c % 4) {
  case lig_tag:
    if (d ≥ nl) abort; break;
  case ext_tag:
    if (d ≥ ne) abort; break;
  case list_tag: ⟨ Check for charlist cycle 569 ⟩ break;
  default: do_nothing;    /* no_tag */
  }
}
```

This code is used in section 561.

569. 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(A)
        { if ((A < bc) ∨ (A > ec)) abort; }
#define current_character_being_worked_on  k + bc − fmem_ptr
⟨Check for charlist cycle 569⟩ ≡
  { check_byte_range(d);
    while (d < current_character_being_worked_on) { 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) goto not_found;
      d = qo(rem_byte(qw));    /* next character on the list */
    }
    if (d ≡ current_character_being_worked_on) abort;    /* yes, there's a cycle */
  not_found: ;
  }
```

This code is used in section 568.

570. 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(A)
        { fget;
          a = fbyte;
          fget;
          b = fbyte;
          fget;
          c = fbyte;
          fget;
          d = fbyte;
          sw = (((((d * z)/°400) + (c * z))/°400) + (b * z))/beta;
          if (a ≡ 0) A = sw; else if (a ≡ 255) A = sw − alpha; else abort;
        }
```

```
⟨Read box dimensions 570⟩ ≡
  { ⟨Replace z by z′ and compute α, β 571⟩;
    for (k = width_base[f]; k ≤ lig_kern_base[f] − 1; k++) store_scaled(font_info[k].sc);
    if (font_info[width_base[f]].sc ≠ 0) abort;     /* width[0] must be zero */
    if (font_info[height_base[f]].sc ≠ 0) abort;    /* height[0] must be zero */
    if (font_info[depth_base[f]].sc ≠ 0) abort;     /* depth[0] must be zero */
    if (font_info[italic_base[f]].sc ≠ 0) abort;    /* italic[0] must be zero */
  }
```

This code is used in section 561.

571. ⟨Replace z by z' and compute α, β 571⟩ ≡

```
  { alpha = 16;
    while (z ≥ °40000000) { z = z/2;
      alpha = alpha + alpha;
    }
    beta = 256/alpha;
    alpha = alpha * z;
  }
```

This code is used in section 570.

572. **#define** *check_existence*(A)
{ *check_byte_range*(A);
qw = *char_info*(f, A); /* N.B.: not *qi*(A) */
if $(\neg$*char_exists*$(qw))$ *abort*;
}

⟨ Read ligature/kern program 572 ⟩ ≡
bch_label = °77777;
bchar = 256;
if $(nl > 0)$ { **for** $(k = lig\_kern\_base[f];\ k \le kern\_base[f] + kern\_base\_offset - 1;\ k{+}{+})$ {
store_four_quarters$(font\_info[k].qqqq)$;
if $(a > 128)$ { **if** $(256 * c + d \ge nl)$ *abort*;
if $(a \equiv 255)$
if $(k \equiv lig\_kern\_base[f])$ $bchar = b$;
}
else { **if** $(b \ne bchar)$ *check_existence*(b);
if $(c < 128)$ *check_existence*(d) /* check ligature */
else if $(256 * (c - 128) + d \ge nk)$ *abort*; /* check kern */
if $(a < 128)$
if $(k - lig\_kern\_base[f] + a + 1 \ge nl)$ *abort*;
}
}
if $(a \equiv 255)$ $bch\_label = 256 * c + d$;
}
for $(k = kern\_base[f] + kern\_base\_offset;\ k \le exten\_base[f] - 1;\ k{+}{+})$ *store_scaled*$(font\_info[k].sc)$;

This code is used in section 561.

573. ⟨ Read extensible character recipes 573 ⟩ ≡
for $(k = exten\_base[f];\ k \le param\_base[f] - 1;\ k{+}{+})$ { *store_four_quarters*$(font\_info[k].qqqq)$;
if $(a \ne 0)$ *check_existence*(a);
if $(b \ne 0)$ *check_existence*(b);
if $(c \ne 0)$ *check_existence*(c);
check_existence(d);
}

This code is used in section 561.

574. 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 574⟩ ≡
{ **for** $(k = 1;\ k \le np;\ k{+}{+})$
  if $(k \equiv 1)$ /∗ the *slant* parameter is a pure number ∗/
  { *fget*;
   $sw = fbyte$;
   if $(sw > 127)$ $sw = sw - 256$;
   fget;
   $sw = sw * {}^\circ 400 + fbyte$;
   fget;
   $sw = sw * {}^\circ 400 + fbyte$;
   fget;
   $font\_info[param\_base[f]].sc = (sw * {}^\circ 20) + (fbyte / {}^\circ 20)$;
  }
  else $store\_scaled(font\_info[param\_base[f] + k - 1].sc)$;
 if $(eof(tfm\_file))$ *abort*;
 for $(k = np + 1;\ k \le 7;\ k{+}{+})$ $font\_info[param\_base[f] + k - 1].sc = 0$;
}

This code is used in section 561.

575. 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 $adjust(A)$ $A[f] = qo(A[f])$ /∗ correct for the excess *min_quarterword* that was added ∗/

⟨Make final adjustments and **goto** *done* 575⟩ ≡
 if $(np \ge 7)$ $font\_params[f] = np$; **else** $font\_params[f] = 7$;
 $hyphen\_char[f] = default\_hyphen\_char$;
 $skew\_char[f] = default\_skew\_char$;
 if $(bch\_label < nl)$ $bchar\_label[f] = bch\_label + lig\_kern\_base[f]$;
 else $bchar\_label[f] = non\_address$;
 $font\_bchar[f] = qi(bchar)$;
 $font\_false\_bchar[f] = qi(bchar)$;
 if $(bchar \le ec)$
  if $(bchar \ge bc)$ { $qw = char\_info(f, bchar)$; /∗ N.B.: not $qi(bchar)$ ∗/
   if $(char\_exists(qw))$ $font\_false\_bchar[f] = non\_char$;
  }
 $font\_name[f] = nom$;
 $font\_area[f] = aire$;
 $font\_bc[f] = bc$;
 $font\_ec[f] = ec$;
 $font\_glue[f] = null$;
 $adjust(char\_base)$;
 $adjust(width\_base)$;
 $adjust(lig\_kern\_base)$;
 $adjust(kern\_base)$;
 $adjust(exten\_base)$;
 $decr(param\_base[f])$;
 $fmem\_ptr = fmem\_ptr + lf$;
 $font\_ptr = f$;
 $g = f$; **goto** *done*

This code is used in section 561.

576. 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 576 ⟩ ≡

```
static void scan_font_ident(void)
{ internal_font_number f;
  halfword m;
  ⟨ Get the next non-blank non-call token 405 ⟩;
  if (cur_cmd ≡ def_font) f = cur_font;
  else if (cur_cmd ≡ set_font) f = cur_chr;
  else if (cur_cmd ≡ def_family) { m = cur_chr;
    scan_four_bit_int();
    f = equiv(m + cur_val);
  }
  else { 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;
  }
  cur_val = f;
}
```

See also section 577.

This code is used in section 408.

577. 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 576 ⟩ +≡

```
static void find_font_dimen(bool writing)   /* sets cur_val to font_info location */
{ internal_font_number f;
  int n;   /* the parameter number */
  scan_int();
  n = cur_val;
  scan_font_ident();
  f = cur_val;
  if (n ≤ 0) cur_val = fmem_ptr;
  else { if (writing ∧ (n ≤ space_shrink_code) ∧
          (n ≥ space_code) ∧ (font_glue[f] ≠ null)) { delete_glue_ref(font_glue[f]);
      font_glue[f] = null;
    }
    if (n > font_params[f])
      if (f < font_ptr) cur_val = fmem_ptr;
      else ⟨ Increase the number of parameters in the last font 579 ⟩
    else cur_val = n + param_base[f];
  }
  ⟨ Issue an error message if cur_val = fmem_ptr 578 ⟩;
}
```

578. ⟨ Issue an error message if *cur_val* = *fmem_ptr* 578 ⟩ ≡

```
if (cur_val ≡ fmem_ptr) { print_err("Font␣");
  printn_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( );
}
```

This code is used in section 577.

579. ⟨ Increase the number of parameters in the last font 579 ⟩ ≡

```
{ do {
    if (fmem_ptr ≡ font_mem_size) overflow("font␣memory", font_mem_size);
    font_info[fmem_ptr].sc = 0;
    incr(fmem_ptr);
    incr(font_params[f]);
  } while (¬(n ≡ font_params[f]));
  cur_val = fmem_ptr − 1;    /* this equals param_base[f] + font_params[f] */
}
```

This code is used in section 577.

580. 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.

```
static void char_warning(internal_font_number f, eight_bits c)
{ int old_setting;    /* saved value of tracing_online */
  if (tracing_lost_chars > 0) { old_setting = tracing_online;
    if (eTeX_ex ∧ (tracing_lost_chars > 1)) tracing_online = 1;
    { 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);
    }
    tracing_online = old_setting;
  }
}
```

581. 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.

```
static pointer new_character(internal_font_number f, eight_bits c)
{ pointer p;    /* newly allocated node */
  if (font_bc[f] ≤ c)
    if (font_ec[f] ≥ c)
      if (char_exists(char_info(f, qi(c)))) { p = get_avail();
        font(p) = f;
        character(p) = qi(c);
        return p;
      }
  char_warning(f, c);
  return null;
}
```

582. 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.)

583. 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.

584. 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 $h = h + b$. 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 $(h, v, w, x, y, z) = (0, 0, 0, 0, 0, 0)$ 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 $h = h+b$, 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 $-2^{23} \le b <$.

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

w0 147. Set $h = h+w$; 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 $w = b$ and $h = h + b$. 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, $-2^{23} \le b <$.

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

x0 152. Set $h = h + x$; 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 $x = b$ and $h = h + b$. 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, $-2^{23} \le b <$.

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

down1 157 $a[1]$. Set $v = v + a$, 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 $-2^{23} \le a <$.

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

y0 161. Set $v = v + y$; 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 $y = a$ and $v = v + a$. 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, $-2^{23} \le a <$.

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

z0 166. Set $v = v + z$; 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 $z = a$ and $v = v + a$. 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, $-2^{23} \le a <$.

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

fnt_num_0 171. Set $f = 0$. 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 $f = 1, \ldots, f = 63$, respectively.

fnt1 235 $k[1]$. Set $f = k$. 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 $0 \le k <$.

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 $0 \le k <$.

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

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.

585. **#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 */

586. 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 $i \equiv 3$ 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 $i \equiv 4$, 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 */

587. 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 $a \equiv 0$. 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*.

588. 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.

589. 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

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

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.

590. 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., 0337 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.)

591. 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 ⟩ +≡

```
static int total_pages;  /* the number of pages that have been shipped out */
static scaled max_v;  /* maximum height-plus-depth of pages shipped so far */
static scaled max_h;  /* maximum width of pages shipped so far */
static int max_push;  /* deepest nesting of push commands encountered so far */
static int last_bop;  /* location of previous bop in the DVI output */
static int dead_cycles;  /* recent outputs that didn't ship anything out */
static bool doing_leaders;  /* are we inside a leader box? */

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

592. ⟨ 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;
```

593. 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 $dvi\_buf[0 \mathrel{..} half\_buf - 1]$ constitute the first half, and those in $dvi\_buf[half\_buf \mathrel{..} dvi\_buf\_size - 1]$ constitute the second. The global variable *dvi_ptr* points to the position that will receive the next output byte. When *dvi_ptr* reaches *dvi_limit*, which is always equal to one of the two values *half_buf* or *dvi_buf_size*, the half buffer that is about to be invaded next is sent to the output and *dvi_limit* is changed to its other value. Thus, there is always at least a half buffer's worth of information present, except at the very beginning of the job.

Bytes of the DVI file are numbered sequentially starting with 0; the next byte to be generated will be number $dvi\_offset + dvi\_ptr$. A byte is present in the buffer only if its number is $\geq dvi\_gone$.

⟨ Types in the outer block 18 ⟩ +≡
  typedef int16_t dvi_index; /* an index into the output buffer */

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

⟨ Global variables 13 ⟩ +≡
  static eight_bits $dvi\_buf[dvi\_buf\_size + 1]$; /* buffer for DVI output */
  static dvi_index *half_buf*; /* half of *dvi_buf_size* */
  static dvi_index *dvi_limit*; /* end of the current half buffer */
  static dvi_index *dvi_ptr*; /* the next available buffer address */
  static int *dvi_offset*;
    /* *dvi_buf_size* times the number of times the output buffer has been fully emptied */
  static int *dvi_gone*; /* the number of bytes already output to *dvi_file* */

595. 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 ⟩ +≡
  $half\_buf = dvi\_buf\_size/2$;
  $dvi\_limit = dvi\_buf\_size$;
  $dvi\_ptr = 0$;
  $dvi\_offset = 0$;
  $dvi\_gone = 0$;

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

```
static void write_dvi(dvi_index a, dvi_index b)
{ int k;
  for (k = a; k <= b; k++) pascal_write(dvi_file, "%c", dvi_buf[k]);
}
```

597. 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(A)  { dvi_buf[dvi_ptr] = A;
            incr(dvi_ptr);
            if (dvi_ptr ≡ dvi_limit) dvi_swap();
        }

  static void dvi_swap(void)    /* outputs half of the buffer */
  { if (dvi_limit ≡ dvi_buf_size) { write_dvi(0, half_buf − 1);
        dvi_limit = half_buf;
        dvi_offset = dvi_offset + dvi_buf_size;
        dvi_ptr = 0;
    }
    else { write_dvi(half_buf, dvi_buf_size − 1);
        dvi_limit = dvi_buf_size;
    }
    dvi_gone = dvi_gone + half_buf;
  }
```

598. 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 598⟩ ≡
  if (dvi_limit ≡ half_buf) write_dvi(half_buf, dvi_buf_size − 1);
  if (dvi_ptr > 0) write_dvi(0, dvi_ptr − 1)
```

This code is used in section 641.

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

```
  static void dvi_four(int x)
  { if (x ≥ 0) dvi_out(x/°100000000)
    else { x = x + °10000000000;
        x = x + °10000000000;
        dvi_out((x/°100000000) + 128);
    }
    x = x % °100000000;
    dvi_out(x/°200000);
    x = x % °200000;
    dvi_out(x/°400);
    dvi_out(x % °400);
  }
```

600. 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*.

```
  static void dvi_pop(int l)
  { if ((l ≡ dvi_offset + dvi_ptr) ∧ (dvi_ptr > 0)) decr(dvi_ptr);
    else dvi_out(pop);
  }
```

601. 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.

```
static void dvi_font_def(internal_font_number f)
{ int k;   /* index into str_pool */
  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 602⟩;
}
```

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

```
for (k = str_start[font_area[f]]; k ≤ str_start[font_area[f] + 1] − 1; k++) dvi_out(so(str_pool[k]));
for (k = str_start[font_name[f]]; k ≤ str_start[font_name[f] + 1] − 1; k++) dvi_out(so(str_pool[k]))
```

This code is used in section 601.

603. 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.)

604. 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(A) mem[A + 2].i   /* DVI byte number for a movement command */
⟨ Global variables 13 ⟩ +≡
  static pointer down_ptr, right_ptr;   /* heads of the down and right stacks */
```

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

```
  down_ptr = null;
  right_ptr = null;
```

606. 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: $y1 - down1 \equiv w1 - right1$ and $z1 - down1 \equiv x1 - right1$.

```
  static void movement(scaled w, eight_bits o)
  { small_number mstate;   /* have we seen a y or z? */
    pointer p, q;   /* current and top nodes on the stack */
    int k;   /* index into dvi_buf, modulo dvi_buf_size */

    q = get_node(movement_node_size);   /* new node for the top of the stack */
    width(q) = w;
    location(q) = dvi_offset + dvi_ptr;
    if (o ≡ down1) { link(q) = down_ptr;
      down_ptr = q;
    }
    else { link(q) = right_ptr;
      right_ptr = q;
    }
    ⟨ Look at the other stack entries until deciding what sort of DVI command to generate; goto found if
        node p is a "hit" 610 ⟩;
    ⟨ Generate a down or right command for w and return 609 ⟩;
  found: ⟨ Generate a y0 or z0 command in order to reuse a previous appearance of w 608 ⟩;
  }
```

607. 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 */
```

608. 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 608⟩ ≡

```
info(q) = info(p);
if (info(q) ≡ y_here) { dvi_out(o + y0 − down1);   /* y0 or w0 */
  while (link(q) ≠ p) { q = link(q);
    switch (info(q)) {
    case yz_OK: info(q) = z_OK; break;
    case y_OK: info(q) = d_fixed; break;
    default: do_nothing;
    }
  }
}
else { dvi_out(o + z0 − down1);   /* z0 or x0 */
  while (link(q) ≠ p) { q = link(q);
    switch (info(q)) {
    case yz_OK: info(q) = y_OK; break;
    case z_OK: info(q) = d_fixed; break;
    default: do_nothing;
    }
  }
}
```

This code is used in section 606.

609. ⟨ Generate a *down* or *right* command for w and **return** 609 ⟩ ≡

```
info(q) = yz_OK;
if (abs(w) ≥ °40000000) { dvi_out(o + 3);   /* down4 or right4 */
  dvi_four(w);
  return;
}
if (abs(w) ≥ °100000) { dvi_out(o + 2);   /* down3 or right3 */
  if (w < 0) w = w + °100000000;
  dvi_out(w/°200000);
  w = w % °200000;
  goto label2;
}
if (abs(w) ≥ °200) { dvi_out(o + 1);   /* down2 or right2 */
  if (w < 0) w = w + °200000;
  goto label2;
}
dvi_out(o);   /* down1 or right1 */
if (w < 0) w = w + °400;
goto label1;
label2: dvi_out(w/°400);
label1: dvi_out(w % °400); return
```

This code is used in section 606.

610. 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" 610 ⟩ ≡

```
p = link(q);
mstate = none_seen;
while (p ≠ null) { if (width(p) ≡ w)
    ⟨ Consider a node with matching width; goto found if it's a hit 611 ⟩
  else
    switch (mstate + info(p)) {
    case none_seen + y_here: mstate = y_seen; break;
    case none_seen + z_here: mstate = z_seen; break;
    case y_seen + z_here: case z_seen + y_here: goto not_found;
    default: do_nothing;
    }
  p = link(p);
}
not_found:
```

This code is used in section 606.

611. 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 611⟩ ≡
switch $(mstate + info(p))$ {
case $none\_seen + yz\_OK$: **case** $none\_seen + y\_OK$: **case** $z\_seen + yz\_OK$: **case** $z\_seen + y\_OK$:
  if $(location(p) < dvi\_gone)$ **goto** *not_found*;
  else ⟨Change buffered instruction to y or w and **goto** *found* 612⟩ **break**;
case $none\_seen + z\_OK$: **case** $y\_seen + yz\_OK$: **case** $y\_seen + z\_OK$:
  if $(location(p) < dvi\_gone)$ **goto** *not_found*;
  else ⟨Change buffered instruction to z or x and **goto** *found* 613⟩ **break**;
case $none\_seen + y\_here$: **case** $none\_seen + z\_here$: **case** $y\_seen + z\_here$: **case** $z\_seen + y\_here$:
  goto *found*;
default: *do_nothing*;
}

This code is used in section 610.

612. ⟨Change buffered instruction to y or w and **goto** *found* 612⟩ ≡
{ $k = location(p) - dvi\_offset$;
  if $(k < 0)$ $k = k + dvi\_buf\_size$;
  $dvi\_buf[k] = dvi\_buf[k] + y1 - down1$;
  $info(p) = y\_here$;
  goto *found*;
}

This code is used in section 611.

613. ⟨Change buffered instruction to z or x and **goto** *found* 613⟩ ≡
{ $k = location(p) - dvi\_offset$;
  if $(k < 0)$ $k = k + dvi\_buf\_size$;
  $dvi\_buf[k] = dvi\_buf[k] + z1 - down1$;
  $info(p) = z\_here$;
  goto *found*;
}

This code is used in section 611.

614. 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.

```
static void prune_movements(int l)   /* delete movement nodes with location ≥ l */
{ pointer p;   /* node being deleted */
  while (down_ptr ≠ null) { if (location(down_ptr) < l) goto done;
    p = down_ptr;
    down_ptr = link(p);
    free_node(p, movement_node_size);
  }
done:
  while (right_ptr ≠ null) { if (location(right_ptr) < l) return;
    p = right_ptr;
    right_ptr = link(p);
    free_node(p, movement_node_size);
  }
}
```

615. 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) { movement(cur_h − dvi_h, right1);
          dvi_h = cur_h;
        }
#define synch_v
        if (cur_v ≠ dvi_v) { movement(cur_v − dvi_v, down1);
          dvi_v = cur_v;
        }
```

⟨Global variables 13⟩ +≡

```
  static scaled dvi_h, dvi_v;   /* a DVI reader program thinks we are here */
  static scaled cur_h, cur_v;   /* TEX thinks we are here */
  static internal_font_number dvi_f;   /* the current font */
  static int cur_s;   /* current depth of output box nesting, initially −1 */
```

616. ⟨ Initialize variables as *ship_out* begins 616 ⟩ ≡

```
dvi_h = 0;
dvi_v = 0;
cur_h = h_offset;
dvi_f = null_font;
ensure_dvi_open;
if (total_pages ≡ 0) { 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/60);
  print_two(time % 60);
  selector = old_setting;
  dvi_out(cur_length);
  for (s = str_start[str_ptr]; s ≤ pool_ptr − 1; s++) dvi_out(so(str_pool[s]));
  pool_ptr = str_start[str_ptr];    /* flush the current string */
}
```

This code is used in section 639.

617. 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*).

```
static void vlist_out(void);    /* hlist_out and vlist_out are mutually recursive */
```

618. 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.

⟨Declare procedures needed in *hlist_out*, *vlist_out* 1367⟩

```
static void hlist_out(void)   /* output an hlist_node box */
{ scaled base_line;   /* the baseline coordinate for this box */
  scaled left_edge;   /* the left coordinate for this box */
  scaled save_h, save_v;   /* what dvi_h and dvi_v should pop to */
  pointer this_box;   /* pointer to containing box */
  glue_ord g_order;   /* applicable order of infinity for glue */
  int g_sign;   /* selects type of glue */
  pointer p;   /* current position in the hlist */
  int save_loc;   /* DVI byte location upon entry */
  pointer leader_box;   /* the leader box being replicated */
  scaled leader_wd;   /* width of leader box being replicated */
  scaled lx;   /* extra space between leader boxes */
  bool outer_doing_leaders;   /* were we doing leaders? */
  scaled edge;   /* left edge of sub-box, or right edge of leader space */
  double glue_temp;   /* glue value before rounding */
  double cur_glue;   /* glue seen so far */
  scaled cur_g;   /* rounded equivalent of cur_glue times the glue ratio */

  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) dvi_out(push);
  if (cur_s > max_push) max_push = cur_s;
  save_loc = dvi_offset + dvi_ptr;
  base_line = cur_v;
  left_edge = cur_h;
  while (p ≠ null) ⟨Output node p for hlist_out and move to the next node, maintaining the
        condition cur_v = base_line 619⟩;
  prune_movements(save_loc);
  if (cur_s > 0) dvi_pop(save_loc);
  decr(cur_s);
}
```

619. 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 $set\_char\_0 \equiv 0$.

⟨ Output node p for *hlist_out* and move to the next node, maintaining the condition $cur\_v = base\_line$ 619 ⟩ ≡
reswitch:
  if (*is_char_node*(*p*)) { *synch_h*;
    synch_v;
    do { $f = font(p)$;
      $c = character(p)$;
      if ($f \neq dvi\_f$) ⟨ Change font *dvi_f* to *f* 620 ⟩;
      if ($c \geq qi(128)$) *dvi_out*(*set1*);
      dvi_out(*qo*(*c*));
      $cur\_h = cur\_h + char\_width(f, char\_info(f, c))$;
      $p = link(p)$;
    } **while** ($\neg(\neg is\_char\_node(p))$);
    $dvi\_h = cur\_h$;
  }
  else ⟨ Output the non-*char_node* *p* for *hlist_out* and move to the next node 621 ⟩

This code is used in section 618.

620. ⟨ Change font *dvi_f* to *f* 620 ⟩ ≡
  { **if** ($\neg font\_used[f]$) { *dvi_font_def*(*f*);
      $font\_used[f] = true$;
    }
    if ($f \leq 64 + font\_base$) $dvi\_out(f - font\_base - 1 + fnt\_num\_0)$
    else { *dvi_out*(*fnt1*);
      $dvi\_out(f - font\_base - 1)$;
    }
    $dvi\_f = f$;
  }

This code is used in section 619.

621. ⟨Output the non-*char_node* p for *hlist_out* and move to the next node 621⟩ ≡

```
{ switch (type(p)) {
    case hlist_node: case vlist_node: ⟨Output a box in an hlist 622⟩ break;
    case rule_node:
      { rule_ht = height(p);
        rule_dp = depth(p);
        rule_wd = width(p);
        goto fin_rule;
      }
    case whatsit_node: ⟨Output the whatsit node p in an hlist 1366⟩; break;
    case glue_node: ⟨Move right or output leaders 624⟩
    case kern_node: case math_node: cur_h = cur_h + width(p); break;
    case ligature_node: ⟨Make node p look like a char_node and goto reswitch 651⟩
    default: do_nothing;
    }
    goto next_p;
  fin_rule: ⟨Output a rule in an hlist 623⟩;
  move_past: cur_h = cur_h + rule_wd;
  next_p: p = link(p);
  }
```

This code is used in section 619.

622. ⟨Output a box in an hlist 622⟩ ≡

```
  if (list_ptr(p) ≡ null) cur_h = cur_h + width(p);
  else { 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) vlist_out(); else hlist_out();
    dvi_h = save_h;
    dvi_v = save_v;
    cur_h = edge + width(p);
    cur_v = base_line;
  }
```

This code is used in section 621.

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

```
  if (is_running(rule_ht)) rule_ht = height(this_box);
  if (is_running(rule_dp)) rule_dp = depth(this_box);
  rule_ht = rule_ht + rule_dp;      /* this is the rule thickness */
  if ((rule_ht > 0) ∧ (rule_wd > 0))      /* we don't output empty rules */
  { 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;
  }
```

This code is used in section 621.

624. **#define** *billion* *float_constant*(1000000000)
#define *vet_glue*(A) *glue_temp* $= A$;
  if (*glue_temp* $>$ *billion*) *glue_temp* $=$ *billion*;
  else if (*glue_temp* $< -$*billion*) *glue_temp* $= -$*billion*

⟨Move right or output leaders 624⟩ ≡
{ $g =$ *glue_ptr*(p);
  rule_wd $=$ *width*(g) $-$ *cur_g*;
  if (*g_sign* $\neq$ *normal*) { **if** (*g_sign* $\equiv$ *stretching*) { **if** (*stretch_order*(g) $\equiv$ *g_order*) {
      cur_glue $=$ *cur_glue* $+$ *stretch*(g);
      vet_glue(*unfix*(*glue_set*(*this_box*)) $*$ *cur_glue*);
      cur_g $=$ *round*(*glue_temp*);
    }
  }
  else if (*shrink_order*(g) $\equiv$ *g_order*) { *cur_glue* $=$ *cur_glue* $-$ *shrink*(g);
    vet_glue(*unfix*(*glue_set*(*this_box*)) $*$ *cur_glue*);
    cur_g $=$ *round*(*glue_temp*);
  }
  }
  rule_wd $=$ *rule_wd* $+$ *cur_g*;
  if (*subtype*(p) $\geq$ *a_leaders*)
    ⟨Output leaders in an hlist, **goto** *fin_rule* if a rule or to *next_p* if done 625⟩;
  goto *move_past*;
}

This code is used in section 621.

625. ⟨Output leaders in an hlist, **goto** *fin_rule* if a rule or to *next_p* if done 625⟩ ≡
{ *leader_box* $=$ *leader_ptr*(p);
  if (*type*(*leader_box*) $\equiv$ *rule_node*) { *rule_ht* $=$ *height*(*leader_box*);
    rule_dp $=$ *depth*(*leader_box*);
    goto *fin_rule*;
  }
  leader_wd $=$ *width*(*leader_box*);
  if ((*leader_wd* > 0) $\wedge$ (*rule_wd* > 0)) { *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 626⟩;
    while (*cur_h* $+$ *leader_wd* $\leq$ *edge*)
      ⟨Output a leader box at *cur_h*, then advance *cur_h* by *leader_wd* $+$ *lx* 627⟩;
    cur_h $=$ *edge* $- 10$;
    goto *next_p*;
  }
}

This code is used in section 624.

626. 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 / 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 626⟩ ≡

```
if (subtype(p) ≡ a_leaders) { save_h = cur_h;
  cur_h = left_edge + leader_wd * ((cur_h − left_edge)/leader_wd);
  if (cur_h < save_h) cur_h = cur_h + leader_wd;
}
else { lq = rule_wd/leader_wd;   /* the number of box copies */
  lr = rule_wd % leader_wd;   /* the remaining space */
  if (subtype(p) ≡ c_leaders) cur_h = cur_h + (lr/2);
  else { lx = lr/(lq + 1);
    cur_h = cur_h + ((lr − (lq − 1) * lx)/2);
  }
}
```

This code is used in section 625.

627. 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* 627⟩ ≡

```
{ 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) 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;
}
```

This code is used in section 625.

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

```
static void vlist_out(void) /* output a vlist_node box */
{ scaled left_edge; /* the left coordinate for this box */
  scaled top_edge; /* the top coordinate for this box */
  scaled save_h, save_v; /* what dvi_h and dvi_v should pop to */
  pointer this_box; /* pointer to containing box */
  glue_ord g_order; /* applicable order of infinity for glue */
  int g_sign; /* selects type of glue */
  pointer p; /* current position in the vlist */
  int save_loc; /* DVI byte location upon entry */
  pointer leader_box; /* the leader box being replicated */
  scaled leader_ht; /* height of leader box being replicated */
  scaled lx; /* extra space between leader boxes */
  bool outer_doing_leaders; /* were we doing leaders? */
  scaled edge; /* bottom boundary of leader space */
  double glue_temp; /* glue value before rounding */
  double cur_glue; /* glue seen so far */
  scaled cur_g; /* rounded equivalent of cur_glue times the glue ratio */

  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) dvi_out(push);
  if (cur_s > max_push) 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) ⟨Output node p for vlist_out and move to the next node, maintaining the condition
        cur_h = left_edge 629⟩;
  prune_movements(save_loc);
  if (cur_s > 0) dvi_pop(save_loc);
  decr(cur_s);
}
```

629. ⟨Output node *p* for *vlist_out* and move to the next node, maintaining the condition *cur_h* = *left_edge* 629⟩ ≡

```
{ if (is_char_node(p)) confusion("vlistout");
  else ⟨Output the non-char_node p for vlist_out 630⟩;
next_p: p = link(p);
}
```

This code is used in section 628.

630. ⟨ Output the non-*char_node* p for *vlist_out* 630 ⟩ ≡
{ **switch** (*type*(*p*)) {
  case *hlist_node*: **case** *vlist_node*: ⟨ Output a box in a vlist 631 ⟩ **break**;
  case *rule_node*:
    { *rule_ht* = *height*(*p*);
      rule_dp = *depth*(*p*);
      rule_wd = *width*(*p*);
      goto *fin_rule*;
    }
  case *whatsit_node*: ⟨ Output the whatsit node p in a vlist 1365 ⟩; **break**;
  case *glue_node*: ⟨ Move down or output leaders 633 ⟩
  case *kern_node*: *cur_v* = *cur_v* + *width*(*p*); **break**;
  default: *do_nothing*;
  }
  goto *next_p*;
fin_rule: ⟨ Output a rule in a vlist, **goto** *next_p* 632 ⟩;
move_past: *cur_v* = *cur_v* + *rule_ht*;
}

This code is used in section 629.

631. 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 631 ⟩ ≡
  if (*list_ptr*(*p*) ≡ *null*) *cur_v* = *cur_v* + *height*(*p*) + *depth*(*p*);
  else { *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*) *vlist_out*( ); **else** *hlist_out*( );
    dvi_h = *save_h*;
    dvi_v = *save_v*;
    cur_v = *save_v* + *depth*(*p*);
    cur_h = *left_edge*;
  }

This code is used in section 630.

632. ⟨ Output a rule in a vlist, **goto** *next_p* 632 ⟩ ≡
  if (*is_running*(*rule_wd*)) *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))   /∗ we don't output empty rules ∗/
  { *synch_h*;
    synch_v;
    dvi_out(*put_rule*);
    dvi_four(*rule_ht*);
    dvi_four(*rule_wd*);
  }
  goto *next_p*

This code is used in section 630.

633. ⟨Move down or output leaders 633⟩ ≡

```
{ g = glue_ptr(p);
  rule_ht = width(g) − cur_g;
  if (g_sign ≠ normal) { if (g_sign ≡ stretching) { if (stretch_order(g) ≡ g_order) {
          cur_glue = cur_glue + stretch(g);
          vet_glue(unfix(glue_set(this_box)) ∗ cur_glue);
          cur_g = round(glue_temp);
        }
      }
    else if (shrink_order(g) ≡ g_order) { cur_glue = cur_glue − shrink(g);
        vet_glue(unfix(glue_set(this_box)) ∗ cur_glue);
        cur_g = round(glue_temp);
      }
  }
  rule_ht = rule_ht + cur_g;
  if (subtype(p) ≥ a_leaders) ⟨Output leaders in a vlist, goto fin_rule if a rule or to next_p if done 634⟩;
  goto move_past;
}
```

This code is used in section 630.

634. ⟨Output leaders in a vlist, **goto** *fin_rule* if a rule or to *next_p* if done 634⟩ ≡

```
{ leader_box = leader_ptr(p);
  if (type(leader_box) ≡ rule_node) { rule_wd = width(leader_box);
    rule_dp = 0;
    goto fin_rule;
  }
  leader_ht = height(leader_box) + depth(leader_box);
  if ((leader_ht > 0) ∧ (rule_ht > 0)) { rule_ht = rule_ht + 10;
      /∗ compensate for floating-point rounding ∗/
    edge = cur_v + rule_ht;
    lx = 0;
    ⟨Let cur_v be the position of the first box, and set leader_ht + lx to the spacing between
        corresponding parts of boxes 635⟩;
    while (cur_v + leader_ht ≤ edge)
      ⟨Output a leader box at cur_v, then advance cur_v by leader_ht + lx 636⟩;
    cur_v = edge − 10;
    goto next_p;
  }
}
```

This code is used in section 633.

635. ⟨Let cur_v be the position of the first box, and set $leader\_ht + lx$ to the spacing between corresponding parts of boxes 635⟩ ≡

```
if (subtype(p) ≡ a_leaders) { save_v = cur_v;
  cur_v = top_edge + leader_ht * ((cur_v − top_edge)/leader_ht);
  if (cur_v < save_v) cur_v = cur_v + leader_ht;
}
else { lq = rule_ht/leader_ht;     /* the number of box copies */
  lr = rule_ht % leader_ht;     /* the remaining space */
  if (subtype(p) ≡ c_leaders) cur_v = cur_v + (lr/2);
  else { lx = lr/(lq + 1);
    cur_v = cur_v + ((lr − (lq − 1) * lx)/2);
  }
}
```

This code is used in section 634.

636. 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 $leader\_ht + lx$ 636⟩ ≡

```
{ 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) 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;
}
```

This code is used in section 634.

637. 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.

```
static void ship_out(pointer p)      /* output the box p */
{ int page_loc;      /* location of the current bop */
  int j, k;      /* indices to first ten count registers */
  int s;      /* index into str_pool */
  int old_setting;      /* saved selector setting */
  ⟨Local variables to save the profiling context 1761⟩
  ⟨Charge the time used here on ship_out 1766⟩
  if (tracing_output > 0) { print_nl("");
    print_ln();
    print("Completed␣box␣being␣shipped␣out");
  }
  if (term_offset > max_print_line − 9) print_ln();
  else if ((term_offset > 0) ∨ (file_offset > 0)) print_char('␣');
  print_char('[');
  j = 9;
  while ((count(j) ≡ 0) ∧ (j > 0)) decr(j);
  for (k = 0; k ≤ j; k++) { print_int(count(k));
    if (k < j) print_char('.');
  }
  update_terminal;
  if (tracing_output > 0) { print_char(']');
    begin_diagnostic();
    show_box(p);
    end_diagnostic(true);
  }
  ⟨Ship box p out 639⟩;
  if (tracing_output ≤ 0) print_char(']');
  dead_cycles = 0;
  update_terminal;      /* progress report */
  ⟨Flush the box from memory, showing statistics if requested 638⟩;
  ⟨restore the previous current file, line, and command 1763⟩
}
```

638. ⟨ Flush the box from memory, showing statistics if requested 638 ⟩ ≡

```
#ifdef STAT
  if (tracing_stats > 1) { print_nl("Memory␣usage␣before:␣");
    print_int(var_used);
    print_char('&');
    print_int(dyn_used);
    print_char(';');
  }
#endif
  flush_node_list(p);
#ifdef STAT
  if (tracing_stats > 1) { 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();
  }
#endif
```

This code is used in section 637.

639. ⟨ Ship box *p* out 639 ⟩ ≡

```
  ⟨ Update the values of max_h and max_v; but if the page is too large, goto done 640 ⟩;
  ⟨ Initialize variables as ship_out begins 616 ⟩;
  page_loc = dvi_offset + dvi_ptr;
  dvi_out(bop);
  for (k = 0; k ≤ 9; k++) 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) vlist_out(); else hlist_out();
  dvi_out(eop);
  incr(total_pages);
  cur_s = −1; done:
```

This code is used in section 637.

640. 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* 640 ⟩ ≡

```
if ((height(p) > max_dimen) ∨
      (depth(p) > max_dimen) ∨
      (height(p) + depth(p) + v_offset > max_dimen) ∨
      (width(p) + h_offset > max_dimen)) { 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) { begin_diagnostic( );
     print_nl("The␣following␣box␣has␣been␣deleted:");
     show_box(p);
     end_diagnostic(true);
   }
   goto done;
}
if (height(p) + depth(p) + v_offset > max_v) max_v = height(p) + depth(p) + v_offset;
if (width(p) + h_offset > max_h) max_h = width(p) + h_offset
```

This code is used in section 639.

641. 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 641⟩ ≡
  while (cur_s > −1) { if (cur_s > 0) dvi_out(pop)
    else { dvi_out(eop);
      incr(total_pages);
    }
    decr(cur_s);
  }
  if (total_pages ≡ 0) print_nl("No␣pages␣of␣output.");
  else { 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/256);
    dvi_out(max_push % 256);
    dvi_out((total_pages/256) % 256);
    dvi_out(total_pages % 256);
    ⟨Output the font definitions for all fonts that were used 642⟩;
    dvi_out(post_post);
    dvi_four(last_bop);
    dvi_out(id_byte);
    k = 4 + ((dvi_buf_size − dvi_ptr) % 4);    /∗ the number of 223's ∗/
    while (k > 0) { dvi_out(223);
      decr(k);
    }
    ⟨Empty the last bytes out of dvi_buf 598⟩;
    print_nl("Output␣written␣on␣");
    slow_print(output_file_name);
    print("␣(");
    print_int(total_pages);
    print("␣page");
    if (total_pages ≠ 1) print_char('s');
    print(",␣");
    print_int(dvi_offset + dvi_ptr);
    print("␣bytes).");
    b_close(&dvi_file);
  }
```

This code is used in section 1332.

642. ⟨Output the font definitions for all fonts that were used 642⟩ ≡

```
  while (font_ptr > font_base) { if (font_used[font_ptr]) dvi_font_def(font_ptr);
    decr(font_ptr);
  }
```

This code is used in section 641.

643. Packaging. We're essentially done with the parts of TEX that are concerned with the input (*get_next*) and the output (*ship_out*). So it's time to get heavily into the remaining part, which does the real work of typesetting.

After lists are constructed, TEX wraps them up and puts them into boxes. Two major subroutines are given the responsibility for this task: *hpack* applies to horizontal lists (hlists) and *vpack* applies to vertical lists (vlists). The main duty of *hpack* and *vpack* is to compute the dimensions of the resulting boxes, and to adjust the glue if one of those dimensions is pre-specified. The computed sizes normally enclose all of the material inside the new box; but some items may stick out if negative glue is used, if the box is overfull, or if a `\vbox` includes other boxes that have been shifted left.

The subroutine call $hpack(p, w, m)$ returns a pointer to an *hlist_node* for a box containing the hlist that starts at p. Parameter w specifies a width; and parameter m is either '*exactly*' or '*additional*'. Thus, $hpack(p, w, exactly)$ produces a box whose width is exactly w, while $hpack(p, w, additional)$ yields a box whose width is the natural width plus w. It is convenient to define a macro called '*natural*' to cover the most common case, so that we can say $hpack(p, natural)$ to get a box that has the natural width of list p.

Similarly, $vpack(p, w, m)$ returns a pointer to a *vlist_node* for a box containing the vlist that starts at p. In this case w represents a height instead of a width; the parameter m is interpreted as in *hpack*.

```
#define exactly 0     /* a box dimension is pre-specified */
#define additional 1     /* a box dimension is increased from the natural one */
#define natural 0, additional     /* shorthand for parameters to hpack and vpack */
```

644. The parameters to *hpack* and *vpack* correspond to TEX's primitives like '`\hbox to 300pt`', '`\hbox spread 10pt`'; note that '`\hbox`' with no dimension following it is equivalent to '`\hbox spread 0pt`'. The *scan_spec* subroutine scans such constructions in the user's input, including the mandatory left brace that follows them, and it puts the specification onto *save_stack* so that the desired box can later be obtained by executing the following code:

$$save\_ptr = save\_ptr - 2;$$
$$hpack(p, saved(1), saved(0)) .$$

Special care is necessary to ensure that the special *save_stack* codes are placed just below the new group code, because scanning can change *save_stack* when `\csname` appears.

```
static void scan_spec(group_code c, bool three_codes)     /* scans a box specification and left brace */
{ int s;     /* temporarily saved value */
  int spec_code;
  if (three_codes) s = saved(0);
  if (scan_keyword("to")) spec_code = exactly;
  else if (scan_keyword("spread")) spec_code = additional;
  else { spec_code = additional;
    cur_val = 0;
    goto found;
  }
  scan_normal_dimen;
found:
  if (three_codes) { saved(0) = s;
    incr(save_ptr);
  }
  saved(0) = spec_code;
  saved(1) = cur_val;
  save_ptr = save_ptr + 2;
  new_save_level(c);
  scan_left_brace();
}
```

645. 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 ⟩ +≡
static scaled *total_stretch0* [*filll* − *normal* + 1], ∗**const** *total_stretch* = *total_stretch0* − *normal*, *total_shrink0* [*filll* − *normal* + 1], ∗**const** *total_shrink* = *total_shrink0* − *normal*;
/∗ glue found by *hpack* or *vpack* ∗/
static int *last_badness*; /∗ badness of the most recently packaged box ∗/

646. 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 ⟩ +≡
static pointer *adjust_tail*; /∗ tail of adjustment list ∗/

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

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

static pointer *hpack*(**pointer** p, **scaled** w, **small_number** m)
{ **pointer** r; /* the box node that will be returned */
  pointer q; /* trails behind p */
  scaled h, d, x; /* height, depth, and natural width */
  scaled s; /* shift amount */
  pointer g; /* points to a glue specification */
  glue_ord o; /* order of infinity */
  internal_font_number f; /* the font in a *char_node* */
  four_quarters i; /* font information about a *char_node* */
  eight_bits *hd*; /* height and depth indices for a character */
  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 649 ⟩;
  while ($p \neq$ *null*) ⟨ 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 650 ⟩;
  if (*adjust_tail* ≠ *null*) *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* 656 ⟩;
common_ending: ⟨ Finish issuing a diagnostic message for an overfull or underfull hbox 662 ⟩;
end: **return** r;
}

649. ⟨ Clear dimensions to zero 649 ⟩ ≡
$d = 0$;
$x = 0$;
total_stretch[*normal*] = 0;
total_shrink[*normal*] = 0;
total_stretch[*fil*] = 0;
total_shrink[*fil*] = 0;
total_stretch[*fill*] = 0;
total_shrink[*fill*] = 0;
total_stretch[*filll*] = 0; *total_shrink*[*filll*] = 0

This code is used in sections 648 and 667.

650. ⟨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 650⟩ ≡

```
{ reswitch:
  while (is_char_node(p)) ⟨Incorporate character dimensions into the dimensions of the hbox that will
        contain it, then move to the next node 653⟩;
  if (p ≠ null) { switch (type(p)) {
    case hlist_node: case vlist_node: case rule_node: case unset_node:
      ⟨Incorporate box dimensions into the dimensions of the hbox that will contain it 652⟩ break;
    case ins_node: case mark_node: case adjust_node:
      if (adjust_tail ≠ null) ⟨Transfer node p to the adjustment list 654⟩ break;
    case whatsit_node: ⟨Incorporate a whatsit node into an hbox 1359⟩; break;
    case glue_node: ⟨Incorporate glue into the horizontal totals 655⟩ break;
    case kern_node: case math_node: x = x + width(p); break;
    case ligature_node: ⟨Make node p look like a char_node and goto reswitch 651⟩
    default: do_nothing;
    }
    p = link(p);
  }
}
```

This code is used in section 648.

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

```
{ mem[lig_trick] = mem[lig_char(p)];
  link(lig_trick) = link(p);
  p = lig_trick;
  goto reswitch;
}
```

This code is used in sections 621, 650, and 1146.

652. 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 652⟩ ≡

```
{ x = x + width(p);
  if (type(p) ≥ rule_node) s = 0; else s = shift_amount(p);
  if (height(p) − s > h) h = height(p) − s;
  if (depth(p) + s > d) d = depth(p) + s;
}
```

This code is used in section 650.

653. 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.

$\langle$ Incorporate character dimensions into the dimensions of the hbox that will contain it, then move to the next node 653 $\rangle \equiv$

{ $f = font(p)$;
  $i = char\_info(f, character(p))$;
  $hd = height\_depth(i)$;
  $x = x + char\_width(f, i)$;
  $s = char\_height(f, hd)$; **if** $(s > h)$ $h = s$;
  $s = char\_depth(f, hd)$; **if** $(s > d)$ $d = s$;
  $p = link(p)$;
}

This code is used in section 650.

654. 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.

$\langle$ Transfer node p to the adjustment list 654 $\rangle \equiv$

{ **while** $(link(q) \neq p)$ $q = link(q)$;
  if $(type(p) \equiv adjust\_node)$ { $link(adjust\_tail) = adjust\_ptr(p)$;
    while $(link(adjust\_tail) \neq null)$ $adjust\_tail = link(adjust\_tail)$;
    $p = link(p)$;
    $free\_node(link(q), small\_node\_size)$;
  }
  else { $link(adjust\_tail) = p$;
    $adjust\_tail = p$;
    $p = link(p)$;
  }
  $link(q) = p$;
  $p = q$;
}

This code is used in section 650.

655. $\langle$ Incorporate glue into the horizontal totals 655 $\rangle \equiv$

{ $g = glue\_ptr(p)$;
  $x = x + width(g)$;
  $o = stretch\_order(g)$;
  $total\_stretch[o] = total\_stretch[o] + stretch(g)$;
  $o = shrink\_order(g)$;
  $total\_shrink[o] = total\_shrink[o] + shrink(g)$;
  if $(subtype(p) \geq a\_leaders)$ { $g = leader\_ptr(p)$;
    if $(height(g) > h)$ $h = height(g)$;
    if $(depth(g) > d)$ $d = depth(g)$;
  }
}

This code is used in section 650.

656. 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* 656 ⟩ ≡
  if ($m \equiv$ *additional*) $w = x + w$;
  width(r) = w;
  $x = w - x$; /* now x is the excess to be made up */
  if ($x \equiv 0$) { *glue_sign*(r) = *normal*;
    glue_order(r) = *normal*;
    set_glue_ratio_zero(*glue_set*(r));
    goto *end*;
  }
  else if ($x > 0$) ⟨ Determine horizontal glue stretch setting, then **return** or **goto** *common_ending* 657 ⟩
  else ⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 663 ⟩

This code is used in section 648.

657. ⟨ Determine horizontal glue stretch setting, then **return** or **goto** *common_ending* 657 ⟩ ≡
  { ⟨ Determine the stretch order 658 ⟩;
    glue_order(r) = o;
    glue_sign(r) = *stretching*;
    if (*total_stretch*[o] $\neq 0$) *glue_set*(r) = *fix*(x/(**double**) *total_stretch*[o]);
    else { *glue_sign*(r) = *normal*;
      set_glue_ratio_zero(*glue_set*(r)); /* there's nothing to stretch */
    }
    if ($o \equiv$ *normal*)
      if (*list_ptr*(r) $\neq$ *null*)
        ⟨ Report an underfull hbox and **goto** *common_ending*, if this box is sufficiently bad 659 ⟩;
    goto *end*;
  }

This code is used in section 656.

658. ⟨ Determine the stretch order 658 ⟩ ≡
  if (*total_stretch*[*filll*] $\neq 0$) o = *filll*;
  else if (*total_stretch*[*fill*] $\neq 0$) o = *fill*;
  else if (*total_stretch*[*fil*] $\neq 0$) o = *fil*;
  else o = *normal*

This code is used in sections 657, 672, and 795.

659. ⟨ Report an underfull hbox and **goto** *common_ending*, if this box is sufficiently bad 659 ⟩ ≡
  { *last_badness* = *badness*(x, *total_stretch*[*normal*]);
    if (*last_badness* > *hbadness*) { *print_ln*( );
      if (*last_badness* > 100) *print_nl*("Underfull"); **else** *print_nl*("Loose");
      print("␣\\hbox␣(badness␣");
      print_int(*last_badness*);
      goto *common_ending*;
    }
  }

This code is used in section 657.

660. 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 ⟩ +≡
  static int *pack_begin_line*; /* source file line where the current paragraph or alignment began; a negative value denotes alignment */

661. ⟨ Set initial values of key variables 21 ⟩ +≡
  pack_begin_line = 0;

662. ⟨ Finish issuing a diagnostic message for an overfull or underfull hbox 662 ⟩ ≡

```
if (output_active) print(")␣has␣occurred␣while␣\\output␣is␣active");
else { if (pack_begin_line ≠ 0) { if (pack_begin_line > 0) print(")␣in␣paragraph␣at␣lines␣");
      else print(")␣in␣alignment␣at␣lines␣");
      print_int(abs(pack_begin_line));
      print("--");
    }
    else print(")␣detected␣at␣line␣");
    print_int(line);
}
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 648.

663. ⟨ Determine horizontal glue shrink setting, then **return** or **goto** *common_ending* 663 ⟩ ≡

```
{ ⟨ Determine the shrink order 664 ⟩;
  glue_order(r) = o;
  glue_sign(r) = shrinking;
  if (total_shrink[o] ≠ 0) glue_set(r) = fix((−x)/(double) total_shrink[o]);
  else { glue_sign(r) = normal;
    set_glue_ratio_zero(glue_set(r));   /* there's nothing to shrink */
  }
  if ((total_shrink[o] < −x) ∧ (o ≡ normal) ∧ (list_ptr(r) ≠ null)) { 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 665 ⟩;
  }
  else if (o ≡ normal)
    if (list_ptr(r) ≠ null)
      ⟨ Report a tight hbox and goto common_ending, if this box is sufficiently bad 666 ⟩;
  goto end;
}
```

This code is used in section 656.

664. ⟨Determine the shrink order 664⟩ ≡
if $(total\_shrink[filll] \neq 0)$ $o = filll$;
else if $(total\_shrink[fill] \neq 0)$ $o = fill$;
else if $(total\_shrink[fil] \neq 0)$ $o = fil$;
else $o = normal$

This code is used in sections 663, 675, and 795.

665. ⟨Report an overfull hbox and **goto** *common_ending*, if this box is sufficiently bad 665⟩ ≡
if $((-x - total\_shrink[normal] > hfuzz) \lor (hbadness < 100))$ {
  if $((overfull\_rule > 0) \land (-x - total\_shrink[normal] > hfuzz))$ { **while** $(link(q) \neq null)$
    $q = link(q)$;
   $link(q) = new\_rule(\,)$;
   $width(link(q)) = overfull\_rule$;
  }
  $print\_ln(\,)$;
  $print\_nl($"Overfull␣\\hbox␣("$)$;
  $print\_scaled(-x - total\_shrink[normal])$;
  $print($"pt␣too␣wide"$)$;
  goto *common_ending*;
}

This code is used in section 663.

666. ⟨Report a tight hbox and **goto** *common_ending*, if this box is sufficiently bad 666⟩ ≡
{ $last\_badness = badness(-x, total\_shrink[normal])$;
  if $(last\_badness > hbadness)$ { $print\_ln(\,)$;
   $print\_nl($"Tight␣\\hbox␣(badness␣"$)$;
   $print\_int(last\_badness)$;
   goto *common_ending*;
  }
}

This code is used in section 663.

667. 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(__VA_ARGS__, max_dimen)   /* special case of unconstrained depth */
  static pointer vpackage(pointer p, scaled h, small_number m, scaled l)
  { pointer r;   /* the box node that will be returned */
    scaled w, d, x;   /* width, depth, and natural height */
    scaled s;   /* shift amount */
    pointer g;   /* points to a glue specification */
    glue_ord o;   /* order of infinity */
    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 649⟩;
    while (p ≠ null) ⟨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 668⟩;
    width(r) = w;
    if (d > l) { x = x + d − l;
      depth(r) = l;
    }
    else depth(r) = d;
    ⟨Determine the value of height(r) and the appropriate glue setting; then return or goto
        common_ending 671⟩;
  common_ending: ⟨Finish issuing a diagnostic message for an overfull or underfull vbox 674⟩;
  end: return r;
  }
```

668. ⟨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 668⟩ ≡

```
  { if (is_char_node(p)) confusion("vpack");
    else
      switch (type(p)) {
      case hlist_node: case vlist_node: case rule_node: case unset_node:
        ⟨Incorporate box dimensions into the dimensions of the vbox that will contain it 669⟩ break;
      case whatsit_node: ⟨Incorporate a whatsit node into a vbox 1358⟩; break;
      case glue_node: ⟨Incorporate glue into the vertical totals 670⟩ break;
      case kern_node:
        { x = x + d + width(p);
          d = 0;
        } break;
      default: do_nothing;
      }
    p = link(p);
  }
```

This code is used in section 667.

669. ⟨Incorporate box dimensions into the dimensions of the vbox that will contain it 669⟩ ≡

{ $x = x + d + height(p)$;
  $d = depth(p)$;
  if $(type(p) \ge rule\_node)$ $s = 0$; **else** $s = shift\_amount(p)$;
  if $(width(p) + s > w)$ $w = width(p) + s$;
}

This code is used in section 668.

670. ⟨Incorporate glue into the vertical totals 670⟩ ≡

{ $x = x + d$;
  $d = 0$;
  $g = glue\_ptr(p)$;
  $x = x + width(g)$;
  $o = stretch\_order(g)$;
  $total\_stretch[o] = total\_stretch[o] + stretch(g)$;
  $o = shrink\_order(g)$;
  $total\_shrink[o] = total\_shrink[o] + shrink(g)$;
  if $(subtype(p) \ge a\_leaders)$ { $g = leader\_ptr(p)$;
    if $(width(g) > w)$ $w = width(g)$;
  }
}

This code is used in section 668.

671. 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* 671⟩ ≡

if $(m \equiv additional)$ $h = x + h$;
$height(r) = h$;
$x = h - x$; /∗ now x is the excess to be made up ∗/
if $(x \equiv 0)$ { $glue\_sign(r) = normal$;
  $glue\_order(r) = normal$;
  $set\_glue\_ratio\_zero(glue\_set(r))$;
  goto *end*;
}
else if $(x > 0)$ ⟨Determine vertical glue stretch setting, then **return** or **goto** *common_ending* 672⟩
else ⟨Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 675⟩

This code is used in section 667.

672. ⟨Determine vertical glue stretch setting, then **return** or **goto** *common_ending* 672⟩ ≡

{ ⟨Determine the stretch order 658⟩;
  $glue\_order(r) = o$;
  $glue\_sign(r) = stretching$;
  if $(total\_stretch[o] \ne 0)$ $glue\_set(r) = fix(x/(\textbf{double})\ total\_stretch[o])$;
  else { $glue\_sign(r) = normal$;
    $set\_glue\_ratio\_zero(glue\_set(r))$; /∗ there's nothing to stretch ∗/
  }
  if $(o \equiv normal)$
    if $(list\_ptr(r) \ne null)$
      ⟨Report an underfull vbox and **goto** *common_ending*, if this box is sufficiently bad 673⟩;
  goto *end*;
}

This code is used in section 671.

673. ⟨Report an underfull vbox and **goto** *common_ending*, if this box is sufficiently bad 673⟩ ≡

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

This code is used in section 672.

674. ⟨Finish issuing a diagnostic message for an overfull or underfull vbox 674⟩ ≡

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

This code is used in section 667.

675. ⟨Determine vertical glue shrink setting, then **return** or **goto** *common_ending* 675⟩ ≡

```
{ ⟨Determine the shrink order 664⟩;
  glue_order(r) = o;
  glue_sign(r) = shrinking;
  if (total_shrink[o] ≠ 0) glue_set(r) = fix((−x)/(double) total_shrink[o]);
  else { glue_sign(r) = normal;
    set_glue_ratio_zero(glue_set(r));    /* there's nothing to shrink */
  }
  if ((total_shrink[o] < −x) ∧ (o ≡ normal) ∧ (list_ptr(r) ≠ null)) { 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 676⟩;
  }
  else if (o ≡ normal)
    if (list_ptr(r) ≠ null)
      ⟨Report a tight vbox and goto common_ending, if this box is sufficiently bad 677⟩;
  goto end;
}
```

This code is used in section 671.

676. ⟨ Report an overfull vbox and **goto** *common_ending*, if this box is sufficiently bad 676 ⟩ ≡

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

This code is used in section 675.

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

```
{ last_badness = badness(−x, total_shrink[normal]);
   if (last_badness > vbadness) { print_ln( );
      print_nl("Tight␣\\vbox␣(badness␣");
      print_int(last_badness);
      goto common_ending;
   }
}
```

This code is used in section 675.

678. When a box is being appended to the current vertical list, the baselineskip calculation is handled by the *append_to_vlist* routine.

```
static void append_to_vlist(pointer b)
{ scaled d;    /* deficiency of space between baselines */
   pointer p;    /* a new glue node */
   if (prev_depth > ignore_depth) { d = width(baseline_skip) − prev_depth − height(b);
      if (d < line_skip_limit) p = new_param_glue(line_skip_code);
      else { p = new_skip_param(baseline_skip_code);
         width(temp_ptr) = d;    /* temp_ptr ≡ glue_ptr(p) */
      }
      link(tail) = p;
      tail = p;
   }
   link(tail) = b;
   tail = b;
   prev_depth = depth(b);
}
```

679. 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.

680. 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., $q \equiv subscr(p)$), there are several possibilities for the subfields, depending on the *math_type* of q.

math_type(q) $\equiv$ *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) $\equiv$ *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) $\equiv$ *empty* indicates a field with no value (the corresponding attribute of noad p is not present).

math_type(q) $\equiv$ *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) $\equiv$ *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) $\equiv$ *null*. This is not the same as *math_type*(q) $\equiv$ *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*(A) $A+1$ /* the *nucleus* field of a noad */
#**define** *supscr*(A) $A+2$ /* the *supscr* field of a noad */
#**define** *subscr*(A) $A+3$ /* the *subscr* field of a noad */
#**define** *math_type*(A) *link*(A) /* 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 */

681. 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 $fam \equiv 0$, $character \equiv {}^\circ 75$). 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 $subtype \equiv normal$. The only exception is that an *op_noad* might have $subtype \equiv limits$ 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 */

682. 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

$$math\_type(numerator) \equiv math\_type(denominator) \equiv sub\_mlist.$$

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*(A) $A+4$ /* first delimiter field of a noad */
#define *right_delimiter*(A) $A+5$ /* 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*(A) $mem[A].qqqq.b0$ /* *fam* for "small" delimiter */
#define *small_char*(A) $mem[A].qqqq.b1$ /* *character* for "small" delimiter */
#define *large_fam*(A) $mem[A].qqqq.b2$ /* *fam* for "large" delimiter */
#define *large_char*(A) $mem[A].qqqq.b3$ /* *character* for "large" delimiter */
#define *thickness*(A) $width(A)$ /* *thickness* field in a fraction noad */
#define *default_code* °10000000000 /* denotes *default_rule_thickness* */
#define *numerator*(A) $supscr(A)$ /* *numerator* field in a fraction noad */
#define *denominator*(A) $subscr(A)$ /* *denominator* field in a fraction noad */

683. 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⟩ +≡
  static two_halves *empty_field*;
  static four_quarters *null_delimiter*;

684. ⟨Set initial values of key variables 21⟩ +≡
  $empty\_field.rh = empty$;
  $empty\_field.lh = null$;
  $null\_delimiter.b0 = 0$;
  $null\_delimiter.b1 = min\_quarterword$;
  $null\_delimiter.b2 = 0$;
  $null\_delimiter.b3 = min\_quarterword$;

685. The *new_noad* function creates an *ord_noad* that is completely null.

```
static pointer new_noad(void)
{ pointer p;
  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;
  return p;
}
```

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

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

```
#define under_noad (fraction_noad + 1)    /* type of a noad for underlining */
#define over_noad (under_noad + 1)    /* type of a noad for overlining */
#define accent_noad (over_noad + 1)    /* type of a noad for accented subformulas */
#define accent_noad_size 5    /* number of mem words in an accent noad */
#define accent_chr(A) A + 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(A) nucleus(A)    /* delimiter field in left and right noads */
#define middle_noad 1    /* subtype of right noad representing \middle */
#define scripts_allowed(A) (type(A) ≥ ord_noad) ∧ (type(A) < left_noad)
```

687. 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 */

  static pointer new_style(small_number s)   /* create a style node */
  { pointer p;   /* the new node */
    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 */
    return p;
  }
```

688. 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(A) info(A + 1)   /* mlist to be used in display style */
#define text_mlist(A) link(A + 1)   /* mlist to be used in text style */
#define script_mlist(A) info(A + 2)   /* mlist to be used in script style */
#define script_script_mlist(A) link(A + 2)   /* mlist to be used in scriptscript style */

  static pointer new_choice(void)   /* create a choice node */
  { pointer p;   /* the new node */
    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;
    return p;
  }
```

689. 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 689⟩ ≡
case *style_node*: *print_style*(*subtype*(*p*)); **break**;
case *choice_node*: ⟨Display choice node *p* 694⟩ **break**;
case *ord_noad*: **case** *op_noad*: **case** *bin_noad*: **case** *rel_noad*: **case** *open_noad*: **case** *close_noad*:
  case *punct_noad*: **case** *inner_noad*: **case** *radical_noad*: **case** *over_noad*: **case** *under_noad*:
  case *vcenter_noad*: **case** *accent_noad*: **case** *left_noad*: **case** *right_noad*: ⟨Display normal noad *p* 695⟩
  break;
case *fraction_noad*: ⟨Display fraction noad *p* 696⟩ **break**;

This code is used in section 182.

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

⟨Declare procedures needed for displaying the elements of mlists 690⟩ ≡
```
static void print_fam_and_char(pointer p)    /* prints family and character */
{ print_esc("fam");
  print_int(fam(p));
  print_char(' ');
  print_ASCII(qo(character(p)));
}
static void print_delimiter(pointer p)    /* prints a delimiter as 24-bit hex value */
{ int a;    /* accumulator */
  a = small_fam(p) * 256 + qo(small_char(p));
  a = a * #1000 + large_fam(p) * 256 + qo(large_char(p));
  if (a < 0) print_int(a);    /* this should never happen */
  else print_hex(a);
}
```

See also sections 691 and 693.

This code is used in section 178.

691. 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 690⟩ +≡

```
static void show_info(void);   /* show_node_list(info(temp_ptr)) */

static void print_subsidiary_data(pointer p, ASCII_code c)   /* display a noad field */
{ if (cur_length ≥ depth_threshold) { if (math_type(p) ≠ empty) print("␣[]");
  }
  else { append_char(c);   /* include c in the recursion history */
    temp_ptr = p;   /* prepare for show_info if recursion is needed */
    switch (math_type(p)) {
    case math_char:
      { print_ln();
        print_current_string();
        print_fam_and_char(p);
      } break;
    case sub_box: show_info(); break;   /* recursive call */
    case sub_mlist:
      if (info(p) ≡ null) { print_ln();
        print_current_string();
        print("{}");
      }
      else show_info(); break;   /* recursive call */
    default: do_nothing;   /* empty */
    }
    flush_char;   /* remove c from the recursion history */
  }
}
```

692. 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.)

```
static void show_info(void)   /* the reader will kindly forgive this */
{ show_node_list(info(temp_ptr));
}
```

693. ⟨Declare procedures needed for displaying the elements of mlists 690⟩ +≡

```
static void print_style(int c)
{ switch (c/2) {
  case 0: print_esc("displaystyle"); break;   /* display_style ≡ 0 */
  case 1: print_esc("textstyle"); break;   /* text_style ≡ 2 */
  case 2: print_esc("scriptstyle"); break;   /* script_style ≡ 4 */
  case 3: print_esc("scriptscriptstyle"); break;   /* script_script_style ≡ 6 */
  default: print("Unknown␣style!");
  }
}
```

694. ⟨ Display choice node p 694 ⟩ ≡

```
{ 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;
}
```

This code is used in section 689.

695. ⟨ Display normal noad *p* 695 ⟩ ≡

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

This code is used in section 689.

696. ⟨ Display fraction noad p 696 ⟩ ≡
{ *print_esc*("fraction,␣thickness␣");
  if (*thickness*(p) ≡ *default_code*) *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*)) { *print*(",␣left-delimiter␣");
    print_delimiter(*left_delimiter*(p));
  }
  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*)) { *print*(",␣right-delimiter␣");
    print_delimiter(*right_delimiter*(p));
  }
  print_subsidiary_data(*numerator*(p), '\\');
  print_subsidiary_data(*denominator*(p), '/');
}

This code is used in section 689.

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

⟨ Cases of *flush_node_list* that arise in mlists only 697 ⟩ ≡
case *style_node*:
  { *free_node*(*p*, *style_node_size*);
    goto *done*;
  }
case *choice_node*:
  { *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*;
  }
case *ord_noad*: **case** *op_noad*: **case** *bin_noad*: **case** *rel_noad*: **case** *open_noad*: **case** *close_noad*:
    case *punct_noad*: **case** *inner_noad*: **case** *radical_noad*: **case** *over_noad*: **case** *under_noad*:
    case *vcenter_noad*: **case** *accent_noad*:
  { **if** (*math_type*(*nucleus*(*p*)) $\geq$ *sub_box*) *flush_node_list*(*info*(*nucleus*(*p*)));
    if (*math_type*(*supscr*(*p*)) $\geq$ *sub_box*) *flush_node_list*(*info*(*supscr*(*p*)));
    if (*math_type*(*subscr*(*p*)) $\geq$ *sub_box*) *flush_node_list*(*info*(*subscr*(*p*)));
    if (*type*(*p*) $\equiv$ *radical_noad*) *free_node*(*p*, *radical_noad_size*);
    else if (*type*(*p*) $\equiv$ *accent_noad*) *free_node*(*p*, *accent_noad_size*);
    else *free_node*(*p*, *noad_size*);
    goto *done*;
  }
case *left_noad*: **case** *right_noad*:
  { *free_node*(*p*, *noad_size*);
    goto *done*;
  }
case *fraction_noad*:
  { *flush_node_list*(*info*(*numerator*(*p*)));
    flush_node_list(*info*(*denominator*(*p*)));
    free_node(*p*, *fraction_noad_size*);
    goto *done*;
  }

This code is used in section 201.

698. 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 55⟩ +≡

```
  static void print_size(int s)
  { if (s ≡ text_size) print_esc("textfont");
    else if (s ≡ script_size) print_esc("scriptfont");
    else print_esc("scriptscriptfont");
  }
```

699. 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(A) fam_fnt(2 + A) ] ] . sc
#define mathsy(A) font_info [ A + 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
```

700. 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(A) font_info[A + 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
```

701. 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(A) 2 * (A/2) + cramped   /* cramp the style */
#define sub_style(A) 2 * (A/4) + script_style + cramped   /* smaller and cramped */
#define sup_style(A) 2 * (A/4) + script_style + (A % 2)   /* smaller */
#define num_style(A) A + 2 - 2 * (A/6)   /* smaller unless already script-script */
#define denom_style(A) 2 * (A/2) + cramped + 2 - 2 * (A/6)   /* smaller, cramped */
```

702. 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* 702⟩ ≡

```
{ if (cur_style < script_style) cur_size = text_size;
  else cur_size = 16 * ((cur_style - text_style)/2);
  cur_mu = x_over_n(math_quad(cur_size), 18);
}
```

This code is used in sections 719, 725, 726, 729, 753, 759, 761, and 762.

703. 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.

```
static pointer fraction_rule(scaled t)   /* construct the bar for a fraction */
{ pointer p;   /* the new node */
  p = new_rule();
  height(p) = t;
  depth(p) = 0;
  return p;
}
```

704. 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.

```
static pointer overbar(pointer b, scaled k, scaled t)
{ pointer p, q;   /* nodes being constructed */
  p = new_kern(k);
  link(p) = b;
  q = fraction_rule(t);
  link(q) = p;
  p = new_kern(t);
  link(p) = q;
  return vpack(p, natural);
}
```

705. 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* 708⟩

```
static pointer var_delimiter(pointer d, small_number s, scaled v)
{ pointer b;     /* the box that will be constructed */
  internal_font_number f, g;     /* best-so-far and tentative font codes */
  quarterword c, x, y;     /* best-so-far and tentative character codes */
  int m, n;     /* the number of extensible pieces */
  scaled u;     /* height-plus-depth of a tentative character */
  scaled w;     /* largest height-plus-depth so far */
  four_quarters q;     /* character info */
  eight_bits hd;     /* height-depth byte */
  four_quarters r;     /* extensible pieces */
  small_number z;     /* runs through font family members */
  bool large_attempt;     /* are we trying the "large" variant? */
  f = null_font;
  w = 0;
  large_attempt = false;
  z = small_fam(d);
  x = small_char(d);
  loop { ⟨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 706⟩;
    if (large_attempt) goto found;     /* there were none large enough */
    large_attempt = true;
    z = large_fam(d);
    x = large_char(d);
  }
found:
  if (f ≠ null_font) ⟨Make variable b point to a box for (f, c) 709⟩;
  else { b = new_null_box();
    width(b) = null_delimiter_space;     /* use this width if no delimiter was found */
  }
  shift_amount(b) = half(height(b) − depth(b)) − axis_height(s);
  return b;
}
```

706. 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 706 ⟩ ≡

if $((z \neq 0) \lor (x \neq min\_quarterword))$ { $z = z + s + 16$;
  do { $z = z - 16$;
    $g = fam\_fnt(z)$;
    if $(g \neq null\_font)$ ⟨ 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 707 ⟩;
  } **while** $(\neg(z < 16))$;
}

This code is used in section 705.

707. ⟨ 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 707 ⟩ ≡

{ $y = x$;
  if $((qo(y) \geq font\_bc[g]) \land (qo(y) \leq font\_ec[g]))$ { *resume*: $q = char\_info(g, y)$;
    if $(char\_exists(q))$ { **if** $(char\_tag(q) \equiv ext\_tag)$ { $f = g$;
        $c = y$;
        goto *found*;
      }
      $hd = height\_depth(q)$;
      $u = char\_height(g, hd) + char\_depth(g, hd)$;
      if $(u > w)$ { $f = g$;
        $c = y$;
        $w = u$;
        if $(u \geq v)$ **goto** *found*;
      }
      if $(char\_tag(q) \equiv list\_tag)$ { $y = rem\_byte(q)$;
        goto *resume*;
      }
    }
  }
}

This code is used in section 706.

708. 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* 708 ⟩ ≡

```
static pointer char_box(internal_font_number f, quarterword c)
{ four_quarters q;
  eight_bits hd;    /* height_depth byte */
  pointer b, p;     /* the new box and its character node */
  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;
  return b;
}
```

See also sections 710 and 711.
This code is used in section 705.

709. 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) 709 ⟩ ≡

```
if (char_tag(q) ≡ ext_tag)
  ⟨ Construct an extensible character in a new box b, using recipe rem_byte(q) and font f 712 ⟩
else b = char_box(f, c)
```

This code is used in section 705.

710. 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* 708 ⟩ +≡

```
static void stack_into_box(pointer b, internal_font_number f, quarterword c)
{ pointer p;    /* new node placed into b */
  p = char_box(f, c);
  link(p) = list_ptr(b);
  list_ptr(b) = p;
  height(b) = height(p);
}
```

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

⟨ Declare subprocedures for *var_delimiter* 708 ⟩ +≡

```
static scaled height_plus_depth(internal_font_number f, quarterword c)
{ four_quarters q;
  eight_bits hd;    /* height_depth byte */
  q = char_info(f, c);
  hd = height_depth(q);
  return char_height(f, hd) + char_depth(f, hd);
}
```

712. ⟨ Construct an extensible character in a new box b, using recipe *rem_byte*(q) and font f 712 ⟩ ≡

```
{ 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) 713 ⟩;
  c = ext_bot(r);
  if (c ≠ min_quarterword) stack_into_box(b, f, c);
  c = ext_rep(r);
  for (m = 1; m ≤ n; m++) stack_into_box(b, f, c);
  c = ext_mid(r);
  if (c ≠ min_quarterword) { stack_into_box(b, f, c);
    c = ext_rep(r);
    for (m = 1; m ≤ n; m++) stack_into_box(b, f, c);
  }
  c = ext_top(r);
  if (c ≠ min_quarterword) stack_into_box(b, f, c);
  depth(b) = w − height(b);
}
```

This code is used in section 709.

713. 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) 713 ⟩ ≡

```
c = ext_rep(r);
u = height_plus_depth(f, c);
w = 0;
q = char_info(f, c);
width(b) = char_width(f, q) + char_italic(f, q);
c = ext_bot(r); if (c ≠ min_quarterword) w = w + height_plus_depth(f, c);
c = ext_mid(r); if (c ≠ min_quarterword) w = w + height_plus_depth(f, c);
c = ext_top(r); if (c ≠ min_quarterword) w = w + height_plus_depth(f, c);
n = 0;
if (u > 0)
  while (w < v) { w = w + u;
    incr(n);
    if (ext_mid(r) ≠ min_quarterword) w = w + u;
  }
```

This code is used in section 712.

714. 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.

```
static pointer rebox(pointer b, scaled w)
{ pointer p;   /* temporary register for list manipulation */
  internal_font_number f;   /* font in a one-character box */
  scaled v;   /* width of a character without italic correction */
  if ((width(b) ≠ w) ∧ (list_ptr(b) ≠ null)) { if (type(b) ≡ vlist_node) b = hpack(b, natural);
    p = list_ptr(b);
    if ((is_char_node(p)) ∧ (link(p) ≡ null)) { f = font(p);
      v = char_width(f, char_info(f, character(p)));
      if (v ≠ width(b)) link(p) = new_kern(width(b) − v);
    }
    free_node(b, box_node_size);
    b = new_glue(ss_glue);
    link(b) = p;
    while (link(p) ≠ null) p = link(p);
    link(p) = new_glue(ss_glue);
    return hpack(b, w, exactly);
  }
  else { width(b) = w;
    return b;
  }
}
```

715. 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(A) nx_plus_y(n, A, xn_over_d(A, f, °200000))

static pointer math_glue(pointer g, scaled m)
{ pointer p;   /* the new glue specification */
  int n;   /* integer part of m */
  scaled f;   /* fraction part of m */
  n = x_over_n(m, °200000);
  f = rem;
  if (f < 0) { decr(n);
    f = f + °200000;
  }
  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) stretch(p) = mu_mult(stretch(g));
  else stretch(p) = stretch(g);
  shrink_order(p) = shrink_order(g);
  if (shrink_order(p) ≡ normal) shrink(p) = mu_mult(shrink(g));
  else shrink(p) = shrink(g);
  return p;
}
```

716. The *math_kern* subroutine removes *mu_glue* from a kern node, given the value of the math unit.

```
static void math_kern(pointer p, scaled m){ int n;    /* integer part of m */
    scaled f;    /* fraction part of m */
    if (subtype(p) ≡ mu_glue) { n = x_over_n(m, °200000);
    f = rem;
    if (f < 0) { decr(n);
      f = f + °200000;
    }
    width(p) = mu_mult(width(p)); subtype(p) = explicit; } }
```

717. Sometimes it is necessary to destroy an mlist. The following subroutine empties the current list, assuming that $abs(mode) \equiv mmode$.

```
static void flush_math(void)
{ flush_node_list(link(head));
  flush_node_list(incompleat_noad);
  link(head) = null;
  tail = head;
  incompleat_noad = null;
}
```

718. 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 ⟩ +≡
static pointer *cur_mlist*; /* beginning of mlist to be translated */
static small_number *cur_style*; /* style code at current place in the list */
static small_number *cur_size*; /* size code corresponding to *cur_style* */
static scaled *cur_mu*; /* the math unit width corresponding to *cur_size* */
static bool *mlist_penalties*; /* should *mlist_to_hlist* insert penalties? */

719. 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.

```
static void mlist_to_hlist(void);

static pointer clean_box(pointer p, small_number s)
{ pointer q;    /* beginning of a list to be boxed */
  small_number save_style;    /* cur_style to be restored */
  pointer x;    /* box to be returned */
  pointer r;    /* temporary pointer */
  switch (math_type(p)) {
  case math_char:
    { cur_mlist = new_noad();
      mem[nucleus(cur_mlist)] = mem[p];
    } break;
  case sub_box:
    { q = info(p);
      goto found;
    }
  case sub_mlist: cur_mlist = info(p); break;
  default:
    { q = new_null_box();
      goto found;
    }
  }
  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 702⟩;
found:
  if (is_char_node(q) ∨ (q ≡ null)) x = hpack(q, natural);
  else if ((link(q) ≡ null) ∧ (type(q) ≤ vlist_node) ∧ (shift_amount(q) ≡ 0)) x = q;
      /* it's already clean */
  else x = hpack(q, natural);
  ⟨Simplify a trivial box 720⟩;
  return x;
}
```

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

⟨ Simplify a trivial box 720 ⟩ ≡
```
q = list_ptr(x);
if (is_char_node(q)) { r = link(q);
  if (r ≠ null)
    if (link(r) ≡ null)
      if (¬is_char_node(r))
        if (type(r) ≡ kern_node)    /* unneeded italic correction */
        { free_node(r, small_node_size);
          link(q) = null;
        }
}
```
This code is used in section 719.

721. 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*.

```
static void fetch(pointer a)    /* unpack the math_char field a */
{ cur_c = character(a);
  cur_f = fam_fnt(fam(a) + cur_size);
  if (cur_f ≡ null_font) ⟨ Complain about an undefined family and set cur_i null 722 ⟩
  else { if ((qo(cur_c) ≥ font_bc[cur_f]) ∧ (qo(cur_c) ≤ font_ec[cur_f]))
        cur_i = char_info(cur_f, cur_c);
    else cur_i = null_character;
    if (¬(char_exists(cur_i))) { char_warning(cur_f, qo(cur_c));
      math_type(a) = empty;
      cur_i = null_character;
    }
  }
}
```

722. ⟨ Complain about an undefined family and set *cur_i* null 722 ⟩ ≡
```
{ 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;
}
```
This code is used in section 721.

723. The outputs of *fetch* are placed in global variables.

⟨ Global variables 13 ⟩ +≡
  static internal_font_number *cur_f*; /* the *font* field of a *math_char* */
  static quarterword *cur_c*; /* the *character* field of a *math_char* */
  static four_quarters *cur_i*; /* the *char_info* of a *math_char*, or a lig/kern instruction */

724. 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*(*A*) *mem*[*nucleus*(*A*)].*i* /* the translation of an mlist */

725. Here is the overall plan of *mlist_to_hlist*, and the list of its local variables.

⟨ Declare math construction procedures 733 ⟩

```
static void mlist_to_hlist(void)
{ pointer mlist;   /* beginning of the given list */
  bool penalties;   /* should penalty nodes be inserted? */
  small_number style;   /* the given style */
  small_number save_style;   /* holds cur_style during recursion */
  pointer q;   /* runs through the mlist */
  pointer r;   /* the most recent noad preceding q */
  small_number r_type;   /* the type of noad r, or op_noad if r ≡ null */
  small_number t;   /* the effective type of noad q during the second pass */
  pointer p, x, y, z;   /* temporary registers for list construction */
  int pen;   /* a penalty to be inserted */
  small_number s;   /* the size of a noad to be deleted */
  scaled max_h, max_d;   /* maximum height and depth of the list translated so far */
  scaled delta;   /* offset between subscript and superscript */

  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 702 ⟩;
  while (q ≠ null) ⟨ 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 726 ⟩;
  ⟨ Convert a final bin_noad to an ord_noad 728 ⟩;
  ⟨ Make a second pass over the mlist, removing all noads and inserting the proper spacing and
      penalties 759 ⟩;
}
```

726. 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 726 ⟩ ≡
{ ⟨ 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 727 ⟩;
$check\_dimensions$: $z = hpack(new\_hlist(q), natural)$;
if $(height(z) > max\_h)$ $max\_h = height(z)$;
if $(depth(z) > max\_d)$ $max\_d = depth(z)$;
$free\_node(z, box\_node\_size)$;
$done\_with\_noad$: $r = q$;
$r\_type = type(r)$;
if $(r\_type \equiv right\_noad)$ { $r\_type = left\_noad$;
$cur\_style = style$;
⟨ Set up the values of cur_size and cur_mu, based on cur_style 702 ⟩;
}
$done\_with\_node$: $q = link(q)$;
}

This code is used in section 725.

727. 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 727 ⟩ ≡
$reswitch$: $delta = 0$;
switch $(type(q))$ {
case bin_noad:
switch (r_type) {
case bin_noad: **case** op_noad: **case** rel_noad: **case** $open\_noad$: **case** $punct\_noad$: **case** $left\_noad$:
{ $type(q) = ord\_noad$;
goto $reswitch$;
}
default: $do\_nothing$;
} **break**;
case rel_noad: **case** $close\_noad$: **case** $punct\_noad$: **case** $right\_noad$:
{
⟨ Convert a final bin_noad to an ord_noad 728 ⟩;
if $(type(q) \equiv right\_noad)$ **goto** $done\_with\_noad$;
} **break**;
⟨ Cases for noads that can follow a bin_noad 732 ⟩
⟨ Cases for nodes that can appear in an mlist, after which we **goto** $done\_with\_node$ 729 ⟩
default: $confusion($"mlist1"$)$;
}
⟨ Convert $nucleus(q)$ to an hlist and attach the sub/superscripts 753 ⟩

This code is used in section 726.

728. ⟨ Convert a final bin_noad to an ord_noad 728 ⟩ ≡
if $(r\_type \equiv bin\_noad)$ $type(r) = ord\_noad$

This code is used in sections 725 and 727.

729. ⟨ Cases for nodes that can appear in an mlist, after which we **goto** *done_with_node* 729 ⟩ ≡

```
case style_node:
  { cur_style = subtype(q);
    ⟨ Set up the values of cur_size and cur_mu, based on cur_style 702 ⟩;
    goto done_with_node;
  }
case choice_node:
  ⟨ Change this node to a style node followed by the correct choice, then goto done_with_node 730 ⟩
case ins_node: case mark_node: case adjust_node: case whatsit_node: case penalty_node:
  case disc_node: goto done_with_node;
case rule_node:
  { if (height(q) > max_h) max_h = height(q);
    if (depth(q) > max_d) max_d = depth(q);
    goto done_with_node;
  }
case glue_node:
  { ⟨ Convert math glue to ordinary glue 731 ⟩;
    goto done_with_node;
  }
case kern_node:
  { math_kern(q, cur_mu);
    goto done_with_node;
  }
```

This code is used in section 727.

730. #**define** *choose_mlist*(*A*)

```
        { p = A(q);
          A(q) = null; }
```

⟨ Change this node to a style node followed by the correct choice, then **goto** *done_with_node* 730 ⟩ ≡

```
  { switch (cur_style/2) {
    case 0: choose_mlist(display_mlist) break;      /* display_style ≡ 0 */
    case 1: choose_mlist(text_mlist) break;      /* text_style ≡ 2 */
    case 2: choose_mlist(script_mlist) break;      /* script_style ≡ 4 */
    case 3: choose_mlist(script_script_mlist);      /* script_script_style ≡ 6 */
    }      /* 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) { z = link(q);
      link(q) = p;
      while (link(p) ≠ null) p = link(p);
      link(p) = z;
    }
    goto done_with_node;
  }
```

This code is used in section 729.

731. Conditional math glue ('\nonscript') results in a *glue_node* pointing to *zero_glue*, with $subtype(q) \equiv cond\_math\_glue$; 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 731⟩ ≡
if $(subtype(q) \equiv mu\_glue)$ { $x = glue\_ptr(q)$;
$y = math\_glue(x, cur\_mu)$;
$delete\_glue\_ref(x)$;
$glue\_ptr(q) = y$;
$subtype(q) = normal$;
}
else if $((cur\_size \neq text\_size) \land (subtype(q) \equiv cond\_math\_glue))$ { $p = link(q)$;
if $(p \neq null)$
if $((type(p) \equiv glue\_node) \lor (type(p) \equiv kern\_node))$ { $link(q) = link(p)$;
$link(p) = null$;
$flush\_node\_list(p)$;
}
}

This code is used in section 729.

732. ⟨Cases for noads that can follow a *bin_noad* 732⟩ ≡
case *left_noad*: **goto** *done_with_noad*;
case *fraction_noad*:
{ $make\_fraction(q)$;
goto *check_dimensions*;
}
case *op_noad*:
{ $delta = make\_op(q)$;
if $(subtype(q) \equiv limits)$ **goto** *check_dimensions*;
} **break**;
case *ord_noad*: $make\_ord(q)$; **break**;
case *open_noad*: **case** *inner_noad*: *do_nothing*; **break**;
case *radical_noad*: $make\_radical(q)$; **break**;
case *over_noad*: $make\_over(q)$; **break**;
case *under_noad*: $make\_under(q)$; **break**;
case *accent_noad*: $make\_math\_accent(q)$; **break**;
case *vcenter_noad*: $make\_vcenter(q)$; **break**;

This code is used in section 727.

733. 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 733⟩ ≡
static void *make_over*(**pointer** *q*)
{ $info(nucleus(q)) =$
$overbar(clean\_box(nucleus(q), cramped\_style(cur\_style)),$
$3 * default\_rule\_thickness, default\_rule\_thickness)$;
$math\_type(nucleus(q)) = sub\_box$;
}

See also sections 734, 735, 736, 737, 742, 748, 751, 755, and 761.

This code is used in section 725.

734. ⟨ Declare math construction procedures 733 ⟩ +≡

```
static void make_under(pointer q)
{ pointer p, x, y;   /* temporary registers for box construction */
  scaled delta;   /* overall height plus depth */
  x = clean_box(nucleus(q), cur_style);
  p = new_kern(3 * default_rule_thickness);
  link(x) = p;
  link(p) = fraction_rule(default_rule_thickness);
  y = vpack(x, natural);
  delta = height(y) + depth(y) + default_rule_thickness;
  height(y) = height(x);
  depth(y) = delta − height(y);
  info(nucleus(q)) = y;
  math_type(nucleus(q)) = sub_box;
}
```

735. ⟨ Declare math construction procedures 733 ⟩ +≡

```
static void make_vcenter(pointer q)
{ pointer v;   /* the box that should be centered vertically */
  scaled delta;   /* its height plus depth */
  v = info(nucleus(q));
  if (type(v) ≠ vlist_node) confusion("vcenter");
  delta = height(v) + depth(v);
  height(v) = axis_height(cur_size) + half(delta);
  depth(v) = delta − height(v);
}
```

736. 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 733 ⟩ +≡

```
static void make_radical(pointer q)
{ pointer x, y;   /* temporary registers for box construction */
  scaled delta, clr;   /* dimensions involved in the calculation */
  x = clean_box(nucleus(q), cramped_style(cur_style));
  if (cur_style < text_style)   /* display style */
    clr = default_rule_thickness + (abs(math_x_height(cur_size))/4);
  else { clr = default_rule_thickness;
    clr = clr + (abs(clr)/4);
  }
  y = var_delimiter(left_delimiter(q), cur_size, height(x) + depth(x) + clr + default_rule_thickness);
  delta = depth(y) − (height(x) + depth(x) + clr);
  if (delta > 0) clr = clr + half(delta);   /* increase the actual clearance */
  shift_amount(y) = −(height(x) + clr);
  link(y) = overbar(x, clr, height(y));
  info(nucleus(q)) = hpack(y, natural);
  math_type(nucleus(q)) = sub_box;
}
```

737. 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 733⟩ +≡
static void *make_math_accent*(**pointer** q)
{ **pointer** p, x, y; /∗ temporary registers for box construction ∗/
  int a; /∗ address of lig/kern instruction ∗/
  quarterword c; /∗ accent character ∗/
  internal_font_number f; /∗ its font ∗/
  four_quarters i; /∗ its *char_info* ∗/
  scaled s; /∗ amount to skew the accent to the right ∗/
  scaled h; /∗ height of character being accented ∗/
  scaled *delta*; /∗ space to remove between accent and accentee ∗/
  scaled w; /∗ width of the accentee, not including sub/superscripts ∗/

  fetch(*accent_chr*(q));
  if (*char_exists*(*cur_i*)) { i = *cur_i*;
    c = *cur_c*;
    f = *cur_f*;
    ⟨Compute the amount of skew 740⟩;
    x = *clean_box*(*nucleus*(q), *cramped_style*(*cur_style*));
    w = *width*(x);
    h = *height*(x);
    ⟨Switch to a larger accent if available and appropriate 739⟩;
    if (h < *x_height*(f)) *delta* = h; **else** *delta* = *x_height*(f);
    if ((*math_type*(*supscr*(q)) ≠ *empty*) ∨ (*math_type*(*subscr*(q)) ≠ *empty*))
      if (*math_type*(*nucleus*(q)) ≡ *math_char*) ⟨Swap the subscript and superscript into box x 741⟩;
    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) ⟨Make the height of box y equal to h 738⟩;
    info(*nucleus*(q)) = y;
    math_type(*nucleus*(q)) = *sub_box*;
  }
}

738. ⟨Make the height of box y equal to h 738⟩ ≡
{ p = *new_kern*(h − *height*(y));
  link(p) = *list_ptr*(y);
  list_ptr(y) = p;
  height(y) = h;
}

This code is used in section 737.

739. ⟨Switch to a larger accent if available and appropriate 739⟩ ≡
loop { **if** $(char\_tag(i) \neq list\_tag)$ **goto** *done*;
  $y = rem\_byte(i)$;
  $i = char\_info(f, y)$;
  if $(\neg char\_exists(i))$ **goto** *done*;
  if $(char\_width(f, i) > w)$ **goto** *done*;
  $c = y$;
}
done:

This code is used in section 737.

740. ⟨Compute the amount of skew 740⟩ ≡
$s = 0$;
if $(math\_type(nucleus(q)) \equiv math\_char)$ { $fetch(nucleus(q))$;
  if $(char\_tag(cur\_i) \equiv lig\_tag)$ { $a = lig\_kern\_start(cur\_f, cur\_i)$;
    $cur\_i = font\_info[a].qqqq$;
    if $(skip\_byte(cur\_i) > stop\_flag)$ { $a = lig\_kern\_restart(cur\_f, cur\_i)$;
      $cur\_i = font\_info[a].qqqq$;
    }
    loop { **if** $(qo(next\_char(cur\_i)) \equiv skew\_char[cur\_f])$ { **if** $(op\_byte(cur\_i) \geq kern\_flag)$
          if $(skip\_byte(cur\_i) \leq stop\_flag)$ $s = char\_kern(cur\_f, cur\_i)$;
        goto *done1*;
      }
      if $(skip\_byte(cur\_i) \geq stop\_flag)$ **goto** *done1*;
      $a = a + qo(skip\_byte(cur\_i)) + 1$;
      $cur\_i = font\_info[a].qqqq$;
    }
  }
}
done1:

This code is used in section 737.

741. ⟨Swap the subscript and superscript into box x 741⟩ ≡
{ $flush\_node\_list(x)$;
  $x = new\_noad()$;
  $mem[nucleus(x)] = mem[nucleus(q)]$;
  $mem[supscr(x)] = mem[supscr(q)]$;
  $mem[subscr(x)] = mem[subscr(q)]$;
  $mem[supscr(q)].hh = empty\_field$;
  $mem[subscr(q)].hh = empty\_field$;
  $math\_type(nucleus(q)) = sub\_mlist$;
  $info(nucleus(q)) = x$;
  $x = clean\_box(nucleus(q), cur\_style)$;
  $delta = delta + height(x) - h$;
  $h = height(x)$;
}

This code is used in section 737.

742. 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 733 ⟩ +≡

```
static void make_fraction(pointer q)
{ pointer p, v, x, y, z;   /* temporary registers for box construction */
  scaled delta, delta1, delta2, shift_up, shift_down, clr;   /* dimensions for box calculations */
  if (thickness(q) ≡ default_code) 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 743 ⟩;
  if (thickness(q) ≡ 0) ⟨ Adjust shift_up and shift_down for the case of no fraction line 744 ⟩
  else ⟨ Adjust shift_up and shift_down for the case of a fraction line 745 ⟩;
  ⟨ Construct a vlist box for the fraction, according to shift_up and shift_down 746 ⟩;
  ⟨ Put the fraction into a box with its delimiters, and make new_hlist(q) point to it 747 ⟩;
}
```

743. ⟨ 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 743 ⟩ ≡

```
x = clean_box(numerator(q), num_style(cur_style));
z = clean_box(denominator(q), denom_style(cur_style));
if (width(x) < width(z)) x = rebox(x, width(z));
else z = rebox(z, width(x));
if (cur_style < text_style)   /* display style */
{ shift_up = num1(cur_size);
  shift_down = denom1(cur_size);
}
else { shift_down = denom2(cur_size);
  if (thickness(q) ≠ 0) shift_up = num2(cur_size);
  else shift_up = num3(cur_size);
}
```

This code is used in section 742.

744. 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 744 ⟩ ≡

```
{ if (cur_style < text_style) 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) { shift_up = shift_up + delta;
    shift_down = shift_down + delta;
  }
}
```

This code is used in section 742.

745. 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 745 ⟩ ≡

```
{ if (cur_style < text_style) 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) shift_up = shift_up + delta1;
  if (delta2 > 0) shift_down = shift_down + delta2;
}
```

This code is used in section 742.

746. ⟨ Construct a vlist box for the fraction, according to *shift_up* and *shift_down* 746 ⟩ ≡

```
v = 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) { p = new_kern((shift_up - depth(x)) - (height(z) - shift_down));
  link(p) = z;
}
else { y = fraction_rule(thickness(q));
  p = new_kern((axis_height(cur_size) - delta) -
      (height(z) - shift_down));
  link(y) = p;
  link(p) = z;
  p = new_kern((shift_up - depth(x)) - (axis_height(cur_size) + delta));
  link(p) = y;
}
link(x) = p; list_ptr(v) = x
```

This code is used in section 742.

747. ⟨ Put the fraction into a box with its delimiters, and make *new_hlist*(*q*) point to it 747 ⟩ ≡

```
if (cur_style < text_style) 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 742.

748. 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 733 ⟩ +≡

```
static scaled make_op(pointer q)
{ scaled delta;   /* offset between subscript and superscript */
  pointer p, v, x, y, z;   /* temporary registers for box construction */
  quarterword c; four_quarters i;   /* registers for character examination */
  scaled shift_up, shift_down;   /* dimensions for box calculation */
  if ((subtype(q) ≡ normal) ∧ (cur_style < text_style)) subtype(q) = limits;
  if (math_type(nucleus(q)) ≡ math_char) { fetch(nucleus(q));
    if ((cur_style < text_style) ∧ (char_tag(cur_i) ≡ list_tag))   /* make it larger */
    { c = rem_byte(cur_i);
      i = char_info(cur_f, c);
      if (char_exists(i)) { cur_c = c;
        cur_i = i;
        character(nucleus(q)) = c;
      }
    }
    delta = char_italic(cur_f, cur_i);
    x = clean_box(nucleus(q), cur_style);
    if ((math_type(subscr(q)) ≠ empty) ∧ (subtype(q) ≠ limits)) 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;
  }
  else delta = 0;
  if (subtype(q) ≡ limits) ⟨ Construct a box with limits above and below it, skewed by delta 749 ⟩;
  return delta;
}
```

749. 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* 749 ⟩ ≡
{ $x = clean\_box(supscr(q), sup\_style(cur\_style))$;
  $y = clean\_box(nucleus(q), cur\_style)$;
  $z = clean\_box(subscr(q), sub\_style(cur\_style))$;
  $v = new\_null\_box()$;
  $type(v) = vlist\_node$;
  $width(v) = width(y)$;
  if $(width(x) > width(v))$ $width(v) = width(x)$;
  if $(width(z) > width(v))$ $width(v) = width(z)$;
  $x = rebox(x, width(v))$;
  $y = rebox(y, width(v))$;
  $z = rebox(z, width(v))$;
  $shift\_amount(x) = half(delta)$;
  $shift\_amount(z) = -shift\_amount(x)$;
  $height(v) = height(y)$;
  $depth(v) = depth(y)$;
  ⟨ Attach the limits to y and adjust $height(v)$, $depth(v)$ to account for their presence 750 ⟩;
  $new\_hlist(q) = v$;
}

This code is used in section 748.

750. 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 750 ⟩ ≡
if $(math\_type(supscr(q)) \equiv empty)$ { $free\_node(x, box\_node\_size)$;
  $list\_ptr(v) = y$;
}
else { $shift\_up = big\_op\_spacing3 - depth(x)$;
  if $(shift\_up < big\_op\_spacing1)$ $shift\_up = big\_op\_spacing1$;
  $p = new\_kern(shift\_up)$;
  $link(p) = y$;
  $link(x) = p$;
  $p = new\_kern(big\_op\_spacing5)$;
  $link(p) = x$;
  $list\_ptr(v) = p$;
  $height(v) = height(v) + big\_op\_spacing5 + height(x) + depth(x) + shift\_up$;
}
if $(math\_type(subscr(q)) \equiv empty)$ $free\_node(z, box\_node\_size)$;
else { $shift\_down = big\_op\_spacing4 - height(z)$;
  if $(shift\_down < big\_op\_spacing2)$ $shift\_down = big\_op\_spacing2$;
  $p = new\_kern(shift\_down)$;
  $link(y) = p$;
  $link(p) = z$;
  $p = new\_kern(big\_op\_spacing5)$;
  $link(z) = p$;
  $depth(v) = depth(v) + big\_op\_spacing5 + height(z) + depth(z) + shift\_down$;
}

This code is used in section 749.

751. 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 $space \equiv 0$).

No boundary characters enter into these ligatures.

```
⟨Declare math construction procedures 733⟩ +≡
  static void make_ord(pointer q)
  { int a;     /* address of lig/kern instruction */
    pointer p, r;     /* temporary registers for list manipulation */
  restart:
    if (math_type(subscr(q)) ≡ empty)
      if (math_type(supscr(q)) ≡ empty)
        if (math_type(nucleus(q)) ≡ math_char) { p = link(q);
          if (p ≠ null)
            if ((type(p) ≥ ord_noad) ∧ (type(p) ≤ punct_noad))
              if (math_type(nucleus(p)) ≡ math_char)
                if (fam(nucleus(p)) ≡ fam(nucleus(q))) { math_type(nucleus(q)) = math_text_char;
                  fetch(nucleus(q));
                  if (char_tag(cur_i) ≡ lig_tag) { 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) { a = lig_kern_restart(cur_f, cur_i);
                      cur_i = font_info[a].qqqq;
                    }
                    loop { ⟨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 752⟩;
                      if (skip_byte(cur_i) ≥ stop_flag) return;
                      a = a + qo(skip_byte(cur_i)) + 1;
                      cur_i = font_info[a].qqqq;
                    }
                  }
                }
        }
  }
```

752. 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* 752⟩ ≡

```
if (next_char(cur_i) ≡ cur_c)
  if (skip_byte(cur_i) ≤ stop_flag)
    if (op_byte(cur_i) ≥ kern_flag) { p = new_kern(char_kern(cur_f, cur_i));
      link(p) = link(q);
      link(q) = p;
      return;
    }
    else { check_interrupt;      /* allow a way out of infinite ligature loop */
      switch (op_byte(cur_i)) {
      case qi(1): case qi(5): character(nucleus(q)) = rem_byte(cur_i); break;      /* =:|, =:|> */
      case qi(2): case qi(6): character(nucleus(p)) = rem_byte(cur_i); break;      /* |=:, |=:> */
      case qi(3): case qi(7): case qi(11):
        { 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)) math_type(nucleus(r)) = math_char;
          else math_type(nucleus(r)) = math_text_char;      /* prevent combination */
        } break;
      default:
        { 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);
        }
      }
      if (op_byte(cur_i) > qi(3)) return;
      math_type(nucleus(q)) = math_char;
      goto restart;
    }
```

This code is used in section 751.

753. 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 753⟩ ≡

```
switch (math_type(nucleus(q))) {
case math_char: case 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 754⟩ break;
case empty: p = null; break;
case sub_box: p = info(nucleus(q)); break;
case sub_mlist:
  { 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 702⟩;
    p = hpack(link(temp_head), natural);
  } break;
default: confusion("mlist2");
}
new_hlist(q) = p;
if ((math_type(subscr(q)) ≡ empty) ∧ (math_type(supscr(q)) ≡ empty)) goto check_dimensions;
make_scripts(q, delta)
```

This code is used in section 727.

754. ⟨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 754⟩ ≡

```
{ fetch(nucleus(q));
  if (char_exists(cur_i)) { delta = char_italic(cur_f, cur_i);
    p = new_character(cur_f, qo(cur_c));
    if ((math_type(nucleus(q)) ≡ math_text_char) ∧ (space(cur_f) ≠ 0)) delta = 0;
        /* no italic correction in mid-word of text font */
    if ((math_type(subscr(q)) ≡ empty) ∧ (delta ≠ 0)) { link(p) = new_kern(delta);
      delta = 0;
    }
  }
  else p = null;
}
```

This code is used in section 753.

755. 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 733⟩ +≡
```
static void make_scripts(pointer q, scaled delta)
{ pointer p, x, y, z;    /* temporary registers for box construction */
  scaled shift_up, shift_down, clr;    /* dimensions in the calculation */
  small_number t;    /* subsidiary size code */

  p = new_hlist(q);
  if (is_char_node(p)) { shift_up = 0;
    shift_down = 0;
  }
  else { z = hpack(p, natural);
    if (cur_style < script_style) 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);
  }
  if (math_type(supscr(q)) ≡ empty) ⟨Construct a subscript box x when there is no superscript 756⟩
  else { ⟨Construct a superscript box x 757⟩;
    if (math_type(subscr(q)) ≡ empty) shift_amount(x) = −shift_up;
    else ⟨Construct a sub/superscript combination box x, with the superscript offset by delta 758⟩;
  }
  if (new_hlist(q) ≡ null) new_hlist(q) = x;
  else { p = new_hlist(q);
    while (link(p) ≠ null) p = link(p);
    link(p) = x;
  }
}
```

756. 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 756⟩ ≡
```
{ x = clean_box(subscr(q), sub_style(cur_style));
  width(x) = width(x) + script_space;
  if (shift_down < sub1(cur_size)) shift_down = sub1(cur_size);
  clr = height(x) − (abs(math_x_height(cur_size) * 4)/5);
  if (shift_down < clr) shift_down = clr;
  shift_amount(x) = shift_down;
}
```
This code is used in section 755.

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

⟨ Construct a superscript box x 757 ⟩ ≡

```
{ x = clean_box(supscr(q), sup_style(cur_style));
  width(x) = width(x) + script_space;
  if (odd(cur_style)) clr = sup3(cur_size);
  else if (cur_style < text_style) clr = sup1(cur_size);
  else clr = sup2(cur_size);
  if (shift_up < clr) shift_up = clr;
  clr = depth(x) + (abs(math_x_height(cur_size))/4);
  if (shift_up < clr) shift_up = clr;
}
```

This code is used in section 755.

758. 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* 758 ⟩ ≡

```
{ y = clean_box(subscr(q), sub_style(cur_style));
  width(y) = width(y) + script_space;
  if (shift_down < sub2(cur_size)) shift_down = sub2(cur_size);
  clr = 4 * default_rule_thickness − ((shift_up − depth(x)) − (height(y) − shift_down));
  if (clr > 0) { shift_down = shift_down + clr;
    clr = (abs(math_x_height(cur_size) * 4)/5) − (shift_up − depth(x));
    if (clr > 0) { shift_up = shift_up + clr;
      shift_down = shift_down − clr;
    }
  }
  shift_amount(x) = delta;    /* superscript is delta to the right of the subscript */
  p = new_kern((shift_up − depth(x)) − (height(y) − shift_down));
  link(x) = p;
  link(p) = y;
  x = vpack(x, natural);
  shift_amount(x) = shift_down;
}
```

This code is used in section 755.

759. 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 759⟩ ≡
  $p = temp\_head$;
  $link(p) = null$;
  $q = mlist$;
  $r\_type = 0$;
  $cur\_style = style$;
  ⟨Set up the values of *cur_size* and *cur_mu*, based on *cur_style* 702⟩;
  while $(q \neq null)$ { ⟨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 760⟩;
    ⟨Append inter-element spacing based on *r_type* and t 765⟩;
    ⟨Append any *new_hlist* entries for q, and any appropriate penalties 766⟩;
    if $(type(q) \equiv right\_noad)$ $t = open\_noad$;
    $r\_type = t$;
  delete_q: $r = q$;
    $q = link(q)$;
    $free\_node(r, s)$;
  done: ;
  }

This code is used in section 725.

760. 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 760⟩ ≡

```
t = ord_noad;
s = noad_size;
pen = inf_penalty;
switch (type(q)) {
case op_noad: case open_noad: case close_noad: case punct_noad: case inner_noad: t = type(q);
  break;
case bin_noad:
  { t = bin_noad;
    pen = bin_op_penalty;
  } break;
case rel_noad:
  { t = rel_noad;
    pen = rel_penalty;
  } break;
case ord_noad: case vcenter_noad: case over_noad: case under_noad: do_nothing; break;
case radical_noad: s = radical_noad_size; break;
case accent_noad: s = accent_noad_size; break;
case fraction_noad: s = fraction_noad_size; break;
case left_noad: case right_noad: t = make_left_right(q, style, max_d, max_h); break;
case style_node: ⟨Change the current style and goto delete_q 762⟩
case whatsit_node: case penalty_node: case rule_node: case disc_node: case adjust_node:
    case ins_node: case mark_node: case glue_node: case kern_node:
  { link(p) = q;
    p = q;
    q = link(q);
    link(p) = null;
    goto done;
  }
default: confusion("mlist3");
}
```

This code is used in section 759.

761. 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 $right\_noad - left\_noad \equiv close\_noad - open\_noad$.

⟨Declare math construction procedures 733⟩ +≡

```
static small_number make_left_right(pointer q, small_number style, scaled max_d, scaled max_h)
{ scaled delta, delta1, delta2;   /* dimensions used in the calculation */
  cur_style = style;
  ⟨Set up the values of cur_size and cur_mu, based on cur_style 702⟩;
  delta2 = max_d + axis_height(cur_size);
  delta1 = max_h + max_d − delta2;
  if (delta2 > delta1) delta1 = delta2;   /* delta1 is max distance from axis */
  delta = (delta1/500) ∗ delimiter_factor;
  delta2 = delta1 + delta1 − delimiter_shortfall;
  if (delta < delta2) delta = delta2;
  new_hlist(q) = var_delimiter(delimiter(q), cur_size, delta);
  return type(q) − (left_noad − open_noad);   /* open_noad or close_noad */
}
```

762. ⟨Change the current style and **goto** *delete_q* 762⟩ ≡

```
{ cur_style = subtype(q);
  s = style_node_size;
  ⟨Set up the values of cur_size and cur_mu, based on cur_style 702⟩;
  goto delete_q;
}
```

This code is used in section 760.

763. 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"
```

764. ⟨Global variables 13⟩ +≡

```
static const int magic_offset = −9 ∗ ord_noad;   /* used to find inter-element spacing */
```

765. ⟨ Append inter-element spacing based on *r_type* and *t* 765 ⟩ ≡

```
if (r_type > 0)    /* not the first noad */
{ switch (so(math_spacing[r_type * 8 + t + magic_offset])) {
  case '0': x = 0; break;
  case '1':
    if (cur_style < script_style) x = thin_mu_skip_code; else x = 0; break;
  case '2': x = thin_mu_skip_code; break;
  case '3':
    if (cur_style < script_style) x = med_mu_skip_code; else x = 0; break;
  case '4':
    if (cur_style < script_style) x = thick_mu_skip_code; else x = 0; break;
  default: confusion("mlist4");
  }
  if (x ≠ 0) { 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 */
  }
}
```

This code is used in section 759.

766. 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 766 ⟩ ≡

```
if (new_hlist(q) ≠ null) { link(p) = new_hlist(q);
  do { p = link(p);
  } while (¬(link(p) ≡ null));
}
if (penalties)
  if (link(q) ≠ null)
    if (pen < inf_penalty) { r_type = type(link(q));
      if (r_type ≠ penalty_node)
        if (r_type ≠ rel_noad) { z = new_penalty(pen);
          link(p) = z;
          p = z;
        }
    }
```

This code is used in section 759.

767. 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.

768. 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*(A) $mem[A + height\_offset].i$ /* pointer to $\langle u_j \rangle$ token list */
#**define** *v_part*(A) $mem[A + depth\_offset].i$ /* pointer to $\langle v_j \rangle$ token list */
#**define** *extra_info*(A) $info(A + list\_offset)$ /* info to remember during template */

769. 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 ⟩ +≡
static pointer *cur_align*; /∗ current position in preamble list ∗/
static pointer *cur_span*; /∗ start of currently spanned columns in preamble list ∗/
static pointer *cur_loop*; /∗ place to copy when extending a periodic preamble ∗/
static pointer *align_ptr*; /∗ most recently pushed-down alignment stack node ∗/
static pointer *cur_head*, *cur_tail*; /∗ adjustment list pointers ∗/

770. 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*;

771. Alignment stack maintenance is handled by a pair of trivial routines called *push_alignment* and *pop_alignment*.

static void *push_alignment*(**void**)
{ **pointer** *p*; /* the new alignment stack node */
  $p = get\_node(align\_stack\_node\_size)$;
  $link(p) = align\_ptr$;
  $info(p) = cur\_align$;
  $llink(p) = preamble$;
  $rlink(p) = cur\_span$;
  $mem[p+2].i = cur\_loop$;
  $mem[p+3].i = align\_state$;
  $info(p+4) = cur\_head$;
  $link(p+4) = cur\_tail$;
  $align\_ptr = p$;
  $cur\_head = get\_avail()$;
}

static void *pop_alignment*(**void**)
{ **pointer** *p*; /* the top alignment stack node */
  $free\_avail(cur\_head)$;
  $p = align\_ptr$;
  $cur\_tail = link(p+4)$;
  $cur\_head = info(p+4)$;
  $align\_state = mem[p+3].i$;
  $cur\_loop = mem[p+2].i$;
  $cur\_span = rlink(p)$;
  $preamble = llink(p)$;
  $cur\_align = info(p)$;
  $align\_ptr = link(p)$;
  $free\_node(p, align\_stack\_node\_size)$;
}

772. 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*.

773. 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* 781⟩
```
static void align_peek(void);
static void normal_paragraph(void);
static void init_align(void)
{ pointer save_cs_ptr;   /* warning_index value for error messages */
  pointer p;   /* for short-term temporary use */
  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 775⟩;
  push_nest();   /* enter a new semantic level */
  ⟨Change current mode to −vmode for \halign, −hmode for \valign 774⟩;
  scan_spec(align_group, false);
  ⟨Scan the preamble and record it in the preamble list 776⟩;
  new_save_level(align_group);
  if (every_cr ≠ null) begin_token_list(every_cr, every_cr_text);
  align_peek();   /* look for \noalign or \omit */
}
```

774. 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` 774⟩ ≡
```
if (mode ≡ mmode) { mode = −vmode;
  prev_depth = nest[nest_ptr − 2].aux_field.sc;
}
else if (mode > 0) negate(mode)
```
This code is used in section 773.

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

⟨Check for improper alignment in displayed math 775⟩ ≡
```
if ((mode ≡ mmode) ∧ ((tail ≠ head) ∨ (incompleat_noad ≠ null))) { 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();
}
```
This code is used in section 773.

776. ⟨ Scan the preamble and record it in the *preamble* list 776 ⟩ ≡
$preamble = null$;
$cur\_align = align\_head$;
$cur\_loop = null$;
$scanner\_status = aligning$;
$warning\_index = save\_cs\_ptr$;
$align\_state = -1000000$; /* at this point, $cur\_cmd \equiv left\_brace$ */
loop { ⟨ Append the current tabskip glue to the preamble list 777 ⟩;
if ($cur\_cmd \equiv car\_ret$) **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 778 ⟩;
}
done: $scanner\_status = normal$

This code is used in section 773.

777. ⟨ Append the current tabskip glue to the preamble list 777 ⟩ ≡
$link(cur\_align) = new\_param\_glue(tab\_skip\_code)$; $cur\_align = link(cur\_align)$

This code is used in section 776.

778. ⟨ 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 778 ⟩ ≡
⟨ Scan the template $\langle u_j \rangle$, putting the resulting token list in *hold_head* 782 ⟩;
$link(cur\_align) = new\_null\_box()$;
$cur\_align = link(cur\_align)$; /* a new alignrecord */
$info(cur\_align) = end\_span$;
$width(cur\_align) = null\_flag$;
$u\_part(cur\_align) = link(hold\_head)$;
⟨ Scan the template $\langle v_j \rangle$, putting the resulting token list in *hold_head* 783 ⟩;
$v\_part(cur\_align) = link(hold\_head)$

This code is used in section 776.

779. 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 225 ⟩ +≡
primitive("`span`", *tab_mark*, *span_code*);
primitive("`cr`", *car_ret*, *cr_code*);
text(*frozen_cr*) = *text*(*cur_val*);
eqtb[*frozen_cr*] = *eqtb*[*cur_val*];
primitive("`crcr`", *car_ret*, *cr_cr_code*);
text(*frozen_end_template*) = *text*(*frozen_endv*) = *s_no*("`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*;

780. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *tab_mark*:
  if (*chr_code* ≡ *span_code*) *print_esc*("`span`");
  else *chr_cmd*("`alignment␣tab␣character␣`") **break**;
case *car_ret*:
  if (*chr_code* ≡ *cr_code*) *print_esc*("`cr`");
  else *print_esc*("`crcr`"); **break**;

781. 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* 781⟩ ≡

```
static void get_preamble_token(void)
{ restart: get_token();
  while ((cur_chr ≡ span_code) ∧ (cur_cmd ≡ tab_mark)) { get_token();
      /∗ this token will be expanded once ∗/
    if (cur_cmd > max_command) { expand();
      get_token();
    }
  }
  if (cur_cmd ≡ endv) fatal_error("(interwoven␣alignment␣preambles␣are␣not␣allowed)");
  if ((cur_cmd ≡ assign_glue) ∧ (cur_chr ≡ glue_base + tab_skip_code)) { scan_optional_equals();
    scan_glue(glue_val);
    if (global_defs > 0) 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;
  }
}
```

This code is used in section 773.

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

⟨Scan the template $\langle u_j \rangle$, putting the resulting token list in *hold_head* 782⟩ ≡

```
p = hold_head;
link(p) = null;
loop { get_preamble_token();
  if (cur_cmd ≡ mac_param) goto done1;
  if ((cur_cmd ≤ car_ret) ∧ (cur_cmd ≥ tab_mark) ∧ (align_state ≡ −1000000))
    if ((p ≡ hold_head) ∧ (cur_loop ≡ null) ∧ (cur_cmd ≡ tab_mark)) cur_loop = cur_align;
    else { 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;
    }
  else if ((cur_cmd ≠ spacer) ∨ (p ≠ hold_head)) { link(p) = get_avail();
    p = link(p);
    info(p) = cur_tok;
  }
}
done1:
```

This code is used in section 778.

783. ⟨ Scan the template ⟨v_j⟩, putting the resulting token list in *hold_head* 783 ⟩ ≡

```
p = hold_head;
link(p) = null;
loop { resume: get_preamble_token();
  if ((cur_cmd ≤ car_ret) ∧ (cur_cmd ≥ tab_mark) ∧ (align_state ≡ −1000000)) goto done2;
  if (cur_cmd ≡ mac_param) { 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 resume;
  }
  link(p) = get_avail();
  p = link(p);
  info(p) = cur_tok;
}
done2: link(p) = get_avail();
  p = link(p); info(p) = end_template_token     /* put \endtemplate at the end */
```

This code is used in section 778.

784. 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* 784 ⟩ ≡

```
static void align_peek(void)
{ restart: align_state = 1000000;
  do { get_x_or_protected();
  } while (¬(cur_cmd ≠ spacer));
  if (cur_cmd ≡ no_align) { scan_left_brace();
    new_save_level(no_align_group);
    if (mode ≡ −vmode) normal_paragraph();
  }
  else if (cur_cmd ≡ right_brace) fin_align();
  else if ((cur_cmd ≡ car_ret) ∧ (cur_chr ≡ cr_cr_code)) goto restart;     /* ignore \crcr */
  else { init_row();     /* start a new row */
    init_col();     /* start a new column and replace what we peeked at */
  }
}
```

This code is used in section 799.

785. 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 786⟩
    static void init_row(void)
    { push_nest();
      mode = (−hmode − vmode) − mode;
      if (mode ≡ −hmode) 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);
    }
```

786. 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 786⟩ ≡
  static void init_span(pointer p)
  { push_nest();
    if (mode ≡ −hmode) space_factor = 1000;
    else { prev_depth = ignore_depth;
      normal_paragraph();
    }
    cur_span = p;
  }
```

This code is used in section 785.

787. 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.

```
  static void init_col(void)
  { extra_info(cur_align) = cur_cmd;
    if (cur_cmd ≡ omit) align_state = 0;
    else { back_input();
      begin_token_list(u_part(cur_align), u_template);
    }     /∗ now align_state ≡ 1000000 ∗/
  }
```

788. 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* $\equiv 0$, 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* 788 ⟩ ≡
{ **if** ((*scanner_status* ≡ *aligning*) ∨ (*cur_align* ≡ *null*))
    fatal_error(`"(interwoven␣alignment␣preambles␣are␣not␣allowed)"`);
  cur_cmd = *extra_info*(*cur_align*);
  extra_info(*cur_align*) = *cur_chr*;
  if (*cur_cmd* ≡ *omit*) *begin_token_list*(*omit_template*, *v_template*);
  else *begin_token_list*(*v_part*(*cur_align*), *v_template*);
  align_state = 1000000;
  goto *restart*;
}

This code is used in section 341.

789. 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 789 ⟩ ≡
  info(*omit_template*) = *end_template_token*;   /* *link*(*omit_template*) ≡ *null* */

See also sections 796, 819, 980, and 987.

This code is used in section 163.

790. 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.

```
static bool fin_col(void)
{ pointer p;    /* the alignrecord after the current one */
  pointer q, r;    /* temporary pointers for list manipulation */
  pointer s;    /* a new span node */
  pointer u;    /* a new unset box */
  scaled w;    /* natural width */
  glue_ord o;    /* order of infinity */
  halfword n;    /* span counter */
  if (cur_align ≡ null) confusion("endv");
  q = link(cur_align); if (q ≡ null) confusion("endv");
  if (align_state < 500000) fatal_error("(interwoven␣alignment␣preambles␣are␣not␣allowed)");
  p = link(q);
  ⟨If the preamble list has been traversed, check that the row has ended 791⟩;
  if (extra_info(cur_align) ≠ span_code) { unsave();
    new_save_level(align_group);
    ⟨Package an unset box for the current column and record its width 795⟩;
    ⟨Copy the tabskip glue between columns 794⟩;
    if (extra_info(cur_align) ≥ cr_code) { return true;
    }
    init_span(p);
  }
  align_state = 1000000;
  do { get_x_or_protected();
  } while (¬(cur_cmd ≠ spacer));
  cur_align = p;
  init_col();
  return false;
}
```

791. ⟨If the preamble list has been traversed, check that the row has ended 791⟩ ≡

```
  if ((p ≡ null) ∧ (extra_info(cur_align) < cr_code))
    if (cur_loop ≠ null) ⟨Lengthen the preamble periodically 792⟩
    else { 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 ( ) ;
    }
```

This code is used in section 790.

792. ⟨Lengthen the preamble periodically 792⟩ ≡

```
{ 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 793⟩;
  cur_loop = link(cur_loop);
  link(p) = new_glue(glue_ptr(cur_loop));
  subtype(link(p)) = tab_skip_code + 1;
}
```

This code is used in section 791.

793. ⟨Copy the templates from node *cur_loop* into node *p* 793⟩ ≡

```
q = hold_head;
r = u_part(cur_loop);
while (r ≠ null) { link(q) = get_avail();
  q = link(q);
  info(q) = info(r);
  r = link(r);
}
link(q) = null;
u_part(p) = link(hold_head);
q = hold_head;
r = v_part(cur_loop);
while (r ≠ null) { link(q) = get_avail();
  q = link(q);
  info(q) = info(r);
  r = link(r);
}
link(q) = null; v_part(p) = link(hold_head)
```

This code is used in section 792.

794. ⟨Copy the tabskip glue between columns 794⟩ ≡

```
tail_append(new_glue(glue_ptr(link(cur_align)))); subtype(tail) = tab_skip_code + 1
```

This code is used in section 790.

795. ⟨Package an unset box for the current column and record its width 795⟩ ≡

{ **if** ($mode \equiv -hmode$) { $adjust\_tail = cur\_tail$;
    $u = hpack(link(head), natural)$;
    $w = width(u)$;
    $cur\_tail = adjust\_tail$;
    $adjust\_tail = null$;
  }
  else { $u = vpackage(link(head), natural, 0)$;
    $w = height(u)$;
  }
  $n = min\_quarterword$; /∗ this represents a span count of 1 ∗/
  if ($cur\_span \neq cur\_align$) ⟨Update width entry for spanned columns 797⟩
  else if ($w > width(cur\_align)$) $width(cur\_align) = w$;
  $type(u) = unset\_node$;
  $span\_count(u) = n$;
  ⟨Determine the stretch order 658⟩;
  $glue\_order(u) = o$;
  $glue\_stretch(u) = total\_stretch[o]$;
  ⟨Determine the shrink order 664⟩;
  $glue\_sign(u) = o$;
  $glue\_shrink(u) = total\_shrink[o]$;
  $pop\_nest()$;
  $link(tail) = u$;
  $tail = u$;
}

This code is used in section 790.

796. 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 789⟩ +≡
  $link(end\_span) = max\_quarterword + 1$;
  $info(end\_span) = null$;

797. ⟨ Update width entry for spanned columns 797 ⟩ ≡
```
{ q = cur_span;
  do { incr(n);
    q = link(link(q));
  } while (¬(q ≡ cur_align));
  if (n > max_quarterword) confusion("256␣spans");    /* this can happen, but won't */
  q = cur_span;
  while (link(info(q)) < n) q = info(q);
  if (link(info(q)) > n) { s = get_node(span_node_size);
    info(s) = info(q);
    link(s) = n;
    info(q) = s;
    width(s) = w;
  }
  else if (width(info(q)) < w) width(info(q)) = w;
}
```
This code is used in section 795.

798. 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.

```
static void fin_row(void)
{ pointer p;    /* the new unset box */
  if (mode ≡ −hmode) { p = hpack(link(head), natural);
    pop_nest();
    append_to_vlist(p);
    if (cur_head ≠ cur_tail) { link(tail) = link(cur_head);
      tail = cur_tail;
    }
  }
  else { p = vpack(link(head), natural);
    pop_nest();
    link(tail) = p;
    tail = p;
    space_factor = 1000;
  }
  type(p) = unset_node;
  glue_stretch(p) = 0;
  if (every_cr ≠ null) begin_token_list(every_cr, every_cr_text);
  align_peek();
}    /* note that glue_shrink(p) ≡ 0 since glue_shrink ≡≡ shift_amount */
```

799. 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.

```
static void do_assignments(void);
static void resume_after_display(void);
static void build_page(void);

static void fin_align(void)
{ pointer p, q, r, s, u, v;   /* registers for the list operations */
  scaled t, w;   /* width of column */
  scaled o;   /* shift offset for unset boxes */
  halfword n;   /* matching span amount */
  scaled rule_save;   /* temporary storage for overfull_rule */
  memory_word aux_save;   /* temporary storage for aux */

  if (cur_group ≠ align_group) confusion("align1");
  unsave();   /* that align_group was for individual entries */
  if (cur_group ≠ align_group) confusion("align0");
  unsave();   /* that align_group was for the whole alignment */
  if (nest[nest_ptr − 1].mode_field ≡ mmode) o = display_indent;
  else o = 0;
  ⟨Go through the preamble list, determining the column widths and changing the alignrecords to
      dummy unset boxes 800⟩;
  ⟨Package the preamble list, to determine the actual tabskip glue amounts, and let p point to this
      prototype box 803⟩;
  ⟨Set the glue in all the unset boxes of the current list 804⟩;
  flush_node_list(p);
  pop_alignment();
  ⟨Insert the current list into its environment 811⟩;
}
⟨Declare the procedure called align_peek 784⟩
```

800. 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 800 ⟩ ≡

```
q = link(preamble);
do { flush_list(u_part(q));
  flush_list(v_part(q));
  p = link(link(q));
  if (width(q) ≡ null_flag) ⟨ Nullify width(q) and the tabskip glue following this column 801 ⟩;
  if (info(q) ≠ end_span)
    ⟨ Merge the widths in the span nodes of q with those of p, destroying the span nodes of q 802 ⟩;
  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;
} while (¬(q ≡ null))
```

This code is used in section 799.

801. ⟨ Nullify *width*(q) and the tabskip glue following this column 801 ⟩ ≡

```
{ width(q) = 0;
  r = link(q);
  s = glue_ptr(r);
  if (s ≠ zero_glue) { add_glue_ref(zero_glue);
    delete_glue_ref(s);
    glue_ptr(r) = zero_glue;
  }
}
```

This code is used in section 800.

802. Merging of two span-node lists is a typical exercise in the manipulation of linearly linked data structures. The essential invariant in the following **do** { 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 $link(r) \equiv n$ 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 802⟩ ≡

```
{ t = width(q) + width(glue_ptr(link(q)));
  r = info(q);
  s = end_span;
  info(s) = p;
  n = min_quarterword + 1;
  do { width(r) = width(r) − t;
    u = info(r);
    while (link(r) > n) { s = info(s);
      n = link(info(s)) + 1;
    }
    if (link(r) < n) { info(r) = info(s);
      info(s) = r;
      decr(link(r));
      s = r;
    }
    else { if (width(r) > width(info(s))) width(info(s)) = width(r);
      free_node(r, span_node_size);
    }
    r = u;
  } while (¬(r ≡ end_span));
}
```

This code is used in section 800.

803. 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.

$\langle$ Package the preamble list, to determine the actual tabskip glue amounts, and let p point to this prototype box 803 $\rangle \equiv$

```
save_ptr = save_ptr − 2;
pack_begin_line = −mode_line;
if (mode ≡ −vmode) { rule_save = overfull_rule;
  overfull_rule = 0;   /* prevent rule from being packaged */
  p = hpack(preamble, saved(1), saved(0));
  overfull_rule = rule_save;
}
else { q = link(preamble);
  do { height(q) = width(q);
    width(q) = 0;
    q = link(link(q));
  } while (¬(q ≡ null));
  p = vpack(preamble, saved(1), saved(0));
  q = link(preamble);
  do { width(q) = height(q);
    height(q) = 0;
    q = link(link(q));
  } while (¬(q ≡ null));
}
pack_begin_line = 0
```

This code is used in section 799.

804. $\langle$ Set the glue in all the unset boxes of the current list 804 $\rangle \equiv$

```
q = link(head);
s = head;
while (q ≠ null) { if (¬is_char_node(q))
    if (type(q) ≡ unset_node) ⟨Set the unset box q and the unset boxes in it 806⟩
    else if (type(q) ≡ rule_node)
      ⟨Make the running dimensions in rule q extend to the boundaries of the alignment 805⟩;
  s = q;
  q = link(q);
}
```

This code is used in section 799.

805. $\langle$ Make the running dimensions in rule q extend to the boundaries of the alignment 805 $\rangle \equiv$

```
{ if (is_running(width(q))) width(q) = width(p);
  if (is_running(height(q))) height(q) = height(p);
  if (is_running(depth(q))) depth(q) = depth(p);
  if (o ≠ 0) { r = link(q);
    link(q) = null;
    q = hpack(q, natural);
    shift_amount(q) = o;
    link(q) = r;
    link(s) = q;
  }
}
```

This code is used in section 804.

806. 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 806 ⟩ ≡
  { **if** (*mode* ≡ −*vmode*) { *type*(*q*) = *hlist_node*;
      width(*q*) = *width*(*p*);
    }
    else { *type*(*q*) = *vlist_node*;
      height(*q*) = *height*(*p*);
    }
    glue_order(*q*) = *glue_order*(*p*);
    glue_sign(*q*) = *glue_sign*(*p*);
    glue_set(*q*) = *glue_set*(*p*);
    shift_amount(*q*) = *o*;
    r = *link*(*list_ptr*(*q*));
    s = *link*(*list_ptr*(*p*));
    do { ⟨ Set the glue in node *r* and change it from an unset node 807 ⟩;
      r = *link*(*link*(*r*));
      s = *link*(*link*(*s*));
    } **while** (¬(*r* ≡ *null*));
  }

This code is used in section 804.

807. 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 807 ⟩ ≡
  n = *span_count*(*r*);
  t = *width*(*s*);
  w = *t*;
  u = *hold_head*;
  while (*n* > *min_quarterword*) { *decr*(*n*);
    ⟨ Append tabskip glue and an empty box to list *u*, and update *s* and *t* as the prototype nodes are passed 808 ⟩;
  }
  if (*mode* ≡ −*vmode*)
    ⟨ Make the unset node *r* into an *hlist_node* of width *w*, setting the glue as if the width were *t* 809 ⟩
  else ⟨ Make the unset node *r* into a *vlist_node* of height *w*, setting the glue as if the height were *t* 810 ⟩;
  shift_amount(*r*) = 0;
  if (*u* ≠ *hold_head*)   /∗ append blank boxes to account for spanned nodes ∗/
  { *link*(*u*) = *link*(*r*);
    link(*r*) = *link*(*hold_head*);
    r = *u*;
  }

This code is used in section 806.

808. ⟨Append tabskip glue and an empty box to list u, and update s and t as the prototype nodes are passed 808⟩ ≡

$s = link(s)$;
$v = glue\_ptr(s)$;
$link(u) = new\_glue(v)$;
$u = link(u)$;
$subtype(u) = tab\_skip\_code + 1$;
$t = t + width(v)$;
if $(glue\_sign(p) \equiv stretching)$ { **if** $(stretch\_order(v) \equiv glue\_order(p))$
    $t = t + round(unfix(glue\_set(p)) * stretch(v))$;
}
else if $(glue\_sign(p) \equiv shrinking)$ { **if** $(shrink\_order(v) \equiv glue\_order(p))$
    $t = t - round(unfix(glue\_set(p)) * shrink(v))$;
}
$s = link(s)$;
$link(u) = new\_null\_box()$;
$u = link(u)$;
$t = t + width(s)$;
if $(mode \equiv -vmode)$ $width(u) = width(s)$; **else** { $type(u) = vlist\_node$;
  $height(u) = width(s)$;
}

This code is used in section 807.

809. ⟨Make the unset node r into an *hlist_node* of width w, setting the glue as if the width were t 809⟩ ≡

{ $height(r) = height(q)$;
  $depth(r) = depth(q)$;
  if $(t \equiv width(r))$ { $glue\_sign(r) = normal$;
    $glue\_order(r) = normal$;
    $set\_glue\_ratio\_zero(glue\_set(r))$;
  }
  else if $(t > width(r))$ { $glue\_sign(r) = stretching$;
    if $(glue\_stretch(r) \equiv 0)$ $set\_glue\_ratio\_zero(glue\_set(r))$;
    else $glue\_set(r) = fix((t - width(r))/(\textbf{double})\ glue\_stretch(r))$;
  }
  else { $glue\_order(r) = glue\_sign(r)$;
    $glue\_sign(r) = shrinking$;
    if $(glue\_shrink(r) \equiv 0)$ $set\_glue\_ratio\_zero(glue\_set(r))$;
    else if $((glue\_order(r) \equiv normal) \wedge (width(r) - t > glue\_shrink(r)))$
      $set\_glue\_ratio\_one(glue\_set(r))$;
    else $glue\_set(r) = fix((width(r) - t)/(\textbf{double})\ glue\_shrink(r))$;
  }
  $width(r) = w$;
  $type(r) = hlist\_node$;
}

This code is used in section 807.

810. ⟨Make the unset node r into a *vlist_node* of height w, setting the glue as if the height were t 810⟩ ≡

```
{ width(r) = width(q);
  if (t ≡ height(r)) { glue_sign(r) = normal;
    glue_order(r) = normal;
    set_glue_ratio_zero(glue_set(r));
  }
  else if (t > height(r)) { glue_sign(r) = stretching;
    if (glue_stretch(r) ≡ 0) set_glue_ratio_zero(glue_set(r));
    else glue_set(r) = fix((t − height(r))/(double) glue_stretch(r));
  }
  else { glue_order(r) = glue_sign(r);
    glue_sign(r) = shrinking;
    if (glue_shrink(r) ≡ 0) set_glue_ratio_zero(glue_set(r));
    else if ((glue_order(r) ≡ normal) ∧ (height(r) − t > glue_shrink(r)))
      set_glue_ratio_one(glue_set(r));
    else glue_set(r) = fix((height(r) − t)/(double) glue_shrink(r));
  }
  height(r) = w;
  type(r) = vlist_node;
}
```

This code is used in section 807.

811. 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 811⟩ ≡

```
aux_save = aux;
p = link(head);
q = tail;
pop_nest();
if (mode ≡ mmode) ⟨Finish an alignment in a display 1205⟩
else { aux = aux_save;
  link(tail) = p;
  if (p ≠ null) tail = q;
  if (mode ≡ vmode) build_page();
}
```

This code is used in section 799.

812. 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.

813. 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⟩ +≡
  static pointer *just_box*; /∗ the *hlist_node* for the last line of the new paragraph ∗/

814. 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* 825⟩
  static void *line_break*(**int** *final_widow_penalty*)
  { ⟨Local variables for line breaking 861⟩
    ⟨Local variables to save the profiling context 1761⟩
    ⟨Charge the time used here on *line_break* 1762⟩
    pack_begin_line = *mode_line*; /∗ this is for over/underfull box messages ∗/
    ⟨Get ready to start line breaking 815⟩;
    ⟨Find optimal breakpoints 862⟩;
    ⟨Break the paragraph at the chosen breakpoints, justify the resulting lines to the correct widths, and append them to the current vertical list 875⟩;
    ⟨Clean up the memory by removing the break nodes 864⟩;
    pack_begin_line = 0;
    ⟨restore the previous current file, line, and command 1763⟩
  }
⟨Declare ε-TEX procedures for use by *main_control* 1386⟩

815. 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 815 ⟩ ≡
link(*temp_head*) = *link*(*head*);
if (*is_char_node*(*tail*)) *tail_append*(*new_penalty*(*inf_penalty*))
else if (*type*(*tail*) ≠ *glue_node*) *tail_append*(*new_penalty*(*inf_penalty*))
else { *type*(*tail*) = *penalty_node*;
delete_glue_ref(*glue_ptr*(*tail*));
flush_node_list(*leader_ptr*(*tail*));
penalty(*tail*) = *inf_penalty*;
}
link(*tail*) = *new_param_glue*(*par_fill_skip_code*);
init_cur_lang = *prev_graf* % °*200000*;
init_l_hyf = *prev_graf* /°*20000000*;
init_r_hyf = (*prev_graf* /°*200000*) % °*100*;
pop_nest( );

See also sections 826, 833, and 847.

This code is used in section 814.

816. 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 ∗/

817. 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.

818. 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*(*A*) *subtype*(*A*) /∗ *very_loose_fit* .. *tight_fit* on final line for this break ∗/
#define *break_node*(*A*) *rlink*(*A*) /∗ pointer to the corresponding passive node ∗/
#define *line_number*(*A*) *llink*(*A*) /∗ line that begins at this breakpoint ∗/
#define *total_demerits*(*A*) *mem*[*A* + 2].*i* /∗ 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 ∗/

819. ⟨Initialize the special list heads and constant nodes 789⟩ +≡
type(*last_active*) = *hyphenated*;
line_number(*last_active*) = *max_halfword*;
subtype(*last_active*) = 0; /∗ the *subtype* is never examined by the algorithm ∗/

820. 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*(*A*) *rlink*(*A*) /∗ in passive node, points to position of this breakpoint ∗/
#define *prev_break*(*A*) *llink*(*A*) /∗ points to passive node that should precede this one ∗/
#define *serial*(*A*) *info*(*A*) /∗ serial number for symbolic identification ∗/

⟨Global variables 13⟩ +≡
static pointer *passive*; /∗ most recent node on passive list ∗/
static pointer *printed_node*; /∗ most recent node that has been printed ∗/
static halfword *pass_number*; /∗ the number of passive nodes allocated on this pass ∗/

821. 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* $\equiv$ *delta_node*. If p and r are active nodes and if q is a delta node between them, so that $link(p) \equiv q$ and $link(q) \equiv r$, 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 $mem[q+2 \mathrel{..} q+5].sc$; 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 ∗/

822. 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

$$\textbf{for } k = 1 \textbf{ to } 6 \textbf{ do } cur\_active\_width[k] = cur\_active\_width[k] + mem[q+k].sc;$$

and we want to do this without the overhead of **for** loops. The *do_all_six* macro makes such six-tuples convenient.

#**define** *do_all_six*(A) $A(1)$;
$A(2)$;
$A(3)$;
$A(4)$;
$A(5)$; $A(6)$

⟨ Global variables 13 ⟩ +≡
static scaled *active_width0*[6], ∗**const** *active_width* = *active_width0* − 1;
/∗ distance from first active node to *cur_p* ∗/
static scaled *cur_active_width0*[6], ∗**const** *cur_active_width* = *cur_active_width0* − 1;
/∗ distance from current active node ∗/
static scaled *background0*[6], ∗**const** *background* = *background0* − 1; /∗ length of an "empty" line ∗/
static scaled *break_width0*[6], ∗**const** *break_width* = *break_width0* − 1;
/∗ length being computed after current break ∗/

823. Let's state the principles of the delta nodes more precisely and concisely, so that the following programs will be less obscure. For each legal breakpoint p in the paragraph, we define two quantities $\alpha(p)$ and $\beta(p)$ such that the length of material in a line from breakpoint p to breakpoint q is $\gamma+\beta(q)-\alpha(p)$, for some fixed γ. Intuitively, $\alpha(p)$ and $\beta(q)$ are the total length of material from the beginning of the paragraph to a point "after" a break at p and to a point "before" a break at q; and γ is the width of an empty line, namely the length contributed by `\leftskip` and `\rightskip`.

Suppose, for example, that the paragraph consists entirely of alternating boxes and glue skips; let the boxes have widths $x_1 \ldots x_n$ and let the skips have widths $y_1 \ldots y_n$, so that the paragraph can be represented by $x_1 y_1 \ldots x_n y_n$. Let p_i be the legal breakpoint at y_i; then $\alpha(p_i)=x_1+y_1+\cdots+x_i+y_i$, and $\beta(p_i)=x_1+y_1+\cdots+x_i$. To check this, note that the length of material from p_2 to p_5, say, is $\gamma+x_3+y_3+x_4+y_4+x_5=\gamma+\beta(p_5)-\alpha(p_2)$.

The quantities α, β, γ involve glue stretchability and shrinkability as well as a natural width. If we were to compute $\alpha(p)$ and $\beta(p)$ for each p, we would need multiple precision arithmetic, and the multiprecise numbers would have to be kept in the active nodes. TEX avoids this problem by working entirely with relative differences or "deltas." Suppose, for example, that the active list contains $a_1\,\delta_1\,a_2\,\delta_2\,a_3$, where the a's are active breakpoints and the δ's are delta nodes. Then $\delta_1=\alpha(a_1)-\alpha(a_2)$ and $\delta_2=\alpha(a_2)-\alpha(a_3)$. If the line breaking algorithm is currently positioned at some other breakpoint p, the *active_width* array contains the value $\gamma+\beta(p)-\alpha(a_1)$. If we are scanning through the list of active nodes and considering a tentative line that runs from a_2 to p, say, the *cur_active_width* array will contain the value $\gamma+\beta(p)-\alpha(a_2)$. Thus, when we move from a_2 to a_3, we want to add $\alpha(a_2)-\alpha(a_3)$ to *cur_active_width*; and this is just δ_2, which appears in the active list between a_2 and a_3. The *background* array contains γ. The *break_width* array will be used to calculate values of new delta nodes when the active list is being updated.

824. 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*(*A*)
  if ((*shrink_order*(*A*) ≠ *normal*) ∧ (*shrink*(*A*) ≠ 0)) { *A* = *finite_shrink*(*A*);
  }

⟨ Global variables 13 ⟩ +≡
  static bool *no_shrink_error_yet*; /* have we complained about infinite shrinkage? */

825. ⟨ Declare subprocedures for *line_break* 825 ⟩ ≡

```
static pointer finite_shrink(pointer p)    /* recovers from infinite shrinkage */
{ pointer q;    /* new glue specification */
  if (no_shrink_error_yet) { no_shrink_error_yet = false;
#ifdef STAT
    if (tracing_paragraphs > 0) end_diagnostic(true);
#endif
    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( );
#ifdef STAT
    if (tracing_paragraphs > 0) begin_diagnostic( );
#endif
  }
  q = new_spec(p);
  shrink_order(q) = normal;
  delete_glue_ref(p);
  return q;
}
```

See also sections 828, 876, 894, and 941.
This code is used in section 814.

826. ⟨ Get ready to start line breaking 815 ⟩ +≡

```
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);
```

827. 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 ⟩ +≡
  static pointer *cur_p*; /* the current breakpoint under consideration */
  static bool *second_pass*; /* is this our second attempt to break this paragraph? */
  static bool *final_pass*; /* is this our final attempt to break this paragraph? */
  static int *threshold*; /* maximum badness on feasible lines */

828. The heart of the line-breaking procedure is '*try_break*', a subroutine that tests if the current break-point *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 $pi \equiv eject\_penalty$ if the break is forced, and $pi \equiv inf\_penalty$ 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 $cur\_p \equiv null$.

#define *copy_to_cur_active*(*A*) *cur_active_width*[*A*] = *active_width*[*A*]

⟨ Declare subprocedures for *line_break* 825 ⟩ +≡
  static void *try_break*(**int** *pi*, **small_number** *break_type*)
  { **pointer** *r*; /∗ runs through the active list ∗/
    pointer *prev_r*; /∗ stays a step behind *r* ∗/
    halfword *old_l*; /∗ maximum line number in current equivalence class of lines ∗/
    bool *no_break_yet*; /∗ have we found a feasible break at *cur_p*? ∗/
    ⟨ Other local variables for *try_break* 829 ⟩
    ⟨ Make sure that *pi* is in the proper range 830 ⟩;
    no_break_yet = *true*;
    prev_r = *active*;
    old_l = 0;
    do_all_six(*copy_to_cur_active*);
    loop { *resume*: *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** *resume* 831 ⟩;
      ⟨ 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* 834 ⟩;
      ⟨ Consider the demerits for a line from *r* to *cur_p*; deactivate node *r* if it should no longer be active; then **goto** *resume* if a line from *r* to *cur_p* is infeasible, otherwise record a new feasible break 850 ⟩;
    }
  end: ;
#ifdef STAT
    ⟨ Update the value of *printed_node* for symbolic displays 857 ⟩;
#endif
  }

829. ⟨ Other local variables for *try_break* 829 ⟩ ≡
  pointer *prev_prev_r*; /* a step behind *prev_r*, if $type(prev\_r) \equiv delta\_node$ */
  pointer *s*; /* runs through nodes ahead of *cur_p* */
  pointer *q*; /* points to a new node being created */
  pointer *v*; /* points to a glue specification or a node ahead of *cur_p* */
  int *t*; /* node count, if *cur_p* is a discretionary node */
  internal_font_number *f*; /* used in character width calculation */
  halfword *l*; /* line number of current active node */
  bool *node_r_stays_active*; /* should node *r* remain in the active list? */
  scaled *line_width*; /* the current line will be justified to this width */
  int *fit_class*; /* possible fitness class of test line */
  halfword *b*; /* badness of test line */
  int *d*; /* demerits of test line */
  bool *artificial_demerits*; /* has *d* been forced to zero? */
#**ifdef** STAT
  pointer *save_link*; /* temporarily holds value of *link*(*cur_p*) */
#**endif**
  scaled *shortfall*; /* used in badness calculations */

This code is used in section 828.

830. ⟨ Make sure that *pi* is in the proper range 830 ⟩ ≡
  if $(abs(pi) \ge inf\_penalty)$
  if $(pi > 0)$ **goto** *end*; /* this breakpoint is inhibited by infinite penalty */
  else $pi = eject\_penalty$ /* this breakpoint will be forced */

This code is used in section 828.

831. The following code uses the fact that $type(last\_active) \ne delta\_node$.

#**define** *update_width*(*A*) $cur\_active\_width[A] = cur\_active\_width[A] + mem[r + A].sc$

⟨ If node *r* is of type *delta_node*, update *cur_active_width*, set *prev_r* and *prev_prev_r*, then **goto** *resume* 831 ⟩ ≡
  if $(type(r) \equiv delta\_node)$ { *do_all_six*(*update_width*);
    $prev\_prev\_r = prev\_r$;
    $prev\_r = r$;
    goto *resume*;
  }

This code is used in section 828.

832. 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 ⟩ +≡
static int *minimal_demerits0*[*tight_fit* − *very_loose_fit* + 1], ∗**const** *minimal_demerits* = *minimal_demerits0* − *very_loose_fit*;
/∗ best total demerits known for current line class and position, given the fitness ∗/
static int *minimum_demerits*; /∗ best total demerits known for current line class and position ∗/
static pointer *best_place0*[*tight_fit* − *very_loose_fit* + 1], ∗**const** *best_place* = *best_place0* − *very_loose_fit*;
/∗ how to achieve *minimal_demerits* ∗/
static halfword *best_pl_line0*[*tight_fit* − *very_loose_fit* + 1], ∗**const** *best_pl_line* = *best_pl_line0* − *very_loose_fit*; /∗ corresponding line number ∗/

833. ⟨ Get ready to start line breaking 815 ⟩ +≡
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*;

834. 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 $r \equiv$ *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* 834 ⟩ ≡
{ l = *line_number*(r);
  if (l > *old_l*) { /∗ now we are no longer in the inner loop ∗/
    if ((*minimum_demerits* < *awful_bad*) ∧ ((*old_l* ≠ *easy_line*) ∨ (r ≡ *last_active*)))
      ⟨ Create new active nodes for the best feasible breaks just found 835 ⟩;
    if (r ≡ *last_active*) **goto** *end*;
    ⟨ Compute the new line width 849 ⟩;
  }
}

This code is used in section 828.

835. 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 835⟩ ≡
  { **if** (*no_break_yet*) ⟨Compute the values of *break_width* 836⟩;
   ⟨Insert a delta node to prepare for breaks at *cur_p* 842⟩;
   if (*abs*(*adj_demerits*) ≥ *awful_bad* − *minimum_demerits*) *minimum_demerits* = *awful_bad* − 1;
   else *minimum_demerits* = *minimum_demerits* + *abs*(*adj_demerits*);
   for (*fit_class* = *very_loose_fit*; *fit_class* ≤ *tight_fit*; *fit_class*++) {
     if (*minimal_demerits*[*fit_class*] ≤ *minimum_demerits*)
     ⟨Insert a new active node from *best_place*[*fit_class*] to *cur_p* 844⟩;
    minimal_demerits[*fit_class*] = *awful_bad*;
   }
   minimum_demerits = *awful_bad*;
   ⟨Insert a delta node to prepare for the next active node 843⟩;
  }

This code is used in section 834.

836. 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*(*A*) *break_width*[*A*] = *background*[*A*]

⟨Compute the values of *break_width* 836⟩ ≡
  { *no_break_yet* = *false*;
  do_all_six(*set_break_width_to_background*);
  s = *cur_p*;
  if (*break_type* > *unhyphenated*)
   if (*cur_p* ≠ *null*) ⟨Compute the discretionary *break_width* values 839⟩;
  while (*s* ≠ *null*) { **if** (*is_char_node*(*s*)) **goto** *done*;
  switch (*type*(*s*)) {
case *glue_node*: ⟨Subtract glue from *break_width* 837⟩ **break**;
case *penalty_node*: *do_nothing*; **break**;
case *math_node*: *break_width*[1] = *break_width*[1] − *width*(*s*);
  break; **case** *kern_node*: **if** (*subtype*(*s*) ≠ **explicit**) **goto** *done*;
  else *break_width*[1] = *break_width*[1] − *width*(*s*); **break**;
default: **goto** *done*;
  }
  s = *link*(*s*);
  }
done: ;
  }

This code is used in section 835.

837. ⟨ Subtract glue from *break_width* 837 ⟩ ≡
{ $v = glue\_ptr(s)$;
  $break\_width[1] = break\_width[1] - width(v)$;
  $break\_width[2 + stretch\_order(v)] = break\_width[2 + stretch\_order(v)] - stretch(v)$;
  $break\_width[6] = break\_width[6] - shrink(v)$;
}
This code is used in section 836.

838. 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 ⟩ +≡
  static scaled *disc_width*; /* the length of discretionary material preceding a break */

839. ⟨ Compute the discretionary *break_width* values 839 ⟩ ≡
{ $t = replace\_count(cur\_p)$;
  $v = cur\_p$;
  $s = post\_break(cur\_p)$;
  while $(t > 0)$ { $decr(t)$;
    $v = link(v)$;
    ⟨ Subtract the width of node v from *break_width* 840 ⟩;
  }
  while $(s \neq null)$ { ⟨ Add the width of node s to *break_width* 841 ⟩;
    $s = link(s)$;
  }
  $break\_width[1] = break\_width[1] + disc\_width$;
  if $(post\_break(cur\_p) \equiv null)$ $s = link(v)$; /* nodes may be discardable after the break */
}
This code is used in section 836.

840. 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* 840⟩ ≡
if (*is_char_node*(v)) { f = *font*(v);
  break_width[1] = *break_width*[1] − *char_width*(f, *char_info*(f, *character*(v)));
}
else
  switch (*type*(v)) {
  case *ligature_node*:
    { f = *font*(*lig_char*(v));
      break_width[1] =
          break_width[1] − *char_width*(f, *char_info*(f, *character*(*lig_char*(v))));
    } **break**;
  case *hlist_node*: **case** *vlist_node*: **case** *rule_node*: **case** *kern_node*:
    break_width[1] = *break_width*[1] − *width*(v); **break**;
  default: *confusion*("disc1");
  }

This code is used in section 839.

841. ⟨Add the width of node s to *break_width* 841⟩ ≡
if (*is_char_node*(s)) { f = *font*(s);
  break_width[1] =
      break_width[1] + *char_width*(f, *char_info*(f, *character*(s)));
}
else
  switch (*type*(s)) {
  case *ligature_node*:
    { f = *font*(*lig_char*(s));
      break_width[1] = *break_width*[1] + *char_width*(f, *char_info*(f, *character*(*lig_char*(s))));
    } **break**;
  case *hlist_node*: **case** *vlist_node*: **case** *rule_node*: **case** *kern_node*:
    break_width[1] = *break_width*[1] + *width*(s); **break**;
  default: *confusion*("disc2");
  }

This code is used in section 839.

842. We use the fact that $type(active) \neq delta\_node$.

#define $convert\_to\_break\_width(A)$ $mem[prev\_r + A].sc = mem[prev\_r + A].sc - cur\_active\_width[A] + break\_width[A]$
#define $store\_break\_width(A)$ $active\_width[A] = break\_width[A]$
#define $new\_delta\_to\_break\_width(A)$ $mem[q + A].sc = break\_width[A] - cur\_active\_width[A]$

⟨Insert a delta node to prepare for breaks at cur_p 842⟩ ≡

```
if (type(prev_r) ≡ delta_node)   /* modify an existing delta node */
{ do_all_six(convert_to_break_width);
}
else if (prev_r ≡ active)   /* no delta node needed at the beginning */
{ do_all_six(store_break_width);
}
else { q = get_node(delta_node_size);
  link(q) = r;
  type(q) = delta_node;
  subtype(q) = 0;   /* the subtype is not used */
  do_all_six(new_delta_to_break_width);
  link(prev_r) = q;
  prev_prev_r = prev_r;
  prev_r = q;
}
```

This code is used in section 835.

843. 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 $new\_delta\_from\_break\_width(A)$ $mem[q + A].sc = cur\_active\_width[A] - break\_width[A]$

⟨Insert a delta node to prepare for the next active node 843⟩ ≡

```
if (r ≠ last_active) { q = get_node(delta_node_size);
  link(q) = r;
  type(q) = delta_node;
  subtype(q) = 0;   /* the subtype is not used */
  do_all_six(new_delta_from_break_width);
  link(prev_r) = q;
  prev_prev_r = prev_r;
  prev_r = q;
}
```

This code is used in section 835.

844. 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* 844⟩ ≡

```
{ q = get_node(passive_node_size);
   link(q) = passive;
   passive = q;
   cur_break(q) = cur_p;
#ifdef STAT
   incr(pass_number);
   serial(q) = pass_number;
#endif
   prev_break(q) = best_place[fit_class];
   q = get_node(active_node_size);
   break_node(q) = passive;
   line_number(q) = best_pl_line[fit_class] + 1;
   fitness(q) = fit_class;
   type(q) = break_type;
   total_demerits(q) = minimal_demerits[fit_class];
   link(q) = r;
   link(prev_r) = q;
   prev_r = q;
#ifdef STAT
   if (tracing_paragraphs > 0) ⟨Print a symbolic description of the new break node 845⟩;
#endif
}
```

This code is used in section 835.

845. ⟨Print a symbolic description of the new break node 845⟩ ≡

```
{ print_nl("@@");
   print_int(serial(passive));
   print(":␣line␣");
   print_int(line_number(q) − 1);
   print_char('.');
   print_int(fit_class);
   if (break_type ≡ hyphenated) print_char('-');
   print("␣t=");
   print_int(total_demerits(q));
   print("␣->␣@@");
   if (prev_break(passive) ≡ null) print_char('0');
   else print_int(serial(prev_break(passive)));
}
```

This code is used in section 844.

846. 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 $par\_shape\_ptr \equiv null$, the shape of the paragraph depends on the value of $n \equiv hang\_after$; if $n \ge 0$, hanging indentation takes place on lines $n+1$, $n+2$, ..., otherwise it takes place on lines $1, \ldots, |n|$. When hanging indentation is active, the left margin is *hang_indent*, if $hang\_indent \ge 0$, else it is 0; the line length is $hsize - |hang\_indent|$. The normal setting is $par\_shape\_ptr \equiv null$, $hang\_after \equiv 1$, and $hang\_indent \equiv 0$. Note that if $hang\_indent \equiv 0$, the value of *hang_after* is irrelevant.

⟨Global variables 13⟩ +≡
static halfword *easy_line*; /* line numbers > *easy_line* are equivalent in break nodes */
static halfword *last_special_line*; /* line numbers > *last_special_line* all have the same width */
static scaled *first_width*;
/* the width of all lines ≤ *last_special_line*, if no `\parshape` has been specified */
static scaled *second_width*; /* the width of all lines > *last_special_line* */
static scaled *first_indent*; /* left margin to go with *first_width* */
static scaled *second_indent*; /* left margin to go with *second_width* */

847. 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 815⟩ +≡
```
if (par_shape_ptr ≡ null)
  if (hang_indent ≡ 0) { last_special_line = 0;
    second_width = hsize;
    second_indent = 0;
  }
  else ⟨Set line length parameters in preparation for hanging indentation 848⟩
else { 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;
}
if (looseness ≡ 0) easy_line = last_special_line;
else easy_line = max_halfword
```

848. ⟨Set line length parameters in preparation for hanging indentation 848⟩ ≡
```
{ last_special_line = abs(hang_after);
  if (hang_after < 0) { first_width = hsize − abs(hang_indent);
    if (hang_indent ≥ 0) first_indent = hang_indent;
    else first_indent = 0;
    second_width = hsize;
    second_indent = 0;
  }
  else { first_width = hsize;
    first_indent = 0;
    second_width = hsize − abs(hang_indent);
    if (hang_indent ≥ 0) second_indent = hang_indent;
    else second_indent = 0;
  }
}
```
This code is used in section 847.

849. When we come to the following code, we have just encountered the first active node r whose *line_number* field contains l. Thus we want to compute the length of the lth line of the current paragraph. Furthermore, we want to set *old_l* to the last number in the class of line numbers equivalent to l.

⟨ Compute the new line width 849 ⟩ ≡
  if (l > *easy_line*) { *line_width* = *second_width*;
    old_l = *max_halfword* − 1;
  }
  else { *old_l* = l;
    if (l > *last_special_line*) *line_width* = *second_width*;
    else if (*par_shape_ptr* ≡ *null*) *line_width* = *first_width*;
    else *line_width* = *mem*[*par_shape_ptr* + 2 ∗ l].*sc*;
  }

This code is used in section 834.

850. 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** *resume* if a line from r to *cur_p* is infeasible, otherwise record a new feasible break 850 ⟩ ≡
  { *artificial_demerits* = *false*;
    shortfall = *line_width* − *cur_active_width*[1];   /∗ we're this much too short ∗/
    if (*shortfall* > 0) ⟨ Set the value of b to the badness for stretching the line, and compute the corresponding *fit_class* 851 ⟩
    else ⟨ Set the value of b to the badness for shrinking the line, and compute the corresponding *fit_class* 852 ⟩;
    if ((b > *inf_bad*) ∨ (*pi* ≡ *eject_penalty*)) ⟨ Prepare to deactivate node r, and **goto** *deactivate* unless there is a reason to consider lines of text from r to *cur_p* 853 ⟩
    else { *prev_r* = r;
      if (b > *threshold*) **goto** *resume*;
      node_r_stays_active = *true*;
    }
    ⟨ Record a new feasible break 854 ⟩;
    if (*node_r_stays_active*) **goto** *resume*;   /∗ *prev_r* has been set to r ∗/
  deactivate: ⟨ Deactivate node r 859 ⟩;
  }

This code is used in section 828.

851. 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* 851 ⟩ ≡

```
if ((cur_active_width[3] ≠ 0) ∨ (cur_active_width[4] ≠ 0) ∨
        (cur_active_width[5] ≠ 0)) { b = 0;
    fit_class = decent_fit;     /* infinite stretch */
}
else { if (shortfall > 7230584)
      if (cur_active_width[2] < 1663497) { b = inf_bad;
        fit_class = very_loose_fit;
        goto done1;
      }
    b = badness(shortfall, cur_active_width[2]);
    if (b > 12)
      if (b > 99) fit_class = very_loose_fit;
      else fit_class = loose_fit;
    else fit_class = decent_fit;
  done1: ;
}
```

This code is used in section 850.

852. 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* 852 ⟩ ≡

```
{ if (−shortfall > cur_active_width[6]) b = inf_bad + 1;
    else b = badness(−shortfall, cur_active_width[6]);
    if (b > 12) fit_class = tight_fit; else fit_class = decent_fit;
}
```

This code is used in section 850.

853. 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* 853 ⟩ ≡

```
{ if (final_pass ∧ (minimum_demerits ≡ awful_bad) ∧
          (link(r) ≡ last_active) ∧ (prev_r ≡ active)) artificial_demerits = true;
        /* set demerits zero, this break is forced */
    else if (b > threshold) goto deactivate;
    node_r_stays_active = false;
}
```

This code is used in section 850.

854. 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 854⟩ ≡
  if (artificial_demerits) d = 0;
  else ⟨Compute the demerits, d, from r to cur_p 858⟩;
#ifdef STAT
  if (tracing_paragraphs > 0) ⟨Print a symbolic description of this feasible break 855⟩;
#endif
  d = d + total_demerits(r);   /* this is the minimum total demerits from the beginning to cur_p via r */
  if (d ≤ minimal_demerits[fit_class]) { minimal_demerits[fit_class] = d;
    best_place[fit_class] = break_node(r);
    best_pl_line[fit_class] = l;
    if (d < minimum_demerits) minimum_demerits = d;
  }
```

This code is used in section 850.

855. ⟨Print a symbolic description of this feasible break 855⟩ ≡

```
  { if (printed_node ≠ cur_p)
        ⟨Print the list between printed_node and cur_p, then set printed_node: = cur_p 856⟩;
    print_nl("@");
    if (cur_p ≡ null) print_esc("par");
    else if (type(cur_p) ≠ glue_node) { if (type(cur_p) ≡ penalty_node) print_esc("penalty");
      else if (type(cur_p) ≡ disc_node) print_esc("discretionary");
      else if (type(cur_p) ≡ kern_node) print_esc("kern");
      else print_esc("math");
    }
    print("␣via␣@@");
    if (break_node(r) ≡ null) print_char('0');
    else print_int(serial(break_node(r)));
    print("␣b=");
    if (b > inf_bad) print_char('*'); else print_int(b);
    print("␣p=");
    print_int(pi);
    print("␣d=");
    if (artificial_demerits) print_char('*'); else print_int(d);
  }
```

This code is used in section 854.

856. ⟨Print the list between printed_node and cur_p, then set printed_node: = cur_p 856⟩ ≡

```
  { print_nl("");
    if (cur_p ≡ null) short_display(link(printed_node));
    else { save_link = link(cur_p);
      link(cur_p) = null;
      print_nl("");
      short_display(link(printed_node));
      link(cur_p) = save_link;
    }
    printed_node = cur_p;
  }
```

This code is used in section 855.

857. 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 857⟩ ≡
```
if (cur_p ≡ printed_node)
  if (cur_p ≠ null)
    if (type(cur_p) ≡ disc_node) { t = replace_count(cur_p);
      while (t > 0) { decr(t);
        printed_node = link(printed_node);
      }
    }
```
This code is used in section 828.

858. ⟨Compute the demerits, d, from r to *cur_p* 858⟩ ≡
```
{ d = line_penalty + b;
  if (abs(d) ≥ 10000) d = 100000000; else d = d * d;
  if (pi ≠ 0)
    if (pi > 0) d = d + pi * pi;
    else if (pi > eject_penalty) d = d − pi * pi;
  if ((break_type ≡ hyphenated) ∧ (type(r) ≡ hyphenated))
    if (cur_p ≠ null) d = d + double_hyphen_demerits;
    else d = d + final_hyphen_demerits;
  if (abs(fit_class − fitness(r)) > 1) d = d + adj_demerits;
}
```
This code is used in section 854.

859. 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(A) mem[prev_r + A].sc = mem[prev_r + A].sc + mem[r + A].sc
#define downdate_width(A) cur_active_width[A] = cur_active_width[A] − mem[prev_r + A].sc
```

⟨Deactivate node r 859⟩ ≡
```
link(prev_r) = link(r);
free_node(r, active_node_size);
if (prev_r ≡ active) ⟨Update the active widths, since the first active node has been deleted 860⟩
else if (type(prev_r) ≡ delta_node) { r = link(prev_r);
  if (r ≡ last_active) { do_all_six(downdate_width);
    link(prev_prev_r) = last_active;
    free_node(prev_r, delta_node_size);
    prev_r = prev_prev_r;
  }
  else if (type(r) ≡ delta_node) { do_all_six(update_width);
    do_all_six(combine_two_deltas);
    link(prev_r) = link(r);
    free_node(r, delta_node_size);
  }
}
```
This code is used in section 850.

860. The following code uses the fact that $type(last\_active) \neq delta\_node$. 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 $update\_active(A)$ $active\_width[A] = active\_width[A] + mem[r + A].sc$

⟨ Update the active widths, since the first active node has been deleted 860 ⟩ ≡

$\{$ $r = link(active)$;
  if $(type(r) \equiv delta\_node)$ $\{$ $do\_all\_six(update\_active)$;
    $do\_all\_six(copy\_to\_cur\_active)$;
    $link(active) = link(r)$;
    $free\_node(r, delta\_node\_size)$;
  $\}$
$\}$

This code is used in section 859.

861. 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*) $\equiv$ *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 861 ⟩ $\equiv$

bool *auto_breaking*; /* is node *cur_p* outside a formula? */
pointer *prev_p*; /* helps to determine when glue nodes are breakpoints */
pointer *q*, *r*, *s*, *prev_s*; /* miscellaneous nodes of temporary interest */
internal_font_number *f*; /* used when calculating character widths */

See also section 892.

This code is used in section 814.

862. 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 862⟩ ≡

```
  threshold = pretolerance;
  if (threshold ≥ 0) {
#ifdef STAT
    if (tracing_paragraphs > 0) { begin_diagnostic();
      print_nl("@firstpass"); }
#endif
    second_pass = false;
    final_pass = false;
  }
  else { threshold = tolerance;
    second_pass = true;
    final_pass = (emergency_stretch ≤ 0);
#ifdef STAT
    if (tracing_paragraphs > 0) begin_diagnostic();
#endif
  }
  loop { if (threshold > inf_bad) threshold = inf_bad;
    if (second_pass) ⟨Initialize for hyphenating a paragraph 890⟩;
    ⟨Create an active breakpoint representing the beginning of the paragraph 863⟩;
    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)) ⟨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 865⟩;
    if (cur_p ≡ null) ⟨Try the final line break at the end of the paragraph, and goto done if the desired
          breakpoints have been found 872⟩;
    ⟨Clean up the memory by removing the break nodes 864⟩;
    if (¬second_pass) {
#ifdef STAT
      if (tracing_paragraphs > 0) print_nl("@secondpass");
#endif
      threshold = tolerance;
      second_pass = true;
      final_pass = (emergency_stretch ≤ 0);
    }    /* if at first you don't succeed, ... */
    else {
#ifdef STAT
      if (tracing_paragraphs > 0) print_nl("@emergencypass");
#endif
      background[2] = background[2] + emergency_stretch;
      final_pass = true;
    }
  }
done:
#ifdef STAT
  if (tracing_paragraphs > 0) { end_diagnostic(true);
    normalize_selector();
  }
```

#endif

This code is used in section 814.

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

#define *store_background*(*A*) *active_width*[*A*] = *background*[*A*]

⟨ Create an active breakpoint representing the beginning of the paragraph 863 ⟩ ≡

$q = get\_node(active\_node\_size)$;
$type(q) = unhyphenated$;
$fitness(q) = decent\_fit$;
$link(q) = last\_active$;
$break\_node(q) = null$;
$line\_number(q) = prev\_graf + 1$;
$total\_demerits(q) = 0$;
$link(active) = q$;
$do\_all\_six(store\_background)$;
$passive = null$;
$printed\_node = temp\_head$;
$pass\_number = 0$; $font\_in\_short\_display = null\_font$

This code is used in section 862.

864. ⟨ Clean up the memory by removing the break nodes 864 ⟩ ≡

$q = link(active)$;
while $(q \neq last\_active)$ { $cur\_p = link(q)$;
  if $(type(q) \equiv delta\_node)$ $free\_node(q, delta\_node\_size)$;
  else $free\_node(q, active\_node\_size)$;
  $q = cur\_p$;
}
$q = passive$;
while $(q \neq null)$ { $cur\_p = link(q)$;
  $free\_node(q, passive\_node\_size)$;
  $q = cur\_p$;
}

This code is used in sections 814 and 862.

865. 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*
  { **if** (¬*is_char_node*(*link*(*cur_p*)) ∧ *auto_breaking*)
    if (*type*(*link*(*cur_p*)) ≡ *glue_node*) *try_break*(0, *unhyphenated*);
   act_width = *act_width* + *width*(*cur_p*);
  }

⟨ 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 865 ⟩ ≡
 { **if** (*is_char_node*(*cur_p*)) ⟨ Advance *cur_p* to the node following the present string of characters 866 ⟩;
 switch (*type*(*cur_p*)) {
case *hlist_node*: **case** *vlist_node*: **case** *rule_node*: *act_width* = *act_width* + *width*(*cur_p*); **break**;
case *whatsit_node*: ⟨ Advance past a whatsit node in the *line_break* loop 1361 ⟩ **break**;
case *glue_node*:
 { ⟨ 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*) 867 ⟩;
  if (*second_pass* ∧ *auto_breaking*) ⟨ Try to hyphenate the following word 893 ⟩;
 } **break**; **case** *kern_node*: **if** (*subtype*(*cur_p*) ≡ **explicit**) *kern_break*
 else *act_width* = *act_width* + *width*(*cur_p*); **break**;
case *ligature_node*:
 { *f* = *font*(*lig_char*(*cur_p*));
  act_width = *act_width* + *char_width*(*f*, *char_info*(*f*, *character*(*lig_char*(*cur_p*))));
 } **break**;
case *disc_node*: ⟨ Try to break after a discretionary fragment, then **goto** *done5* 868 ⟩
case *math_node*:
 { *auto_breaking* = (*subtype*(*cur_p*) ≡ *after*);
  kern_break;
 } **break**;
case *penalty_node*: *try_break*(*penalty*(*cur_p*), *unhyphenated*); **break**;
case *mark_node*: **case** *ins_node*: **case** *adjust_node*: *do_nothing*; **break**;
default: *confusion*("paragraph");
 }
 prev_p = *cur_p*;
 cur_p = *link*(*cur_p*);
done5: ;
 }

This code is used in section 862.

866. 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 $link(cur\_p) \equiv null$ when cur_p is a character node.

$\langle$ Advance cur_p to the node following the present string of characters 866 $\rangle \equiv$
{ $prev\_p = cur\_p$;
  do { $f = font(cur\_p)$;
    $act\_width = act\_width + char\_width(f, char\_info(f, character(cur\_p)))$;
    $cur\_p = link(cur\_p)$;
  } **while** $(\neg(\neg is\_char\_node(cur\_p)))$;
}

This code is used in section 865.

867. 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.

$\langle$ If node cur_p is a legal breakpoint, call try_break; then update the active widths by including the glue in $glue\_ptr(cur\_p)$ 867 $\rangle \equiv$
if $(auto\_breaking)$ { **if** $(is\_char\_node(prev\_p))$ $try\_break(0, unhyphenated)$;
else if $(precedes\_break(prev\_p))$ $try\_break(0, unhyphenated)$;
else if $((type(prev\_p) \equiv kern\_node) \wedge (subtype(prev\_p) \neq$ **explicit** $))$ $try\_break(0, unhyphenated)$;
}
$check\_shrinkage(glue\_ptr(cur\_p))$;
$q = glue\_ptr(cur\_p)$;
$act\_width = act\_width + width(q)$;

$active\_width[2 + stretch\_order(q)] =$
  $active\_width[2 + stretch\_order(q)] + stretch(q)$;
$active\_width[6] = active\_width[6] + shrink(q)$

This code is used in section 865.

868. 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* 868 ⟩ ≡

```
{ s = pre_break(cur_p);
  disc_width = 0;
  if (s ≡ null) try_break(ex_hyphen_penalty, hyphenated);
  else { do { ⟨ Add the width of node s to disc_width 869 ⟩;
        s = link(s);
      } while (¬(s ≡ null));
      act_width = act_width + disc_width;
      try_break(hyphen_penalty, hyphenated);
      act_width = act_width − disc_width;
  }
  r = replace_count(cur_p);
  s = link(cur_p);
  while (r > 0) { ⟨ Add the width of node s to act_width 870 ⟩;
      decr(r);
      s = link(s);
  }
  prev_p = cur_p;
  cur_p = s;
  goto done5;
}
```

This code is used in section 865.

869. ⟨ Add the width of node s to *disc_width* 869 ⟩ ≡

```
if (is_char_node(s)) { f = font(s);
  disc_width = disc_width + char_width(f, char_info(f, character(s)));
}
else
  switch (type(s)) {
  case ligature_node:
    { f = font(lig_char(s));
      disc_width = disc_width + char_width(f, char_info(f, character(lig_char(s))));
    } break;
  case hlist_node: case vlist_node: case rule_node: case kern_node:
    disc_width = disc_width + width(s); break;
  default: confusion("disc3");
  }
```

This code is used in section 868.

870. ⟨ Add the width of node s to *act_width* 870 ⟩ ≡

if (*is_char_node*(s)) { f = *font*(s);
  act_width = *act_width* + *char_width*(f, *char_info*(f, *character*(s)));
}
else
  switch (*type*(s)) {
  case *ligature_node*:
    { f = *font*(*lig_char*(s));
      act_width = *act_width* + *char_width*(f, *char_info*(f, *character*(*lig_char*(s))));
    } **break**;
  case *hlist_node*: **case** *vlist_node*: **case** *rule_node*: **case** *kern_node*:
    act_width = *act_width* + *width*(s); **break**;
  default: *confusion*("disc4");
  }

This code is used in section 868.

871. 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 ⟩ +≡
  static pointer *best_bet*; /∗ use this passive node and its predecessors ∗/
  static int *fewest_demerits*; /∗ the demerits associated with *best_bet* ∗/
  static halfword *best_line*; /∗ line number following the last line of the new paragraph ∗/
  static int *actual_looseness*;
    /∗ the difference between *line_number*(*best_bet*) and the optimum *best_line* ∗/
  static int *line_diff*; /∗ the difference between the current line number and the optimum *best_line* ∗/

872. ⟨ Try the final line break at the end of the paragraph, and **goto** *done* if the desired breakpoints have been found 872 ⟩ ≡

{ *try_break*(*eject_penalty*, *hyphenated*);
  if (*link*(*active*) ≠ *last_active*) { ⟨ Find an active node with fewest demerits 873 ⟩;
    if (*looseness* ≡ 0) **goto** *done*;
    ⟨ Find the best active node for the desired looseness 874 ⟩;
    if ((*actual_looseness* ≡ *looseness*) ∨ *final_pass*) **goto** *done*;
  }
}

This code is used in section 862.

873. ⟨ Find an active node with fewest demerits 873 ⟩ ≡

r = *link*(*active*);
fewest_demerits = *awful_bad*;
do {
  if (*type*(r) ≠ *delta_node*)
    if (*total_demerits*(r) < *fewest_demerits*) { *fewest_demerits* = *total_demerits*(r);
      best_bet = r;
    }
  r = *link*(r);
} **while** (¬(r ≡ *last_active*)); *best_line* = *line_number*(*best_bet*)

This code is used in section 872.

874. 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 874 ⟩ ≡

```
{ r = link(active);
  actual_looseness = 0;
  do {
    if (type(r) ≠ delta_node) { line_diff = line_number(r) − best_line;
      if (((line_diff < actual_looseness) ∧ (looseness ≤ line_diff)) ∨
            ((line_diff > actual_looseness) ∧ (looseness ≥ line_diff))) { best_bet = r;
        actual_looseness = line_diff;
        fewest_demerits = total_demerits(r);
      }
      else if ((line_diff ≡ actual_looseness) ∧
            (total_demerits(r) < fewest_demerits)) { best_bet = r;
        fewest_demerits = total_demerits(r);
      }
    }
    r = link(r);
  } while (¬(r ≡ last_active));
  best_line = line_number(best_bet);
}
```

This code is used in section 872.

875. 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 875 ⟩ ≡

```
post_line_break(final_widow_penalty)
```

This code is used in section 814.

876. 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 825⟩ +≡
  static void post_line_break(int final_widow_penalty)
  { pointer q, r, s;   /* temporary registers for list manipulation */
    bool disc_break;   /* was the current break at a discretionary node? */
    bool post_disc_break;   /* and did it have a nonempty post-break part? */
    scaled cur_width;   /* width of line number cur_line */
    scaled cur_indent;   /* left margin of line number cur_line */
    quarterword t;   /* used for replacement counts in discretionary nodes */
    int pen;   /* use when calculating penalties between lines */
    halfword cur_line;   /* the current line number being justified */

    ⟨Reverse the links of the relevant passive nodes, setting cur_p to the first breakpoint 877⟩;
    cur_line = prev_graf + 1;
    do { ⟨Justify the line ending at breakpoint cur_p, and append it to the current vertical list, together
            with associated penalties and other insertions 879⟩;
      incr(cur_line);
      cur_p = next_break(cur_p);
      if (cur_p ≠ null)
        if (¬post_disc_break) ⟨Prune unwanted nodes at the beginning of the next line 878⟩;
    } while (¬(cur_p ≡ null));
    if ((cur_line ≠ best_line) ∨ (link(temp_head) ≠ null)) confusion("line␣breaking");
    prev_graf = best_line − 1;
  }
```

877. 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 877⟩ ≡
  q = break_node(best_bet);
  cur_p = null;
  do { r = q;
    q = prev_break(q);
    next_break(r) = cur_p;
    cur_p = r;
  } while (¬(q ≡ null))
```

This code is used in section 876.

878. 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 878⟩ ≡
{ $r = temp\_head$; **loop** { $q = link(r)$;
if $(q \equiv cur\_break(cur\_p))$ **goto** *done1*; /* $cur\_break(cur\_p)$ is the next breakpoint */
/* now q cannot be *null* */
if $(is\_char\_node(q))$ **goto** *done1*;
if $(non\_discardable(q))$ **goto** *done1*;
if $(type(q) \equiv kern\_node)$ **if** ($subtype(q) \neq$ **explicit**) **goto** *done1*;
$r = q$; /* now $type(q) \equiv glue\_node$, *kern_node*, *math_node*, or *penalty_node* */
}
done1:
if $(r \neq temp\_head)$ { $link(r) = null$;
$flush\_node\_list(link(temp\_head))$;
$link(temp\_head) = q$;
}
}

This code is used in section 876.

879. 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 879⟩ ≡
⟨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* 880⟩;
⟨Put the `\leftskip` glue at the left and detach this line 886⟩;
⟨Call the packaging subroutine, setting *just_box* to the justified box 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 887⟩;
⟨Append a penalty node, if a nonzero penalty is appropriate 889⟩

This code is used in section 876.

880. 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* 880 ⟩ ≡

```
q = cur_break(cur_p);
disc_break = false;
post_disc_break = false;
if (q ≠ null)    /* q cannot be a char_node */
  if (type(q) ≡ glue_node) { delete_glue_ref(glue_ptr(q));
    glue_ptr(q) = right_skip;
    subtype(q) = right_skip_code + 1;
    add_glue_ref(right_skip);
    goto done;
  }
  else { if (type(q) ≡ disc_node) ⟨ Change discretionary to compulsory and set disc_break: = true 881 ⟩
    else if ((type(q) ≡ math_node) ∨ (type(q) ≡ kern_node)) width(q) = 0;
  }
else { q = temp_head;
  while (link(q) ≠ null) q = link(q);
}
⟨ Put the \rightskip glue after node q 885 ⟩;
done:
```

This code is used in section 879.

881. ⟨ Change discretionary to compulsory and set *disc_break*: = *true* 881 ⟩ ≡

```
{ t = replace_count(q);
  ⟨ Destroy the t nodes following q, and make r point to the following node 882 ⟩;
  if (post_break(q) ≠ null) ⟨ Transplant the post-break list 883 ⟩;
  if (pre_break(q) ≠ null) ⟨ Transplant the pre-break list 884 ⟩;
  link(q) = r;
  disc_break = true;
}
```

This code is used in section 880.

882. ⟨ Destroy the t nodes following q, and make r point to the following node 882 ⟩ ≡

```
if (t ≡ 0) r = link(q);
else { r = q;
  while (t > 1) { r = link(r);
    decr(t);
  }
  s = link(r);
  r = link(s);
  link(s) = null;
  flush_node_list(link(q));
  replace_count(q) = 0;
}
```

This code is used in section 881.

883. 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 883 ⟩ ≡
{ $s = post\_break(q)$;
  while $(link(s) \neq null)$ $s = link(s)$;
  $link(s) = r$;
  $r = post\_break(q)$;
  $post\_break(q) = null$;
  $post\_disc\_break = true$;
}

This code is used in section 881.

884. 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 884 ⟩ ≡
{ $s = pre\_break(q)$;
  $link(q) = s$;
  while $(link(s) \neq null)$ $s = link(s)$;
  $pre\_break(q) = null$;
  $q = s$;
}

This code is used in section 881.

885. ⟨ Put the `\rightskip` glue after node q 885 ⟩ ≡
  $r = new\_param\_glue(right\_skip\_code)$;
  $link(r) = link(q)$;
  $link(q) = r$; $q = r$

This code is used in section 880.

886. 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 886 ⟩ ≡
  $r = link(q)$;
  $link(q) = null$;
  $q = link(temp\_head)$;
  $link(temp\_head) = r$;
  if $(left\_skip \neq zero\_glue)$ { $r = new\_param\_glue(left\_skip\_code)$;
    $link(r) = q$;
    $q = r$;
  }

This code is used in section 879.

887. ⟨ Append the new box to the current vertical list, followed by the list of special nodes taken out of the box by the packager 887 ⟩ ≡
  $append\_to\_vlist(just\_box)$;
  if $(adjust\_head \neq adjust\_tail)$ { $link(tail) = link(adjust\_head)$;
    $tail = adjust\_tail$;
  }
  $adjust\_tail = null$

This code is used in section 879.

888. 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.

$\langle$ Call the packaging subroutine, setting *just_box* to the justified box 888 $\rangle \equiv$

if $(cur\_line > last\_special\_line)$ { $cur\_width = second\_width$;
 $cur\_indent = second\_indent$;
}
else if $(par\_shape\_ptr \equiv null)$ { $cur\_width = first\_width$;
 $cur\_indent = first\_indent$;
}
else { $cur\_width = mem[par\_shape\_ptr + 2 * cur\_line].sc$;
 $cur\_indent = mem[par\_shape\_ptr + 2 * cur\_line - 1].sc$;
}
$adjust\_tail = adjust\_head$;
$just\_box = hpack(q, cur\_width, exactly)$; $shift\_amount(just\_box) = cur\_indent$

This code is used in section 879.

889. 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.

$\langle$ Append a penalty node, if a nonzero penalty is appropriate 889 $\rangle \equiv$

if $(cur\_line + 1 \neq best\_line)$ { $pen = inter\_line\_penalty$;
 if $(cur\_line \equiv prev\_graf + 1)$ $pen = pen + club\_penalty$;
 if $(cur\_line + 2 \equiv best\_line)$ $pen = pen + final\_widow\_penalty$;
 if $(disc\_break)$ $pen = pen + broken\_penalty$;
 if $(pen \neq 0)$ { $r = new\_penalty(pen)$;
  $link(tail) = r$;
  $tail = r$;
 }
}

This code is used in section 879.

890. 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 $lc\_code(c)\equiv 0$), a lowercase letter (if $lc\_code(c)\equiv c$), 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 890⟩ ≡

```
  {
#ifdef INIT
    if (trie_not_ready) init_trie();
#endif
    cur_lang = init_cur_lang;
    l_hyf = init_l_hyf;
    r_hyf = init_r_hyf;
    set_hyph_index;
  }
```

This code is used in section 862.

891. 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 ⟩ +≡
static int16_t *hc*[66]; /* word to be hyphenated */
static int *hn*; /* the number of positions occupied in *hc*; not always a **small_number** */
static pointer *ha*, *hb*; /* nodes *ha* .. *hb* should be replaced by the hyphenated result */
static internal_font_number *hf*; /* font number of the letters in *hc* */
static int16_t *hu*[64]; /* like *hc*, before conversion to lowercase */
static int *hyf_char*; /* hyphen character of the relevant font */
static ASCII_code *cur_lang*, *init_cur_lang*; /* current hyphenation table of interest */
static int *l_hyf*, *r_hyf*, *init_l_hyf*, *init_r_hyf*; /* limits on fragment sizes */
static halfword *hyf_bchar*; /* boundary character after c_n */

892. Hyphenation routines need a few more local variables.

⟨ Local variables for line breaking 861 ⟩ +≡
small_number *j*; /* an index into *hc* or *hu* */
int *c*; /* character being considered for hyphenation */

893. 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 893 ⟩ ≡
{ *prev_s* = *cur_p*;
s = *link*(*prev_s*);
if ($s \neq null$) { ⟨ Skip to node *ha*, or **goto** *done1* if no hyphenation should be attempted 895 ⟩;
if ($l\_hyf + r\_hyf > 63$) **goto** *done1*;
⟨ Skip to node *hb*, putting letters into *hu* and *hc* 896 ⟩;
⟨ Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto** *done1* 898 ⟩;
hyphenate( );
}
done1: ;
}

This code is used in section 865.

894. ⟨ Declare subprocedures for *line_break* 825 ⟩ +≡
⟨ Declare the function called *reconstitute* 905 ⟩
static void *hyphenate*(**void**)
{ ⟨ Local variables for hyphenation 900 ⟩
⟨ Find hyphen locations for the word in *hc*, or **return** 922 ⟩;
⟨ If no hyphens were found, **return** 901 ⟩;
⟨ Replace nodes *ha* .. *hb* by a sequence of nodes that includes the discretionary hyphens 902 ⟩;
}

895. 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 895⟩ ≡
```
loop { if (is_char_node(s)) { c = qo(character(s));
      hf = font(s);
    }
    else if (type(s) ≡ ligature_node)
      if (lig_ptr(s) ≡ null) goto resume;
      else { q = lig_ptr(s);
        c = qo(character(q));
        hf = font(q);
      }
    else if ((type(s) ≡ kern_node) ∧ (subtype(s) ≡ normal)) goto resume;
    else if (type(s) ≡ whatsit_node) { ⟨Advance past a whatsit node in the pre-hyphenation loop 1362⟩;
      goto resume;
    }
    else goto done1;
    set_lc_code(c);
    if (hc[0] ≠ 0)
      if ((hc[0] ≡ c) ∨ (uc_hyph > 0)) goto done2;
      else goto done1;
  resume: prev_s = s;
    s = link(prev_s);
  }
done2: hyf_char = hyphen_char[hf];
  if (hyf_char < 0) goto done1;
  if (hyf_char > 255) goto done1;
  ha = prev_s
```
This code is used in section 893.

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

⟨Skip to node *hb*, putting letters into *hu* and *hc* 896⟩ ≡

```
hn = 0;
loop { if (is_char_node(s)) { if (font(s) ≠ hf) goto done3;
      hyf_bchar = character(s);
      c = qo(hyf_bchar);
      set_lc_code(c);
      if (hc[0] ≡ 0) goto done3;
      if (hn ≡ 63) goto done3;
      hb = s;
      incr(hn);
      hu[hn] = c;
      hc[hn] = hc[0];
      hyf_bchar = non_char;
    }
    else if (type(s) ≡ ligature_node) ⟨Move the characters of a ligature node to hu and hc; but goto
          done3 if they are not all letters 897⟩
    else if ((type(s) ≡ kern_node) ∧ (subtype(s) ≡ normal)) { hb = s;
      hyf_bchar = font_bchar[hf];
    }
    else goto done3;
    s = link(s);
}
done3:
```

This code is used in section 893.

897. 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 $hn \equiv 0$ 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 897⟩ ≡

```
{ if (font(lig_char(s)) ≠ hf) goto done3;
    j = hn;
    q = lig_ptr(s); if (q > null) hyf_bchar = character(q);
    while (q > null) { c = qo(character(q));
      set_lc_code(c);
      if (hc[0] ≡ 0) goto done3;
      if (j ≡ 63) goto done3;
      incr(j);
      hu[j] = c;
      hc[j] = hc[0];
      q = link(q);
    }
    hb = s;
    hn = j;
    if (odd(subtype(s))) hyf_bchar = font_bchar[hf]; else hyf_bchar = non_char;
}
```

This code is used in section 896.

898. ⟨ Check that the nodes following *hb* permit hyphenation and that at least *l_hyf* + *r_hyf* letters have been found, otherwise **goto** *done1* 898 ⟩ ≡

```
if (hn < l_hyf + r_hyf) goto done1;     /* l_hyf and r_hyf are ≥ 1 */
loop { if (¬(is_char_node(s)))
      switch (type(s)) {
      case ligature_node: do_nothing; break;
      case kern_node:
        if (subtype(s) ≠ normal) goto done4; break;
      case whatsit_node: case glue_node: case penalty_node: case ins_node: case adjust_node:
        case mark_node: goto done4;
      default: goto done1;
      }
    s = link(s);
  }
  done4:
```

This code is used in section 893.

899. 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 ⟩ +≡
static int8_t *hyf* [65]; /∗ odd values indicate discretionary hyphens ∗/
static pointer *init_list*; /∗ list of punctuation characters preceding the word ∗/
static bool *init_lig*; /∗ does *init_list* represent a ligature? ∗/
static bool *init_lft*; /∗ if so, did the ligature involve a left boundary? ∗/

900. ⟨ Local variables for hyphenation 900 ⟩ ≡
int i, j, l; /∗ indices into hc or hu ∗/
pointer q, r, s; /∗ temporary registers for list manipulation ∗/
halfword *bchar*; /∗ boundary character of hyphenated word, or *non_char* ∗/

See also sections 911, 921, and 928.

This code is used in section 894.

901. 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** 901 ⟩ ≡
for ($j = l\_hyf$; $j \le hn - r\_hyf$; j++)
  if ($odd(hyf[j])$) **goto** *found1*;
return; *found1*:

This code is used in section 894.

902. 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 $j = 0$ if $hu[0]$ is to be part of the reconstruction; otherwise $j = 1$. 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 902⟩ ≡

```
  q = link(hb);
  link(hb) = null;
  r = link(ha);
  link(ha) = null;
  bchar = hyf_bchar;
  if (is_char_node(ha))
    if (font(ha) ≠ hf) goto found2;
    else { init_list = ha;
      init_lig = false;
      hu[0] = qo(character(ha));
    }
  else if (type(ha) ≡ ligature_node)
    if (font(lig_char(ha)) ≠ hf) goto found2;
    else { init_list = lig_ptr(ha);
      init_lig = true;
      init_lft = (subtype(ha) > 1);
      hu[0] = qo(character(lig_char(ha)));
      if (init_list ≡ null)
        if (init_lft) { hu[0] = 256;
          init_lig = false;
        }   /* in this case a ligature will be reconstructed from scratch */
      free_node(ha, small_node_size);
    }
  else {   /* no punctuation found; look for left boundary */
    if (¬is_char_node(r))
      if (type(r) ≡ ligature_node)
        if (subtype(r) > 1) goto found2;
    j = 1;
    s = ha;
    init_list = null;
    goto common_ending;
  }
  s = cur_p;   /* we have cur_p ≠ ha because type(cur_p) ≡ glue_node */
  while (link(s) ≠ ha) 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 912⟩;
  flush_list(init_list)
```

This code is used in section 894.

903. We must now face the fact that the battle is not over, even though the hyphens have been found: The process of reconstituting a word can be nontrivial because ligatures might change when a hyphen is present. *The TEXbook* discusses the difficulties of the word "difficult", and the discretionary material surrounding a hyphen can be considerably more complex than that. Suppose `abcdef` is a word in a font for which the only ligatures are `bc`, `cd`, `de`, and `ef`. If this word permits hyphenation between `b` and `c`, the two patterns with and without hyphenation are `a b - cd ef` and `a bc de f`. Thus the insertion of a hyphen might cause effects to ripple arbitrarily far into the rest of the word. A further complication arises if additional hyphens appear together with such rippling, e.g., if the word in the example just given could also be hyphenated between `c` and `d`; TEX avoids this by simply ignoring the additional hyphens in such weird cases.

Still further complications arise in the presence of ligatures that do not delete the original characters. When punctuation precedes the word being hyphenated, TEX's method is not perfect under all possible scenarios, because punctuation marks and letters can propagate information back and forth. For example, suppose the original pre-hyphenation pair `*a` changes to `*y` via a `|=:` ligature, which changes to `xy` via a `=:|` ligature; if $p_{a-1} =$ `x` and $p_a =$ `y`, the reconstitution procedure isn't smart enough to obtain `xy` again. In such cases the font designer should include a ligature that goes from `xa` to `xy`.

904. The processing is facilitated by a subroutine called *reconstitute*. Given a string of characters $x_j \ldots x_n$, there is a smallest index $m \ge j$ 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 $j \le k \le m$ 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 $j \equiv 0$: 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 ⟩ +≡
static small_number *hyphen_passed*; /∗ first hyphen in a ligature, if any ∗/

905. ⟨ Declare the function called *reconstitute* 905 ⟩ ≡

```
static small_number reconstitute(small_number j, small_number n, halfword bchar, halfword
        hchar)
{ pointer p;   /* temporary register for list manipulation */
  pointer t;   /* a node being appended to */
  four_quarters q;   /* character information or a lig/kern instruction */
  halfword cur_rh;   /* hyphen character for ligature testing */
  halfword test_char;   /* hyphen or other character for ligature testing */
  scaled w;   /* amount of kerning */
  font_index k;   /* position of current lig/kern instruction */
  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 907 ⟩;
resume: ⟨ If there's a ligature or kern at the cursor position, update the data structures, possibly
        advancing j; continue until the cursor moves 908 ⟩;
  ⟨ Append a ligature and/or kern to the translation; goto resume if the stack of inserted ligatures is
        nonempty 909 ⟩;
  return j;
}
```

This code is used in section 894.

906. 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

$$cur\_r = \begin{cases} character(lig\_stack), & \text{if } lig\_stack > null; \\ qi(hu[j+1]), & \text{if } lig\_stack \equiv null \text{ and } j < n; \\ bchar, & \text{if } lig\_stack \equiv null \text{ and } j \equiv n. \end{cases}$$

⟨ Global variables 13 ⟩ +≡

```
static halfword cur_l, cur_r;   /* characters before and after the cursor */
static pointer cur_q;   /* where a ligature should be detached */
static pointer lig_stack;   /* unfinished business to the right of the cursor */
static bool ligature_present;   /* should a ligature node be made for cur_l? */
static bool lft_hit, rt_hit;   /* did we hit a ligature with a boundary character? */
```

907. #**define** $append\_charnode\_to\_t(A)$
$\{$ $link(t) = get\_avail()$;
$t = link(t)$;
$font(t) = hf$;
$character(t) = A$;
$\}$
#**define** set_cur_r
$\{$ **if** $(j < n)$ $cur\_r = qi(hu[j+1])$; **else** $cur\_r = bchar$;
if $(odd(hyf[j]))$ $cur\_rh = hchar$; **else** $cur\_rh = non\_char$;
$\}$

⟨ Set up data structures with the cursor following position j 907 ⟩ ≡
$cur\_l = qi(hu[j])$;
$cur\_q = t$;
if $(j \equiv 0)$ $\{$ $ligature\_present = init\_lig$;
$p = init\_list$;
if $(ligature\_present)$ $lft\_hit = init\_lft$;
while $(p > null)$ $\{$ $append\_charnode\_to\_t(character(p))$;
$p = link(p)$;
$\}$
$\}$
else if $(cur\_l < non\_char)$ $append\_charnode\_to\_t(cur\_l)$;
$lig\_stack = null$; set_cur_r

This code is used in section 905.

908. 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 908⟩ ≡

```
if (cur_l ≡ non_char) { k = bchar_label[hf];
  if (k ≡ non_address) goto done; else q = font_info[k].qqqq;
}
else { q = char_info(hf, cur_l);
  if (char_tag(q) ≠ lig_tag) goto done;
  k = lig_kern_start(hf, q);
  q = font_info[k].qqqq;
  if (skip_byte(q) > stop_flag) { k = lig_kern_restart(hf, q);
    q = font_info[k].qqqq;
  }
}   /* now k is the starting address of the lig/kern program */
if (cur_rh < non_char) test_char = cur_rh; else test_char = cur_r;
loop { if (next_char(q) ≡ test_char)
    if (skip_byte(q) ≤ stop_flag)
      if (cur_rh < non_char) { hyphen_passed = j;
        hchar = non_char;
        cur_rh = non_char;
        goto resume;
      }
      else { if (hchar < non_char)
          if (odd(hyf[j])) { hyphen_passed = j;
            hchar = non_char;
          }
        if (op_byte(q) < kern_flag)
          ⟨Carry out a ligature replacement, updating the cursor structure and possibly advancing j;
              goto resume if the cursor doesn't advance, otherwise goto done 910⟩;
        w = char_kern(hf, q);
        goto done;   /* this kern will be inserted below */
      }
  if (skip_byte(q) ≥ stop_flag)
    if (cur_rh ≡ non_char) goto done;
    else { cur_rh = non_char;
      goto resume;
    }
  k = k + qo(skip_byte(q)) + 1;
  q = font_info[k].qqqq;
}
done:
```

This code is used in section 905.

909. #**define** *wrap_lig*(*A*)

```
if (ligature_present) { p = new_ligature(hf, cur_l, link(cur_q));
  if (lft_hit) { subtype(p) = 2;
    lft_hit = false;
  }
  if (A)
    if (lig_stack ≡ null) { incr(subtype(p));
      rt_hit = false;
    }
  link(cur_q) = p;
  t = p;
  ligature_present = false;
}
```

#**define** *pop_lig_stack*

```
{ if (lig_ptr(lig_stack) > null) { link(t) = lig_ptr(lig_stack);
      /* this is a charnode for hu[j + 1] */
    t = link(t);
    incr(j);
  }
  p = lig_stack;
  lig_stack = link(p);
  free_node(p, small_node_size);
  if (lig_stack ≡ null) set_cur_r else cur_r = character(lig_stack);
}     /* if lig_stack isn't null we have cur_rh ≡ non_char */
```

⟨ Append a ligature and/or kern to the translation; **goto** *resume* if the stack of inserted ligatures is nonempty 909 ⟩ ≡

```
wrap_lig(rt_hit);
if (w ≠ 0) { link(t) = new_kern(w);
  t = link(t);
  w = 0;
}
if (lig_stack > null) { cur_q = t;
  cur_l = character(lig_stack);
  ligature_present = true;
  pop_lig_stack;
  goto resume;
}
```

This code is used in section 905.

910. ⟨ Carry out a ligature replacement, updating the cursor structure and possibly advancing j; **goto** *resume* if the cursor doesn't advance, otherwise **goto** *done* 910 ⟩ ≡

```
{ if (cur_l ≡ non_char) lft_hit = true;
  if (j ≡ n)
    if (lig_stack ≡ null) rt_hit = true;
  check_interrupt;    /* allow a way out in case there's an infinite ligature loop */
  switch (op_byte(q)) {
  case qi(1): case qi(5):
    { cur_l = rem_byte(q);    /* =:|, =:|> */
      ligature_present = true;
    } break;
  case qi(2): case qi(6):
    { cur_r = rem_byte(q);    /* |=:, |=:> */
      if (lig_stack > null) character(lig_stack) = cur_r;
      else { lig_stack = new_lig_item(cur_r);
        if (j ≡ n) bchar = non_char;
        else { p = get_avail();
          lig_ptr(lig_stack) = p;
          character(p) = qi(hu[j + 1]);
          font(p) = hf;
        }
      }
    } break;
  case qi(3):
    { cur_r = rem_byte(q);    /* |=:| */
      p = lig_stack;
      lig_stack = new_lig_item(cur_r);
      link(lig_stack) = p;
    } break;
  case qi(7): case qi(11):
    { wrap_lig(false);    /* |=:|>, |=:|>> */
      cur_q = t;
      cur_l = rem_byte(q);
      ligature_present = true;
    } break;
  default:
    { cur_l = rem_byte(q);
      ligature_present = true;    /* =: */
      if (lig_stack > null) pop_lig_stack
      else if (j ≡ n) goto done;
      else { append_charnode_to_t(cur_r);
        incr(j);
        set_cur_r;
      }
    }
  }
  if (op_byte(q) > qi(4))
    if (op_byte(q) ≠ qi(7)) goto done;
  goto resume;
}
```

This code is used in section 908.

911. Okay, we're ready to insert the potential hyphenations that were found. When the following program is executed, we want to append the word $hu[1 \mathinner{..} hn]$ 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 900⟩ +≡
  pointer *major_tail*, *minor_tail*;
    /∗ the end of lists in the main and discretionary branches being reconstructed ∗/
  ASCII_code *c*; /∗ character temporarily replaced by a hyphen ∗/
  int *c_loc*; /∗ where that character came from ∗/
  int *r_count*; /∗ replacement count for discretionary ∗/
  pointer *hyf_node*; /∗ the hyphen, if it exists ∗/

912. When the following code is performed, $hyf[0]$ and $hyf[hn]$ will be zero.

⟨Reconstitute nodes for the hyphenated word, inserting discretionary hyphens 912⟩ ≡
  do { $l = j$;
    $j = reconstitute(j, hn, bchar, qi(hyf\_char)) + 1$;
    if $(hyphen\_passed \equiv 0)$ { $link(s) = link(hold\_head)$;
      while $(link(s) > null)$ $s = link(s)$;
      if $(odd(hyf[j-1]))$ { $l = j$;
        $hyphen\_passed = j - 1$;
        $link(hold\_head) = null$;
      }
    }
    if $(hyphen\_passed > 0)$
      ⟨Create and append a discretionary node as an alternative to the unhyphenated word, and continue to develop both branches until they become equivalent 913⟩;
  } **while** $(\neg(j > hn))$; $link(s) = q$

This code is used in section 902.

913. 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*
  { *major_tail* = *link*(*major_tail*);
   incr(*r_count*);
  }

⟨Create and append a discretionary node as an alternative to the unhyphenated word, and continue to develop both branches until they become equivalent 913⟩ ≡
 do { $r = get\_node(small\_node\_size)$;
  $link(r) = link(hold\_head)$;
  $type(r) = disc\_node$;
  $major\_tail = r$;
  $r\_count = 0$;
  while $(link(major\_tail) > null)$ *advance_major_tail*;
  $i = hyphen\_passed$;
  $hyf[i] = 0$;
  ⟨Put the characters $hu[l\mathrel{..}i]$ and a hyphen into $pre\_break(r)$ 914⟩;
  ⟨Put the characters $hu[i+1\mathrel{..}]$ into $post\_break(r)$, appending to this list and to *major_tail* until synchronization has been achieved 915⟩;
  ⟨Move pointer s to the end of the current list, and set $replace\_count(r)$ appropriately 917⟩;
  $hyphen\_passed = j-1$;
  $link(hold\_head) = null$;
 } **while** $(\neg(\neg odd(hyf[j-1])))$

This code is used in section 912.

914. 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 $hu[l\mathrel{..}i]$ and a hyphen into $pre\_break(r)$ 914⟩ ≡
 $minor\_tail = null$;
 $pre\_break(r) = null$;
 $hyf\_node = new\_character(hf, hyf\_char)$;
 if $(hyf\_node \ne null)$ { $incr(i)$;
  $c = hu[i]$;
  $hu[i] = hyf\_char$;
  $free\_avail(hyf\_node)$;
 }
 while $(l \le i)$ { $l = reconstitute(l, i, font\_bchar[hf], non\_char) + 1$;
  if $(link(hold\_head) > null)$ { **if** $(minor\_tail \equiv null)$ $pre\_break(r) = link(hold\_head)$;
   else $link(minor\_tail) = link(hold\_head)$;
   $minor\_tail = link(hold\_head)$;
   while $(link(minor\_tail) > null)$ $minor\_tail = link(minor\_tail)$;
  }
 }
 if $(hyf\_node \ne null)$ { $hu[i] = c$;   /* restore the character in the hyphen position */
  $l = i$;
  $decr(i)$;
 }

This code is used in section 913.

915. The synchronization algorithm begins with $l \equiv i+1 \le j$.

⟨ Put the characters $hu[i+1\,..]$ into $post\_break(r)$, appending to this list and to $major\_tail$ until synchronization has been achieved 915 ⟩ ≡

```
minor_tail = null;
post_break(r) = null;
c_loc = 0;
if (bchar_label[hf] ≠ non_address)   /* put left boundary at beginning of new line */
{ decr(l);
  c = hu[l];
  c_loc = l;
  hu[l] = 256;
}
while (l < j) { do { l = reconstitute(l, hn, bchar, non_char) + 1;
    if (c_loc > 0) { hu[c_loc] = c;
      c_loc = 0;
    }
    if (link(hold_head) > null) { if (minor_tail ≡ null) post_break(r) = link(hold_head);
      else link(minor_tail) = link(hold_head);
      minor_tail = link(hold_head);
      while (link(minor_tail) > null) minor_tail = link(minor_tail);
    }
  } while (¬(l ≥ j));
  while (l > j) ⟨ Append characters of hu [ j .. ] to major_tail, advancing j 916 ⟩;
}
```

This code is used in section 913.

916. ⟨ Append characters of hu [j ..] to $major\_tail$, advancing j 916 ⟩ ≡

```
{ j = reconstitute(j, hn, bchar, non_char) + 1;
  link(major_tail) = link(hold_head);
  while (link(major_tail) > null) advance_major_tail;
}
```

This code is used in section 915.

917. 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 917 ⟩ ≡

```
if (r_count > 127)   /* we have to forget the discretionary hyphen */
{ link(s) = link(r);
  link(r) = null;
  flush_node_list(r);
}
else { link(s) = r;
  replace_count(r) = r_count;
}
s = major_tail
```

This code is used in section 913.

918. Hyphenation. When a word $hc[1\mathrel{..}hn]$ has been set up to contain a candidate for hyphenation, TEX first looks to see if it is in the user's exception dictionary. If not, hyphens are inserted based on patterns that appear within the given word, using an algorithm due to Frank M. Liang.

Let's consider Liang's method first, since it is much more interesting than the exception-lookup routine. The algorithm begins by setting $hyf[j]$ to zero for all j, and invalid characters are inserted into $hc[0]$ and $hc[hn+1]$ to serve as delimiters. Then a reasonably fast method is used to see which of a given set of patterns occurs in the word $hc[0\mathrel{..}(hn+1)]$. Each pattern $p_1\ldots p_k$ of length k has an associated sequence of $k+1$ numbers $n_0\ldots n_k$; and if the pattern occurs in $hc[(j+1)\mathrel{..}(j+k)]$, TEX will set $hyf[j+i]=\max(hyf[j+i], n\_i)$ 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.

919. 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 $z\_i = trie\_link(z\_i-1)+p\_i$; 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 $hc[(l-k+1)\mathrel{..}l]$ of language t, we perform all of the required operations for this pattern by carrying out the following little program: Set $v=trie\_op(z\_k)$. Then set $v=v+op\_start[t]$, $hyf[l-hyf\_distance[v]]=\max(hyf[l-hyf\_distance[v]], hyf\_num[v])$, and $v=hyf\_next[v]$; repeat, if necessary, until $v\equiv min\_quarterword$.

⟨ Types in the outer block 18 ⟩ +≡
  typedef int32_t trie_pointer; /* an index into *trie* */

920. **#define** *trie_link*(*A*) *trie*[*A*].*rh* /* "downward" link in a trie */
#define *trie_char*(*A*) *trie*[*A*].*b1* /* character matched at this trie location */
#define *trie_op*(*A*) *trie*[*A*].*b0* /* program for hyphenation at this trie location */

⟨ Global variables 13 ⟩ +≡
  static two_halves *trie*[*trie_size* + 1]; /* *trie_link*, *trie_char*, *trie_op* */
  static small_number *hyf_distance0*[*trie_op_size*], ***const** *hyf_distance* = *hyf_distance0* − 1;
    /* position $k-j$ of n_j */
  static small_number *hyf_num0*[*trie_op_size*], ***const** *hyf_num* = *hyf_num0* − 1; /* value of n_j */
  static quarterword *hyf_next0*[*trie_op_size*], ***const** *hyf_next* = *hyf_next0* − 1;
    /* continuation code */
  static uint16_t *op_start*[256]; /* offset for current language */

921. ⟨ Local variables for hyphenation 900 ⟩ +≡
  trie_pointer *z*; /* an index into *trie* */
  int *v*; /* an index into *hyf_distance*, etc. */

922. 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** 922⟩ ≡
  for $(j = 0;\ j \le hn;\ j{+}{+})$ $hyf[j] = 0;$
  ⟨Look for the word $hc[1\,..\,hn]$ in the exception table, and **goto** *found* (with *hyf* containing the hyphens) if an entry is found 929⟩;
  if $(trie\_char(cur\_lang + 1) \ne qi(cur\_lang))$ **return**;   /∗ no patterns for *cur_lang* ∗/
  $hc[0] = 0;$
  $hc[hn + 1] = 0;$
  $hc[hn + 2] = 256;$ /∗ insert delimiters ∗/
  for $(j = 0;\ j \le hn - r\_hyf + 1;\ j{+}{+})$ { $z = trie\_link(cur\_lang + 1) + hc[j];$
   $l = j;$
   while $(hc[l] \equiv qo(trie\_char(z)))$ { **if** $(trie\_op(z) \ne min\_quarterword)$
     ⟨Store maximum values in the *hyf* table 923⟩;
    $incr(l);$
    $z = trie\_link(z) + hc[l];$
   }
  }
found:
  for $(j = 0;\ j \le l\_hyf - 1;\ j{+}{+})$ $hyf[j] = 0;$
  for $(j = 0;\ j \le r\_hyf - 1;\ j{+}{+})$ $hyf[hn - j] = 0$

This code is used in section 894.

923. ⟨Store maximum values in the *hyf* table 923⟩ ≡
  { $v = trie\_op(z);$
   do { $v = v + op\_start[cur\_lang];$
    $i = l - hyf\_distance[v];$
    if $(hyf\_num[v] > hyf[i])$ $hyf[i] = hyf\_num[v];$
    $v = hyf\_next[v];$
   } **while** $(\neg(v \equiv min\_quarterword));$
  }

This code is used in section 922.

924. 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 $\le \alpha$. 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⟩ +≡
  typedef int16_t hyph_pointer;   /∗ an index into the ordered hash table ∗/

925. ⟨Global variables 13⟩ +≡
  static str_number $hyph\_word[hyph\_size + 1];$ /∗ exception words ∗/
  static pointer $hyph\_list[hyph\_size + 1];$ /∗ lists of hyphen positions ∗/
  static hyph_pointer $hyph\_count;$ /∗ the number of words in the exception dictionary ∗/

926. ⟨ Local variables for initialization 19 ⟩ +≡
int z; /* runs through the exception dictionary */

927. ⟨ Set initial values of key variables 21 ⟩ +≡
for ($z = 0$; $z \le hyph\_size$; z++) { $hyph\_word[z] = 0$;
  $hyph\_list[z] = null$;
}
$hyph\_count = 0$;

928. 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 900 ⟩ +≡
hyph_pointer h; /* an index into *hyph_word* and *hyph_list* */
str_number k; /* an index into *str_start* */
pool_pointer u; /* an index into *str_pool* */

929. 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 929 ⟩ ≡
$h = hc[1]$;
$incr(hn)$;
$hc[hn] = cur\_lang$;
for ($j = 2$; $j \le hn$; j++) $h = (h + h + hc[j]) \% hyph\_size$;
loop { ⟨ 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* 930 ⟩;
  if ($h > 0$) $decr(h)$; **else** $h = hyph\_size$;
}
not_found: $decr(hn)$

This code is used in section 922.

930. ⟨ 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* 930 ⟩ ≡
$k = hyph\_word[h]$;
if ($k \equiv 0$) **goto** *not_found*;
if ($length(k) < hn$) **goto** *not_found*;
if ($length(k) \equiv hn$) { $j = 1$;
  $u = str\_start[k]$;
  do {
    if ($so(str\_pool[u]) < hc[j]$) **goto** *not_found*;
    if ($so(str\_pool[u]) > hc[j]$) **goto** *done*;
    $incr(j)$;
    $incr(u)$;
  } **while** ($\neg(j > hn)$);
  ⟨ Insert hyphens as specified in $hyph\_list[h]$ 931 ⟩;
  $decr(hn)$;
  goto *found*;
}
done:

This code is used in section 929.

931. ⟨ Insert hyphens as specified in *hyph_list*[*h*] 931 ⟩ ≡

```
s = hyph_list[h];
while (s ≠ null) { hyf[info(s)] = 1;
  s = link(s);
}
```

This code is used in section 930.

932. ⟨ Search *hyph_list* for pointers to *p* 932 ⟩ ≡

```
for (q = 0; q ≤ hyph_size; q++) { if (hyph_list[q] ≡ p) { print_nl("HYPH(");
      print_int(q);
      print_char(')');
    }
}
```

This code is used in section 171.

933. 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) cur_lang = 0;
        else if (language > 255) cur_lang = 0;
        else cur_lang = language

static void new_hyph_exceptions(void)    /* enters new exceptions */
{ int n;    /* length of current word; not always a small_number */
  int j;    /* an index into hc */
  hyph_pointer h;    /* an index into hyph_word and hyph_list */
  str_number k;    /* an index into str_start */
  pointer p;    /* head of a list of hyphen positions */
  pointer q;    /* used when creating a new node for list p */
  str_number s, t;    /* strings being compared or stored */
  pool_pointer u, v;    /* indices into str_pool */

  scan_left_brace();    /* a left brace must follow \hyphenation */
  set_cur_lang;
#ifdef INIT
  if (trie_not_ready) { hyph_index = 0;
    goto not_found1;
  }
#endif
  set_hyph_index;
not_found1:
  ⟨ Enter as many hyphenation exceptions as are listed, until coming to a right brace; then return 934 ⟩;
}
```

934. ⟨Enter as many hyphenation exceptions as are listed, until coming to a right brace; then **return** 934⟩ ≡

```
n = 0;
p = null;
loop { get_x_token();
reswitch:
  switch (cur_cmd) {
  case letter: case other_char: case char_given: ⟨Append a new letter or hyphen 936⟩ break;
  case char_num:
    { scan_char_num();
      cur_chr = cur_val;
      cur_cmd = char_given;
      goto reswitch;
    }
  case spacer: case right_brace:
    { if (n > 1) ⟨Enter a hyphenation exception 938⟩;
      if (cur_cmd ≡ right_brace) return;
      n = 0;
      p = null;
    } break;
  default: ⟨Give improper \hyphenation error 935⟩
  }
}
```

This code is used in section 933.

935. ⟨Give improper `\hyphenation` error 935⟩ ≡

```
{ 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();
}
```

This code is used in section 934.

936. ⟨Append a new letter or hyphen 936⟩ ≡

```
if (cur_chr ≡ '-') ⟨Append the value n to list p 937⟩
else { set_lc_code(cur_chr);
  if (hc[0] ≡ 0) { print_err("Not␣a␣letter");
    help2("Letters␣in␣\\hyphenation␣words␣must␣have␣\\lccode>0.",
    "Proceed;␣I'll␣ignore␣the␣character␣I␣just␣read.");
    error();
  }
  else if (n < 63) { incr(n);
    hc[n] = hc[0];
  }
}
```

This code is used in section 934.

937. ⟨Append the value n to list p 937⟩ ≡

{ **if** $(n < 63)$ { $q = get\_avail()$;
    $link(q) = p$;
    $info(q) = n$;
    $p = q$;
  }
}

This code is used in section 936.

938. ⟨Enter a hyphenation exception 938⟩ ≡

{ $incr(n)$;
  $hc[n] = cur\_lang$;
  $str\_room(n)$;
  $h = 0$;
  for $(j = 1;\ j \le n;\ j{+}{+})$ { $h = (h + h + hc[j]) \,\%\, hyph\_size$;
    $append\_char(hc[j])$;
  }
  $s = make\_string()$;
  ⟨Insert the pair (s, p) into the exception table 939⟩;
}

This code is used in section 934.

939. ⟨Insert the pair (s, p) into the exception table 939⟩ ≡

if $(hyph\_count \equiv hyph\_size)$ $overflow($"exception␣dictionary"$, hyph\_size)$;
$incr(hyph\_count)$;
while $(hyph\_word[h] \ne 0)$ { ⟨If the string $hyph\_word[h]$ is less than or equal to s, interchange $(hyph\_word[h], hyph\_list[h])$ with (s, p) 940⟩;
  if $(h > 0)$ $decr(h)$; **else** $h = hyph\_size$;
}
$hyph\_word[h] = s$; $hyph\_list[h] = p$

This code is used in section 938.

940. ⟨If the string $hyph\_word[h]$ is less than or equal to s, interchange $(hyph\_word[h], hyph\_list[h])$ with (s, p) 940⟩ ≡

$k = hyph\_word[h]$;
if $(length(k) < length(s))$ **goto** $found$;
if $(length(k) > length(s))$ **goto** not_found;
$u = str\_start[k]$;
$v = str\_start[s]$;
do {
  if $(str\_pool[u] < str\_pool[v])$ **goto** $found$;
  if $(str\_pool[u] > str\_pool[v])$ **goto** not_found;
  $incr(u)$;
  $incr(v)$;
} **while** $(\neg(u \equiv str\_start[k + 1]))$;
$found$: $q = hyph\_list[h]$;
$hyph\_list[h] = p$;
$p = q$;
$t = hyph\_word[h]$;
$hyph\_word[h] = s$;
$s = t$; not_found:

This code is used in section 939.

941. 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* 825 ⟩ +≡
#ifdef `INIT`
  ⟨ Declare procedures for preprocessing hyphenation patterns 943 ⟩
#endif

942. 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 $hyf\_distance[v] \equiv 3$, $hyf\_num[v] \equiv 2$, and $hyf\_next[v] \equiv v'$, where the auxiliary *trie_op* code v' has $hyf\_distance[v'] \equiv 0$, $hyf\_num[v'] \equiv 1$, and $hyf\_next[v'] \equiv min\_quarterword$.

TeX computes an appropriate value v with the *new_trie_op* subroutine below, by setting

$$v' = new\_trie\_op(0, 1, min\_quarterword), \qquad v = new\_trie\_op(3, 2, v').$$

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 ⟩ +≡
#ifdef `INIT`
  static uint16_t *trie_op_hash0* [*trie_op_size* + *trie_op_size* + 1],
    ∗**const** *trie_op_hash* = *trie_op_hash0* + *trie_op_size*; /∗ trie op codes for quadruples ∗/
  static quarterword *trie_used* [256]; /∗ largest opcode used so far for this language ∗/
  static ASCII_code *trie_op_lang0* [*trie_op_size*], ∗**const** *trie_op_lang* = *trie_op_lang0* − 1;
   /∗ language part of a hashed quadruple ∗/
  static quarterword *trie_op_val0* [*trie_op_size*], ∗**const** *trie_op_val* = *trie_op_val0* − 1;
   /∗ opcode corresponding to a hashed quadruple ∗/
  static int *trie_op_ptr*; /∗ number of stored ops so far ∗/
#endif

943. 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 943 ⟩ ≡

```
static quarterword new_trie_op(small_number d, small_number n, quarterword v)
{ int h;    /* trial hash location */
  quarterword u;    /* trial op code */
  int l;    /* pointer to stored data */
  h = abs(n + 313 * d + 361 * v + 1009 * cur_lang) % (trie_op_size + trie_op_size) − trie_op_size;
  loop { l = trie_op_hash[h];
    if (l ≡ 0)    /* empty position found for a new op */
    { if (trie_op_ptr ≡ trie_op_size) overflow("pattern␣memory␣ops", trie_op_size);
      u = trie_used[cur_lang];
      if (u ≡ max_quarterword)
        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;
      return u;
    }
    if ((hyf_distance[l] ≡ d) ∧ (hyf_num[l] ≡ n) ∧ (hyf_next[l] ≡ v) ∧ (trie_op_lang[l] ≡ cur_lang)) {
      return trie_op_val[l];
    }
    if (h > −trie_op_size) decr(h); else h = trie_op_size;
  }
}
```

See also sections 947, 948, 952, 956, 958, 959, and 965.

This code is used in section 941.

944. 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 944 ⟩ ≡

$op\_start[0] = -min\_quarterword$;
for $(j = 1;\ j \le 255;\ j{+}{+})$ $op\_start[j] = op\_start[j-1] + qo(trie\_used[j-1])$;
for $(j = 1;\ j \le trie\_op\_ptr;\ j{+}{+})$ $trie\_op\_hash[j] = op\_start[trie\_op\_lang[j]] + trie\_op\_val[j]$;
  /∗ destination ∗/
for $(j = 1;\ j \le trie\_op\_ptr;\ j{+}{+})$
  while $(trie\_op\_hash[j] > j)$ { $k = trie\_op\_hash[j]$;
    $t = hyf\_distance[k]$;
    $hyf\_distance[k] = hyf\_distance[j]$;
    $hyf\_distance[j] = t$;
    $t = hyf\_num[k]$;
    $hyf\_num[k] = hyf\_num[j]$;
    $hyf\_num[j] = t$;
    $t = hyf\_next[k]$;
    $hyf\_next[k] = hyf\_next[j]$;
    $hyf\_next[j] = t$;
    $trie\_op\_hash[j] = trie\_op\_hash[k]$;
    $trie\_op\_hash[k] = k$;
  }

This code is used in section 951.

945. 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) 163 ⟩ +≡

for $(k = -trie\_op\_size;\ k \le trie\_op\_size;\ k{+}{+})$ $trie\_op\_hash[k] = 0$;
for $(k = 0;\ k \le 255;\ k{+}{+})$ $trie\_used[k] = min\_quarterword$;
$trie\_op\_ptr = 0$;

946. 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

$$trie\_c[trie\_r[z]] > trie\_c[z] \qquad \text{whenever } z \ne 0 \text{ and } trie\_r[z] \ne 0;$$

in other words, sibling nodes are ordered by their c fields.

#define *trie_root* *trie_l*[0] /∗ root of the linked trie ∗/

⟨ Global variables 13 ⟩ +≡

#ifdef INIT
static packed_ASCII_code *trie_c*[*trie_size* + 1]; /∗ characters to match ∗/
  static quarterword *trie_o*[*trie_size* + 1]; /∗ operations to perform ∗/
  static trie_pointer *trie_l*[*trie_size* + 1]; /∗ left subtrie links ∗/
  static trie_pointer *trie_r*[*trie_size* + 1]; /∗ right subtrie links ∗/
  static trie_pointer *trie_ptr*; /∗ the number of nodes in the trie ∗/
  static trie_pointer *trie_hash*[*trie_size* + 1]; /∗ used to identify equivalent subtries ∗/
#endif

947. 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 943⟩ +≡

```
static trie_pointer trie_node(trie_pointer p)   /* converts to a canonical form */
{ trie_pointer h;   /* trial hash location */
  trie_pointer q;   /* trial trie node */
  h = abs(trie_c[p] + 1009 * trie_o[p] +
      2718 * trie_l[p] + 3142 * trie_r[p]) % trie_size;
  loop { q = trie_hash[h];
    if (q ≡ 0) { trie_hash[h] = p;
      return p;
    }
    if ((trie_c[q] ≡ trie_c[p]) ∧ (trie_o[q] ≡ trie_o[p]) ∧
          (trie_l[q] ≡ trie_l[p]) ∧ (trie_r[q] ≡ trie_r[p])) { return q;
    }
    if (h > 0) decr(h); else h = trie_size;
  }
}
```

948. 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 943⟩ +≡

```
static trie_pointer compress_trie(trie_pointer p)
{ if (p ≡ 0) return 0;
  else { trie_l[p] = compress_trie(trie_l[p]);
    trie_r[p] = compress_trie(trie_r[p]);
    return trie_node(p);
  }
}
```

949. 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(A)  trie[A].lh    /* backward links in trie holes */
```

⟨Global variables 13⟩ +≡

```
#ifdef INIT
  static bool trie_taken0[trie_size], *const trie_taken = trie_taken0 − 1;
    /* does a family start here? */
    static trie_pointer trie_min[256];    /* the first possible slot for each character */
    static trie_pointer trie_max;    /* largest location used in trie */
    static bool trie_not_ready;    /* is the trie still in linked form? */
#endif
```

950. 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) 163⟩ +≡

```
  trie_not_ready = true;
  trie_root = 0;
  trie_c[0] = si(0);
  trie_ptr = 0;
```

951. 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 951⟩ ≡

```
  ⟨Sort the hyphenation op tables into proper order 944⟩;
  for (p = 0; p ≤ trie_size; p++) trie_hash[p] = 0;
  hyph_root = compress_trie(hyph_root);
  trie_root = compress_trie(trie_root);    /* identify equivalent subtries */
  for (p = 0; p ≤ trie_ptr; p++) trie_ref[p] = 0;
  for (p = 0; p ≤ 255; p++) trie_min[p] = p + 1;
  trie_link(0) = 1; trie_max = 0
```

This code is used in section 965.

952. 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 $c \equiv trie\_c[p]$, 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 943⟩ +≡

```
static void first_fit(trie_pointer p)   /* packs a family into trie */
{ trie_pointer h;   /* candidate for trie_ref[p] */
  trie_pointer z;   /* runs through holes */
  trie_pointer q;   /* runs through the family starting at p */
  ASCII_code c;   /* smallest character in the family */
  trie_pointer l, r;   /* left and right neighbors */
  int ll;   /* upper limit of trie_min updating */
  c = so(trie_c[p]);
  z = trie_min[c];   /* get the first conceivably good hole */
  loop { h = z − c;
    ⟨Ensure that trie_max ≥ h + 256 953⟩;
    if (trie_taken[h]) goto not_found;
    ⟨If all characters of the family fit relative to h, then goto found, otherwise goto not_found 954⟩;
  not_found: z = trie_link(z);   /* move to the next hole */
  }
found: ⟨Pack the family into trie relative to h 955⟩;
}
```

953. By making sure that *trie_max* is at least $h + 256$, we can be sure that $trie\_max > z$, since $h \equiv z - c$. 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 ≥ h + 256 953⟩ ≡

```
if (trie_max < h + 256) { if (trie_size ≤ h + 256) overflow("pattern␣memory", trie_size);
  do { incr(trie_max);
    trie_taken[trie_max] = false;
    trie_link(trie_max) = trie_max + 1;
    trie_back(trie_max) = trie_max − 1;
  } while (¬(trie_max ≡ h + 256));
}
```

This code is used in section 952.

954. ⟨If all characters of the family fit relative to h, then **goto** *found*, otherwise **goto** *not_found* 954⟩ ≡

```
q = trie_r[p];
while (q > 0) { if (trie_link(h + so(trie_c[q])) ≡ 0) goto not_found;
  q = trie_r[q];
}
goto found
```

This code is used in section 952.

955. ⟨Pack the family into *trie* relative to h 955⟩ ≡

$trie\_taken[h] = true;$
$trie\_ref[p] = h;$
$q = p;$
do { $z = h + so(trie\_c[q]);$
  $l = trie\_back(z);$
  $r = trie\_link(z);$
  $trie\_back(r) = l;$
  $trie\_link(l) = r;$
  $trie\_link(z) = 0;$
  if $(l < 256)$ { **if** $(z < 256)$ $ll = z;$ **else** $ll = 256;$
    do { $trie\_min[l] = r;$
      $incr(l);$
    } **while** $(\neg(l \equiv ll));$
  }
  $q = trie\_r[q];$
} **while** $(\neg(q \equiv 0))$

This code is used in section 952.

956. To pack the entire linked trie, we use the following recursive procedure.

⟨Declare procedures for preprocessing hyphenation patterns 943⟩ +≡

static void *trie_pack*(**trie_pointer** p) /∗ pack subtries of a family ∗/
{ **trie_pointer** q; /∗ a local variable that need not be saved on recursive calls ∗/
  do { $q = trie\_l[p];$
    if $((q > 0) \land (trie\_ref[q] \equiv 0))$ { $first\_fit(q);$
      $trie\_pack(q);$
    }
    $p = trie\_r[p];$
  } **while** $(\neg(p \equiv 0));$
}

957. 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* 957⟩ ≡
```
h.rh = 0;
h.b0 = min_quarterword;
h.b1 = min_quarterword;   /* trie_link = 0, trie_op = min_quarterword, trie_char = qi(0) */
if (trie_max ≡ 0)   /* no patterns were given */
{ for (r = 0; r ≤ 256; r++) trie[r] = h;
  trie_max = 256;
}
else { if (hyph_root > 0) trie_fix(hyph_root);
  if (trie_root > 0) trie_fix(trie_root);   /* this fixes the non-holes in trie */
  r = 0;   /* now we will zero out all the holes */
  do { s = trie_link(r);
    trie[r] = h;
    r = s;
  } while (¬(r > trie_max));
}
trie_char(0) = qi('?');   /* make trie_char(c) ≠ c for all c */
```
This code is used in section 965.

958. 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 943⟩ +≡
```
static void trie_fix(trie_pointer p)   /* moves p and its siblings into trie */
{ trie_pointer q;   /* a local variable that need not be saved on recursive calls */
  ASCII_code c;   /* another one that need not be saved */
  trie_pointer z;   /* trie reference; this local variable must be saved */

  z = trie_ref[p];
  do { q = trie_l[p];
    c = so(trie_c[p]);
    trie_link(z + c) = trie_ref[q];
    trie_char(z + c) = qi(c);
    trie_op(z + c) = trie_o[p];
    if (q > 0) trie_fix(q);
    p = trie_r[p];
  } while (¬(p ≡ 0));
}
```

959. 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 943 ⟩ +≡

```
static void new_patterns(void)   /* initializes the hyphenation pattern data */
{ int k, l;   /* indices into hc and hyf; not always in small_number range */
  bool digit_sensed;   /* should the next digit be treated as a letter? */
  quarterword v;   /* trie op code */
  trie_pointer p, q;   /* nodes of trie traversed during insertion */
  bool first_child;   /* is p ≡ trie_l[q]? */
  int c;   /* character being inserted */
  if (trie_not_ready) { 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 960 ⟩;
    if (saving_hyph_codes > 0) ⟨ Store hyphenation codes for current language 1525 ⟩;
  }
  else { print_err("Too␣late␣for␣");
    print_esc("patterns");
    help1("All␣patterns␣must␣be␣given␣before␣typesetting␣begins.");
    error();
    link(garbage) = scan_toks(false, false);
    flush_list(def_ref);
  }
}
```

960. 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 960 ⟩ ≡

```
k = 0;
hyf[0] = 0;
digit_sensed = false;
loop { get_x_token();
  switch (cur_cmd) {
  case letter: case other_char: ⟨ Append a new letter or a hyphen level 961 ⟩ break;
  case spacer: case right_brace:
    { if (k > 0) ⟨ Insert a new pattern into the linked trie 962 ⟩;
      if (cur_cmd ≡ right_brace) goto done;
      k = 0;
      hyf[0] = 0;
      digit_sensed = false;
    } break;
  default:
    { print_err("Bad␣");
      print_esc("patterns");
      help1("(See␣Appendix␣H.)");
      error();
    }
  }
}
done:
```

This code is used in section 959.

961. ⟨ Append a new letter or a hyphen level 961 ⟩ ≡

```
if (digit_sensed ∨ (cur_chr < '0') ∨ (cur_chr > '9')) { if (cur_chr ≡ '.') cur_chr = 0;
        /* edge-of-word delimiter */
  else { cur_chr = lc_code(cur_chr);
    if (cur_chr ≡ 0) { print_err("Nonletter");
      help1("(See␣Appendix␣H.)");
      error( ) ;
    }
  }
  if (k < 63) { incr(k);
    hc[k] = cur_chr;
    hyf[k] = 0;
    digit_sensed = false;
  }
}
else if (k < 63) { hyf[k] = cur_chr − '0';
  digit_sensed = true;
}
```

This code is used in section 960.

962. 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 962 ⟩ ≡

```
{ ⟨ Compute the trie op code, v, and set l: = 0 964 ⟩;
  q = 0;
  hc[0] = cur_lang;
  while (l ≤ k) { c = hc[l];
    incr(l);
    p = trie_l[q];
    first_child = true;
    while ((p > 0) ∧ (c > so(trie_c[p]))) { q = p;
      p = trie_r[q];
      first_child = false;
    }
    if ((p ≡ 0) ∨ (c < so(trie_c[p])))
      ⟨ Insert a new trie node between q and p, and make p point to it 963 ⟩;
    q = p;    /* now node q represents p_1 ... p_{l−1} */
  }
  if (trie_o[q] ≠ min_quarterword) { print_err("Duplicate␣pattern");
    help1("(See␣Appendix␣H.)");
    error( ) ;
  }
  trie_o[q] = v;
}
```

This code is used in section 960.

963. ⟨Insert a new trie node between q and p, and make p point to it 963⟩ ≡

```
{ if (trie_ptr ≡ trie_size) overflow("pattern␣memory", trie_size);
  incr(trie_ptr);
  trie_r[trie_ptr] = p;
  p = trie_ptr;
  trie_l[p] = 0;
  if (first_child) trie_l[q] = p; else trie_r[q] = p;
  trie_c[p] = si(c);
  trie_o[p] = min_quarterword;
}
```

This code is used in sections 962, 1525, and 1526.

964. ⟨Compute the trie op code, v, and set l: = 0 964⟩ ≡

```
if (hc[1] ≡ 0) hyf[0] = 0;
if (hc[k] ≡ 0) hyf[k] = 0;
l = k;
v = min_quarterword;
loop { if (hyf[l] ≠ 0) v = new_trie_op(k − l, hyf[l], v);
  if (l > 0) decr(l); else goto done1;
}
done1:
```

This code is used in section 962.

965. 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 943⟩ +≡

```
static void init_trie(void)
{ int p;   /∗ pointer for initialization ∗/
  int j, k, t;   /∗ all-purpose registers for initialization ∗/
  int r, s;   /∗ used to clean up the packed trie ∗/
  two_halves h;   /∗ template used to zero out trie's holes ∗/
  ⟨Local variables to save the profiling context 1761⟩
  ⟨Charge the time used here on init_trie 1764⟩
  ⟨Get ready to compress the trie 951⟩;
  if (trie_root ≠ 0) { first_fit(trie_root);
    trie_pack(trie_root);
  }
  if (hyph_root ≠ 0) ⟨Pack all stored hyph_codes 1527⟩;
  ⟨Move the data into trie 957⟩;
  trie_not_ready = false;
  ⟨restore the previous current file, line, and command 1763⟩
}
```

966. 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.

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

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

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

```
static pointer prune_page_top(pointer p, bool s)   /* adjust top after page break */
{ pointer prev_p;   /* lags one step behind p */
  pointer q, r;   /* temporary variables for list manipulation */
  prev_p = temp_head;
  link(temp_head) = p;
  while (p ≠ null)
    switch (type(p)) {
    case hlist_node: case vlist_node: case rule_node:
      ⟨Insert glue for split_top_skip and set p: = null 968⟩ break;
    case whatsit_node: case mark_node: case ins_node:
      { prev_p = p;
        p = link(prev_p);
      } break;
    case glue_node: case kern_node: case penalty_node:
      { q = p;
        p = link(q);
        link(q) = null;
        link(prev_p) = p;
        if (s) { if (split_disc ≡ null) split_disc = q; else link(r) = q;
          r = q;
        }
        else flush_node_list(q);
      } break;
    default: confusion("pruning");
    }
  return link(temp_head);
}
```

968. ⟨Insert glue for *split_top_skip* and set *p*: = *null* 968⟩ ≡

```
{ 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)) width(temp_ptr) = width(temp_ptr) − height(p);
  else width(temp_ptr) = 0;
  p = null;
}
```

This code is used in section 967.

969. 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(A) active_height[A] = 0 /* initialize the height to zero */

  static pointer vert_break(pointer p, scaled h, scaled d) /* finds optimum page break */
  { pointer prev_p; /* if p is a glue node, type(prev_p) determines whether p is a legal breakpoint */
    pointer q, r; /* glue specifications */
    int pi; /* penalty value */
    int b; /* badness at a trial breakpoint */
    int least_cost; /* the smallest badness plus penalties found so far */
    pointer best_place; /* the most recent break that leads to least_cost */
    scaled prev_dp; /* depth of previous box in the list */
    small_number t; /* type of the node following a kern */

    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 { ⟨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 971⟩;
      prev_p = p;
      p = link(prev_p);
    }
  done: return best_place;
  }
```

970. 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⟩ +≡
  static scaled best_height_plus_depth; /* height of the best box, without stretching or shrinking */
```

971. 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 971 ⟩ ≡
  if $(p \equiv null)$ $pi = eject\_penalty$;
  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 972 ⟩;
  ⟨ 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 973 ⟩;
  if $((type(p) < glue\_node) \lor (type(p) > kern\_node))$ **goto** *not_found*;
update_heights:
  ⟨ Update the current height and depth measurements with respect to a glue or kern node p 975 ⟩;
not_found:
  if $(prev\_dp > d)$ { $cur\_height = cur\_height + prev\_dp - d$;
    $prev\_dp = d$;
  }

This code is used in section 969.

972. ⟨ 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 972 ⟩ ≡
  switch $(type(p))$ {
  case *hlist_node*: **case** *vlist_node*: **case** *rule_node*:
    {
      $cur\_height = cur\_height + prev\_dp + height(p)$;
      $prev\_dp = depth(p)$;
      goto *not_found*;
    }
  case *whatsit_node*: ⟨ Process whatsit p in *vert_break* loop, **goto** *not_found* 1364 ⟩;
  case *glue_node*:
    if $(precedes\_break(prev\_p))$ $pi = 0$;
    else goto *update_heights*; **break**;
  case *kern_node*:
    { **if** $(link(p) \equiv null)$ $t = penalty\_node$;
      else $t = type(link(p))$;
      if $(t \equiv glue\_node)$ $pi = 0$; **else goto** *update_heights*;
    } **break**;
  case *penalty_node*: $pi = penalty(p)$; **break**;
  case *mark_node*: **case** *ins_node*: **goto** *not_found*;
  default: *confusion*("vertbreak");
  }

This code is used in section 971.

973. **#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 973 ⟩ ≡

if (*pi* < *inf_penalty*) { ⟨ Compute the badness, b, using *awful_bad* if the box is too full 974 ⟩;
  if (b < *awful_bad*)
    if (*pi* ≤ *eject_penalty*) b = *pi*;
    else if (b < *inf_bad*) $b = b +$ *pi*;
    else b = *deplorable*;
  if (b ≤ *least_cost*) { *best_place* = p;
    least_cost = b;
    best_height_plus_depth = *cur_height* + *prev_dp*;
  }
  if ((b ≡ *awful_bad*) ∨ (*pi* ≤ *eject_penalty*)) **goto** *done*;
}

This code is used in section 971.

974. ⟨ Compute the badness, b, using *awful_bad* if the box is too full 974 ⟩ ≡

if (*cur_height* < h)
  if ((*active_height*[3] ≠ 0) ∨ (*active_height*[4] ≠ 0) ∨ (*active_height*[5] ≠ 0)) b = 0;
  else b = *badness*(h − *cur_height*, *active_height*[2]);
else if (*cur_height* − h > *active_height*[6]) b = *awful_bad*;
else b = *badness*(*cur_height* − h, *active_height*[6])

This code is used in section 973.

975. 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 975 ⟩ ≡

if (*type*(p) ≡ *kern_node*) $q = p$;
else { 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)) {
    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;
    $q = r$;
  }
}
cur_height = *cur_height* + *prev_dp* + *width*(q); *prev_dp* = 0

This code is used in section 971.

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

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

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

⟨Declare the function called *do_marks* 1507⟩

```
static pointer vsplit(halfword n, scaled h)    /* extracts a page of height h from box n */
{ pointer v;    /* the box to be split */
  pointer p;    /* runs through the vlist */
  pointer q;    /* points to where the break occurs */
  cur_val = n;
  fetch_box(v);
  flush_node_list(split_disc);
  split_disc = null;
  if (sa_mark ≠ null)
    if (do_marks(vsplit_init, 0, sa_mark)) sa_mark = null;
  if (split_first_mark ≠ null) { delete_token_ref(split_first_mark);
    split_first_mark = null;
    delete_token_ref(split_bot_mark);
    split_bot_mark = null;
  }
  ⟨Dispense with trivial cases of void or bad boxes 977⟩;
  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 978⟩;
  q = prune_page_top(q, saving_vdiscards > 0);
  p = list_ptr(v);
  free_node(v, box_node_size);
  if (q ≠ null) q = vpack(q, natural);
  change_box(q);    /* the eq_level of the box stays the same */
  return vpackage(p, h, exactly, split_max_depth);
}
```

977. ⟨Dispense with trivial cases of void or bad boxes 977⟩ ≡

```
if (v ≡ null) { return null;
}
if (type(v) ≠ vlist_node) { 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( );
  return null;
}
```

This code is used in section 976.

978. 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 978 ⟩ ≡

$p = list\_ptr(v)$;
if $(p \equiv q)$ $list\_ptr(v) = null$;
else
  loop { **if** $(type(p) \equiv mark\_node)$
      if $(mark\_class(p) \neq 0)$ ⟨ Update the current marks for *vsplit* 1509 ⟩
      else if $(split\_first\_mark \equiv null)$ { $split\_first\_mark = mark\_ptr(p)$;
        $split\_bot\_mark = split\_first\_mark$;
        $token\_ref\_count(split\_first\_mark) =$
            $token\_ref\_count(split\_first\_mark) + 2$;
      }
      else { $delete\_token\_ref(split\_bot\_mark)$;
        $split\_bot\_mark = mark\_ptr(p)$;
        $add\_token\_ref(split\_bot\_mark)$;
      }
    if $(link(p) \equiv q)$ { $link(p) = null$;
      goto *done*;
    }
    $p = link(p)$;
  }
done:

This code is used in section 976.

979. 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 ⟩ +≡
static pointer *page_tail*; /∗ the final node on the current page ∗/
static int *page_contents*; /∗ what is on the current page so far? ∗/
static scaled *page_max_depth*; /∗ maximum box depth on page being built ∗/
static pointer *best_page_break*; /∗ break here to get the best page known so far ∗/
static int *least_page_cost*; /∗ the score for this currently best page ∗/
static scaled *best_size*; /∗ its *page_goal* ∗/

980. 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 $r\_1 \equiv$ *link*(*page_ins_head*); node r_j is followed by $r\_j+1 \equiv$ *link*(r_j); and if there are n items we have $r\_n+1 \equiv$ *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(A) link(A + 1)    /* an insertion for this class will break here if anywhere */
#define broken_ins(A) info(A + 1)    /* this insertion might break at broken_ptr */
#define last_ins_ptr(A) link(A + 2)    /* the most recent insertion for this subtype */
#define best_ins_ptr(A) info(A + 2)    /* the optimum most recent insertion */
```

⟨Initialize the special list heads and constant nodes 789⟩ +≡

```
  subtype(page_ins_head) = qi(255);
  type(page_ins_head) = split_up;
  link(page_ins_head) = page_ins_head;
```

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

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

#define *page_goal* *page_so_far*[0] /∗ desired height of information on page being built ∗/
#define *page_total* *page_so_far*[1] /∗ height of the current page ∗/
#define *page_shrink* *page_so_far*[6] /∗ shrinkability of the current page ∗/
#define *page_depth* *page_so_far*[7] /∗ depth of the current page ∗/

⟨ Global variables 13 ⟩ +≡
static scaled *page_so_far*[8]; /∗ height and glue of the current page ∗/
static pointer *last_glue*; /∗ used to implement `\lastskip` ∗/
static int *last_penalty*; /∗ used to implement `\lastpenalty` ∗/
static scaled *last_kern*; /∗ used to implement `\lastkern` ∗/
static int *last_node_type*; /∗ used to implement `\lastnodetype` ∗/
static int *insert_penalties*; /∗ sum of the penalties for insertions that were held over ∗/

982. ⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
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);

983. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *set_page_dimen*:
switch (*chr_code*) {
case 0: *print_esc*("pagegoal"); **break**;
case 1: *print_esc*("pagetotal"); **break**;
case 2: *print_esc*("pagestretch"); **break**;
case 3: *print_esc*("pagefilstretch"); **break**;
case 4: *print_esc*("pagefillstretch"); **break**;
case 5: *print_esc*("pagefilllstretch"); **break**;
case 6: *print_esc*("pageshrink"); **break**;
default: *print_esc*("pagedepth");
} **break**;

984. #**define** *print_plus*(*A*, *B*)

```
        if (page_so_far[A] ≠ 0) { print("␣plus␣");
          print_scaled(page_so_far[A]);
          print(B); }
  static void print_totals(void)
  { print_scaled(page_total);
    print_plus(2, "");
    print_plus(3, "fil");
    print_plus(4, "fill");
    print_plus(5, "filll");
    if (page_shrink ≠ 0) { print("␣minus␣");
      print_scaled(page_shrink);
    }
  }
```

985. ⟨Show the status of the current page 985⟩ ≡

```
  if (page_head ≠ page_tail) { print_nl("###␣current␣page:");
    if (output_active) print("␣(held␣over␣for␣next␣output)");
    show_box(link(page_head));
    if (page_contents > empty) { print_nl("total␣height␣");
      print_totals();
      print_nl("␣goal␣height␣");
      print_scaled(page_goal);
      r = link(page_ins_head);
      while (r ≠ page_ins_head) { print_ln();
        print_esc("insert");
        t = qo(subtype(r));
        print_int(t);
        print("␣adds␣");
        if (count(t) ≡ 1000) t = height(r);
        else t = x_over_n(height(r), 1000) * count(t);
        print_scaled(t);
        if (type(r) ≡ split_up) { q = page_head;
          t = 0;
          do { q = link(q);
            if ((type(q) ≡ ins_node) ∧ (subtype(q) ≡ subtype(r))) incr(t);
          } while (¬(q ≡ broken_ins(r)));
          print(",␣#");
          print_int(t);
          print("␣might␣split");
        }
        r = link(r);
      }
    }
  }
```

This code is used in section 217.

986. 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(A) page_so_far[A] = 0
  static void freeze_page_specs(small_number s)
  { 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;
#ifdef STAT
    if (tracing_pages > 0) { begin_diagnostic();
      print_nl("%% goal height=");
      print_scaled(page_goal);
      print(", max depth=");
      print_scaled(page_max_depth);
      end_diagnostic(false);
    }
#endif
  }
```

987. 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 789⟩ +≡
  type(page_head) = glue_node;
  subtype(page_head) = normal;
```

988. The global variable *output_active* is true during the time the user's output routine is driving TEX.

```
⟨Global variables 13⟩ +≡
  static bool output_active;   /* are we in the midst of an output routine? */
```

989.
```
⟨Set initial values of key variables 21⟩ +≡
  output_active = false;
  insert_penalties = 0;
```

990. 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 990 ⟩ ≡

```
page_contents = empty;
page_tail = page_head;
link(page_head) = null;
last_glue = max_halfword;
last_penalty = 0;
last_kern = 0;
last_node_type = −1;
page_depth = 0; page_max_depth = 0
```

This code is used in sections 214 and 1016.

991. 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.

```
static void box_error(eight_bits n)
{ 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;
}
```

992. The following procedure guarantees that a given box register does not contain an `\hbox`.

```
static void ensure_vbox(eight_bits n)
{ pointer p;    /* the box register contents */
  p = box(n);
  if (p ≠ null)
    if (type(p) ≡ hlist_node) { 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);
    }
}
```

993. 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.

⟨Declare the procedure called *fire_up* 1011⟩

static void *build_page*(**void**) /* append contributions to the current page */
{ **pointer** *p*; /* the node being appended */
pointer *q*, *r*; /* nodes being examined */
int *b*, *c*; /* badness and cost of current page */
int *pi*; /* penalty to be added to the badness */
int *n*; /* insertion box number */
scaled *delta*, *h*, *w*; /* sizes used for insertion calculations */
⟨Local variables to save the profiling context 1761⟩
if ((*link*(*contrib_head*) ≡ *null*) ∨ *output_active*) **return**;
⟨Charge the time used here on *build_page* 1765⟩
do {
resume: *p* = *link*(*contrib_head*);
⟨Update the values of *last_glue*, *last_penalty*, and *last_kern* 995⟩;
⟨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 996⟩;
} **while** (¬(*link*(*contrib_head*) ≡ *null*));
⟨Make the contribution list empty by setting its tail to *contrib_head* 994⟩;
⟨restore the previous current file, line, and command 1763⟩
}

994. #**define** *contrib_tail* *nest*[0].*tail_field* /* tail of the contribution list */

⟨Make the contribution list empty by setting its tail to *contrib_head* 994⟩ ≡
if (*nest_ptr* ≡ 0) *tail* = *contrib_head*; /* vertical mode */
else *contrib_tail* = *contrib_head* /* other modes */

This code is used in section 993.

995. ⟨Update the values of *last_glue*, *last_penalty*, and *last_kern* 995⟩ ≡
if (*last_glue* ≠ *max_halfword*) *delete_glue_ref*(*last_glue*);
last_penalty = 0;
last_kern = 0;
last_node_type = *type*(*p*) + 1;
if (*type*(*p*) ≡ *glue_node*) { *last_glue* = *glue_ptr*(*p*);
add_glue_ref(*last_glue*);
}
else { *last_glue* = *max_halfword*;
if (*type*(*p*) ≡ *penalty_node*) *last_penalty* = *penalty*(*p*);
else if (*type*(*p*) ≡ *kern_node*) *last_kern* = *width*(*p*);
}

This code is used in section 993.

996. 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 996⟩ ≡
  ⟨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 999⟩;
  ⟨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* 1004⟩;
  if $((type(p) < glue\_node) \lor (type(p) > kern\_node))$ **goto** *contribute*;
update_heights:
  ⟨Update the current page measurements with respect to the glue or kern specified by node p 1003⟩;
contribute: ⟨Make sure that *page_max_depth* is not exceeded 1002⟩;
  ⟨Link node p into the current page and **goto** *done* 997⟩;
done1: ⟨Recycle node p 998⟩;
  done:

This code is used in section 993.

997. ⟨Link node p into the current page and **goto** *done* 997⟩ ≡
  $link(page\_tail) = p$;
  $page\_tail = p$;
  $link(contrib\_head) = link(p)$;
  $link(p) = null$; **goto** *done*

This code is used in section 996.

998. ⟨Recycle node p 998⟩ ≡
  $link(contrib\_head) = link(p)$;
  $link(p) = null$;
  if $(saving\_vdiscards > 0)$ { **if** $(page\_disc \equiv null)$ $page\_disc = p$; **else** $link(tail\_page\_disc) = p$;
    $tail\_page\_disc = p$;
  }
  else $flush\_node\_list(p)$

This code is used in section 996.

999. 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 999⟩ ≡

switch (*type*(*p*)) {
case *hlist_node*: **case** *vlist_node*: **case** *rule_node*:
  if (*page_contents* < *box_there*)
    ⟨Initialize the current page, insert the `\topskip` glue ahead of p, and **goto** *resume* 1000⟩
  else ⟨Prepare to move a box or rule node to the current page, then **goto** *contribute* 1001⟩ **break**;
case *whatsit_node*: ⟨Prepare to move whatsit p to the current page, then **goto** *contribute* 1363⟩;
case *glue_node*:
  if (*page_contents* < *box_there*) **goto** *done1*;
  else if (*precedes_break*(*page_tail*)) *pi* = 0;
  else goto *update_heights*; **break**;
case *kern_node*:
  if (*page_contents* < *box_there*) **goto** *done1*;
  else if (*link*(*p*) ≡ *null*) **return**;
  else if (*type*(*link*(*p*)) ≡ *glue_node*) *pi* = 0;
  else goto *update_heights*; **break**;
case *penalty_node*:
  if (*page_contents* < *box_there*) **goto** *done1*; **else** *pi* = *penalty*(*p*); **break**;
case *mark_node*: **goto** *contribute*;
case *ins_node*: ⟨Append an insertion to the current page and **goto** *contribute* 1007⟩
default: *confusion*("page");
}

This code is used in section 996.

1000. ⟨Initialize the current page, insert the `\topskip` glue ahead of p, and **goto** *resume* 1000⟩ ≡
{ **if** (*page_contents* ≡ *empty*) *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*)) *width*(*temp_ptr*) = *width*(*temp_ptr*) − *height*(*p*);
  else *width*(*temp_ptr*) = 0;
  link(*q*) = *p*;
  link(*contrib_head*) = *q*;
  goto *resume*;
}

This code is used in section 999.

1001. ⟨Prepare to move a box or rule node to the current page, then **goto** *contribute* 1001⟩ ≡
{ *page_total* = *page_total* + *page_depth* + *height*(*p*);
  page_depth = *depth*(*p*);
  goto *contribute*;
}

This code is used in section 999.

1002. ⟨Make sure that *page_max_depth* is not exceeded 1002⟩ ≡

if $(page\_depth > page\_max\_depth)$ { $page\_total =$
$page\_total + page\_depth - page\_max\_depth$;
$page\_depth = page\_max\_depth$;
}

This code is used in section 996.

1003. ⟨Update the current page measurements with respect to the glue or kern specified by node p 1003⟩ ≡

if $(type(p) \equiv kern\_node)$ $q = p$;
else { $q = glue\_ptr(p)$;
$page\_so\_far[2 + stretch\_order(q)] =$
$page\_so\_far[2 + stretch\_order(q)] + stretch(q)$;
$page\_shrink = page\_shrink + shrink(q)$;
if $((shrink\_order(q) \neq normal) \land (shrink(q) \neq 0))$ {
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 ( ) ;
$r = new\_spec(q)$;
$shrink\_order(r) = normal$;
$delete\_glue\_ref(q)$;
$glue\_ptr(p) = r$;
$q = r$;
}
}
$page\_total = page\_total + page\_depth + width(q)$; $page\_depth = 0$

This code is used in section 996.

1004. ⟨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* 1004⟩ ≡

```
  if (pi < inf_penalty) {
    ⟨Compute the badness, b, of the current page, using awful_bad if the box is too full 1006⟩;
    if (b < awful_bad)
      if (pi ≤ eject_penalty) c = pi;
      else if (b < inf_bad) c = b + pi + insert_penalties;
      else c = deplorable;
    else c = b;
    if (insert_penalties ≥ 10000) c = awful_bad;
#ifdef STAT
    if (tracing_pages > 0) ⟨Display the page break cost 1005⟩;
#endif
    if (c ≤ least_page_cost) { best_page_break = p;
      best_size = page_goal;
      least_page_cost = c;
      r = link(page_ins_head);
      while (r ≠ page_ins_head) { best_ins_ptr(r) = last_ins_ptr(r);
        r = link(r);
      }
    }
    if ((c ≡ awful_bad) ∨ (pi ≤ eject_penalty)) { fire_up(p);
        /* output the current page at the best place */
      if (output_active) { ⟨restore the previous current file, line, and command 1763⟩
        return;      /* user's output routine will act */
      }
      goto done;      /* the page has been shipped out by default output routine */
    }
  }
```

This code is used in section 996.

1005. ⟨Display the page break cost 1005⟩ ≡

```
  { begin_diagnostic();
    print_nl("%");
    print("␣t=");
    print_totals();
    print("␣g=");
    print_scaled(page_goal);
    print("␣b=");
    if (b ≡ awful_bad) print_char('*'); else print_int(b);
    print("␣p=");
    print_int(pi);
    print("␣c=");
    if (c ≡ awful_bad) print_char('*'); else print_int(c);
    if (c ≤ least_page_cost) print_char('#');
    end_diagnostic(false);
  }
```

This code is used in section 1004.

1006. ⟨Compute the badness, b, of the current page, using *awful_bad* if the box is too full 1006⟩ ≡

if $(page\_total < page\_goal)$
  if $((page\_so\_far[3] \neq 0) \lor (page\_so\_far[4] \neq 0) \lor$
      $(page\_so\_far[5] \neq 0))$ $b = 0$;
  else $b = badness(page\_goal - page\_total, page\_so\_far[2])$;
else if $(page\_total - page\_goal > page\_shrink)$ $b = awful\_bad$;
else $b = badness(page\_total - page\_goal, page\_shrink)$

This code is used in section 1004.

1007. ⟨Append an insertion to the current page and **goto** *contribute* 1007⟩ ≡

{ **if** $(page\_contents \equiv empty)$ $freeze\_page\_specs(inserts\_only)$;
  $n = subtype(p)$;
  $r = page\_ins\_head$;
  while $(n \geq subtype(link(r)))$ $r = link(r)$;
  $n = qo(n)$;
  if $(subtype(r) \neq qi(n))$ ⟨Create a page insertion node with $subtype(r) = qi(n)$, and include the glue correction for box n in the current page state 1008⟩;
  if $(type(r) \equiv split\_up)$ $insert\_penalties = insert\_penalties + float\_cost(p)$;
  else { $last\_ins\_ptr(r) = p$;
    $delta = page\_goal - page\_total - page\_depth + page\_shrink$;
     /* this much room is left if we shrink the maximum */
    if $(count(n) \equiv 1000)$ $h = height(p)$;
    else $h = x\_over\_n(height(p), 1000) * count(n)$;   /* this much room is needed */
    if $(((h \leq 0) \lor (h \leq delta)) \land (height(p) + height(r) \leq dimen(n)))$ { $page\_goal = page\_goal - h$;
     $height(r) = height(r) + height(p)$;
    }
    else ⟨Find the best way to split the insertion, and change $type(r)$ to *split_up* 1009⟩;
  }
  goto *contribute*;
}

This code is used in section 999.

1008. 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 1008⟩ ≡

```
{ 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) height(r) = 0;
  else height(r) = height(box(n)) + depth(box(n));
  best_ins_ptr(r) = null;
  q = skip(n);
  if (count(n) ≡ 1000) 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)) {
    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();
  }
}
```

This code is used in section 1007.

1009. 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* 1009 ⟩ ≡

```
  { if (count(n) ≤ 0) w = max_dimen;
    else { w = page_goal − page_total − page_depth;
      if (count(n) ≠ 1000) w = x_over_n(w, count(n)) ∗ 1000;
    }
    if (w > dimen(n) − height(r)) w = dimen(n) − height(r);
    q = vert_break(ins_ptr(p), w, depth(p));
    height(r) = height(r) + best_height_plus_depth;
#ifdef STAT
    if (tracing_pages > 0) ⟨ Display the insertion split cost 1010 ⟩;
#endif
    if (count(n) ≠ 1000) 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) insert_penalties = insert_penalties + eject_penalty;
    else if (type(q) ≡ penalty_node) insert_penalties = insert_penalties + penalty(q);
  }
```

This code is used in section 1007.

1010. ⟨ Display the insertion split cost 1010 ⟩ ≡

```
  { 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) print_int(eject_penalty);
    else if (type(q) ≡ penalty_node) print_int(penalty(q));
    else print_char('0');
    end_diagnostic(false);
  }
```

This code is used in section 1009.

1011. 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 $bot\_mark \equiv null$ implies $top\_mark \equiv first\_mark \equiv null$.

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* 1011⟩ ≡
```
static void fire_up(pointer c)
{ pointer p, q, r, s;    /* nodes being examined and/or changed */
  pointer prev_p;    /* predecessor of p */
  int n;    /* insertion box number */
  bool wait;    /* should the present insertion be held over? */
  int save_vbadness;    /* saved value of vbadness */
  scaled save_vfuzz;    /* saved value of vfuzz */
  pointer save_split_top_skip;    /* saved value of split_top_skip */
  ⟨Set the value of output_penalty 1012⟩;
  if (sa_mark ≠ null)
    if (do_marks(fire_up_init, 0, sa_mark)) sa_mark = null;
  if (bot_mark ≠ null) { if (top_mark ≠ null) delete_token_ref(top_mark);
    top_mark = bot_mark;
    add_token_ref(top_mark);
    delete_token_ref(first_mark);
    first_mark = null;
  }
  ⟨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 1013⟩;
  if (sa_mark ≠ null)
    if (do_marks(fire_up_done, 0, sa_mark)) sa_mark = null;
  if ((top_mark ≠ null) ∧ (first_mark ≡ null)) { first_mark = top_mark;
    add_token_ref(top_mark);
  }
  if (output_routine ≠ null)
    if (dead_cycles ≥ max_dead_cycles)
      ⟨Explain that too many dead cycles have occurred in a row 1023⟩
    else ⟨Fire up the user's output routine and return 1024⟩;
  ⟨Perform the default output routine 1022⟩;
}
```
This code is used in section 993.

1012. ⟨Set the value of *output_penalty* 1012⟩ ≡
```
  if (type(best_page_break) ≡ penalty_node) {
    geq_word_define(int_base + output_penalty_code, penalty(best_page_break));
    penalty(best_page_break) = inf_penalty;
  }
  else geq_word_define(int_base + output_penalty_code, inf_penalty)
```
This code is used in section 1011.

1013. 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 1013 ⟩ ≡

```
if (c ≡ best_page_break) best_page_break = null;    /* c not yet linked in */
⟨ Ensure that box 255 is empty before output 1014 ⟩;
insert_penalties = 0;    /* this will count the number of insertions held over */
save_split_top_skip = split_top_skip;
if (holding_inserts ≤ 0) ⟨ Prepare all the boxes involved in insertions to act as queues 1017 ⟩;
q = hold_head;
link(q) = null;
prev_p = page_head;
p = link(prev_p);
while (p ≠ best_page_break) { if (type(p) ≡ ins_node) { if (holding_inserts ≤ 0)
        ⟨ 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 1019 ⟩;
  }
  else if (type(p) ≡ mark_node)
    if (mark_class(p) ≠ 0) ⟨ Update the current marks for fire_up 1512 ⟩
    else ⟨ Update the values of first_mark and bot_mark 1015 ⟩;
  prev_p = p;
  p = link(prev_p);
}
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 1016 ⟩;
⟨ Delete the page-insertion nodes 1018 ⟩
```

This code is used in section 1011.

1014. ⟨ Ensure that box 255 is empty before output 1014 ⟩ ≡

```
if (box(255) ≠ null) { 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);
}
```

This code is used in section 1013.

1015. ⟨ Update the values of *first_mark* and *bot_mark* 1015 ⟩ ≡

```
{ if (first_mark ≡ null) { first_mark = mark_ptr(p);
    add_token_ref(first_mark);
  }
  if (bot_mark ≠ null) delete_token_ref(bot_mark);
  bot_mark = mark_ptr(p);
  add_token_ref(bot_mark);
}
```

This code is used in section 1013.

1016. 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 $q \equiv$ *hold_head*; and $p \equiv$ *null* if and only if *prev_p* $\equiv$ *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 1016 ⟩ ≡

```
if (p ≠ null) { if (link(contrib_head) ≡ null)
      if (nest_ptr ≡ 0) tail = page_tail;
      else contrib_tail = page_tail;
    link(page_tail) = link(contrib_head);
    link(contrib_head) = p;
    link(prev_p) = null;
  }
  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) delete_glue_ref(last_glue);
  ⟨ Start a new current page 990 ⟩;   /* this sets last_glue = max_halfword */
  if (q ≠ hold_head) { link(page_head) = link(hold_head);
    page_tail = q;
  }
```

This code is used in section 1013.

1017. 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 1017 ⟩ ≡

```
{ r = link(page_ins_head);
  while (r ≠ page_ins_head) { if (best_ins_ptr(r) ≠ null) { n = qo(subtype(r));
        ensure_vbox(n);
        if (box(n) ≡ null) box(n) = new_null_box();
        p = box(n) + list_offset;
        while (link(p) ≠ null) p = link(p);
        last_ins_ptr(r) = p;
      }
      r = link(r);
  }
}
```

This code is used in section 1013.

1018. ⟨Delete the page-insertion nodes 1018⟩ ≡
$r = link(page\_ins\_head)$;
while $(r \neq page\_ins\_head)$ { $q = link(r)$;
$free\_node(r, page\_ins\_node\_size)$;
$r = q$;
}
$link(page\_ins\_head) = page\_ins\_head$
This code is used in section 1013.

1019. We will set $best\_ins\_ptr = null$ 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 1019⟩ ≡
{ $r = link(page\_ins\_head)$;
while $(subtype(r) \neq subtype(p))$ $r = link(r)$;
if $(best\_ins\_ptr(r) \equiv null)$ $wait = true$;
else { $wait = false$;
$s = last\_ins\_ptr(r)$;
$link(s) = ins\_ptr(p)$;
if $(best\_ins\_ptr(r) \equiv p)$ ⟨Wrap up the box specified by node r, splitting node p if called for; set $wait\colon = true$ if node p holds a remainder after splitting 1020⟩
else { **while** $(link(s) \neq null)$ $s = link(s)$;
$last\_ins\_ptr(r) = s$;
}
}
⟨Either append the insertion node p after node q, and remove it from the current page, or delete $node(p)$ 1021⟩;
}
This code is used in section 1013.

1020. ⟨Wrap up the box specified by node r, splitting node p if called for; set $wait\colon = true$ if node p holds a remainder after splitting 1020⟩ ≡
{ **if** $(type(r) \equiv split\_up)$
if $((broken\_ins(r) \equiv p) \wedge (broken\_ptr(r) \neq null))$ { **while** $(link(s) \neq broken\_ptr(r))$ $s = link(s)$;
$link(s) = null$;
$split\_top\_skip = split\_top\_ptr(p)$;
$ins\_ptr(p) = prune\_page\_top(broken\_ptr(r), false)$;
if $(ins\_ptr(p) \neq null)$ { $temp\_ptr = vpack(ins\_ptr(p), natural)$;
$height(p) = height(temp\_ptr) + depth(temp\_ptr)$;
$free\_node(temp\_ptr, box\_node\_size)$;
$wait = true$;
}
}
$best\_ins\_ptr(r) = null$;
$n = qo(subtype(r))$;
$temp\_ptr = list\_ptr(box(n))$;
$free\_node(box(n), box\_node\_size)$;
$box(n) = vpack(temp\_ptr, natural)$;
}
This code is used in section 1019.

1021. ⟨Either append the insertion node p after node q, and remove it from the current page, or delete $node(p)$ 1021⟩ ≡

```
link(prev_p) = link(p);
link(p) = null;
if (wait) { link(q) = p;
  q = p;
  incr(insert_penalties);
}
else { delete_glue_ref(split_top_ptr(p));
  free_node(p, ins_node_size);
}
p = prev_p
```

This code is used in section 1019.

1022. 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 1022⟩ ≡

```
{ if (link(page_head) ≠ null) { if (link(contrib_head) ≡ null)
        if (nest_ptr ≡ 0) 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;
    }
  flush_node_list(page_disc);
  page_disc = null;
  ship_out(box(255));
  box(255) = null;
}
```

This code is used in section 1011.

1023. ⟨Explain that too many dead cycles have occurred in a row 1023⟩ ≡

```
{ 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 ( ) ;
}
```

This code is used in section 1011.

1024. ⟨ Fire up the user's output routine and **return** 1024 ⟩ ≡

```
{ 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;
}
```

This code is used in section 1011.

1025. 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 1025 ⟩ ≡

```
{ if ((loc ≠ null) ∨ ((token_type ≠ output_text) ∧ (token_type ≠ backed_up)))
    ⟨ Recover from an unbalanced output routine 1026 ⟩;
  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 1027 ⟩;
  if (tail ≠ head)    /* current list goes after heldover insertions */
  { link(page_tail) = link(head);
    page_tail = tail;
  }
  if (link(page_head) ≠ null)    /* and both go before heldover contributions */
  { if (link(contrib_head) ≡ null) contrib_tail = page_tail;
    link(page_tail) = link(contrib_head);
    link(contrib_head) = link(page_head);
    link(page_head) = null;
    page_tail = page_head;
  }
  flush_node_list(page_disc);
  page_disc = null;
  pop_nest();
  build_page();
}
```

This code is used in section 1099.

1026. ⟨ Recover from an unbalanced output routine 1026 ⟩ ≡
```
{ 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 ( ) ;
  do { get_token( );
  } while (¬(loc ≡ null));
}    /* loops forever if reading from a file, since null ≡ min_halfword ≤ 0 */
```
This code is used in section 1025.

1027. ⟨ Ensure that box 255 is empty after output 1027 ⟩ ≡
```
if (box(255) ≠ null) { 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);
}
```
This code is used in section 1025.

1028. 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.

1029. We shall concentrate first on the inner loop of *main_control*, deferring consideration of the other cases until later.

⟨ Declare action procedures for use by *main_control* 1042 ⟩
⟨ Declare the procedure called *handle_right_brace* 1067 ⟩

```
static void main_control(void)      /* governs TEX's activities */
{ int t;      /* general-purpose temporary variable */
    ⟨ Initialize profiling 1754 ⟩
    if (every_job ≠ null) begin_token_list(every_job, every_job_text);
  big_switch: ⟨ record timing information 1756 ⟩
    get_x_token();
  big_reswitch: ⟨ set current file, line, and command for the current time slot 1760 ⟩
    ⟨ Give diagnostic information, if requested 1030 ⟩;
    switch (abs(mode) + cur_cmd) {
    case hmode + letter: case hmode + other_char: case hmode + char_given: goto main_loop;
    case hmode + char_num:
      { scan_char_num();
        cur_chr = cur_val;
        goto main_loop; }
    case hmode + no_boundary:
      { get_x_token();
        if ((cur_cmd ≡ letter) ∨ (cur_cmd ≡ other_char) ∨ (cur_cmd ≡ char_given) ∨ (cur_cmd ≡
              char_num)) cancel_boundary = true;
        goto big_reswitch;
      }
    case hmode + spacer:
      if (space_factor ≡ 1000) goto append_normal_space;
      else app_space(); break;
    case hmode + ex_space: case mmode + ex_space: goto append_normal_space;
    ⟨ Cases of main_control that are not part of the inner loop 1044 ⟩
    }      /* of the big case statement */
    goto big_switch;
  main_loop: ⟨ record timing information 1756 ⟩
    ⟨ set current file, line, and command for the current time slot 1760 ⟩
    ⟨ Append character cur_chr and the following characters (if any) to the current hlist in the current
        font; goto big_reswitch when a non-character has been fetched 1033 ⟩;
  append_normal_space:
    ⟨ Append a normal inter-word space to the current list, then goto big_switch 1040 ⟩;
}
```

1030. When a new token has been fetched at *big_switch*, we have an ideal place to monitor TEX's activity.

⟨ Give diagnostic information, if requested 1030 ⟩ ≡

```
if (interrupt ≠ 0)
  if (OK_to_interrupt) { back_input();
    check_interrupt;
    goto big_switch;
  }
#ifdef DEBUG
if (panicking) check_mem(false);
#endif
if (tracing_commands > 0) show_cur_cmd_chr()
```

This code is used in section 1029.

1031. 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 $character(lig\_stack) \equiv font\_false\_bchar[cur\_font]$. Several additional global variables are needed.

⟨ Global variables 13 ⟩ +≡

```
static internal_font_number main_f;   /* the current font */
static four_quarters main_i;   /* character information bytes for cur_l */
static four_quarters main_j;   /* ligature/kern command */
static font_index main_k;   /* index into font_info */
static pointer main_p;   /* temporary register for list manipulation */
static int main_s;   /* space factor value */
static halfword bchar;   /* boundary character of current font, or non_char */
static halfword false_bchar;   /* nonexistent character matching bchar, or non_char */
static bool cancel_boundary;   /* should the left boundary be ignored? */
static bool ins_disc;   /* should we insert a discretionary node? */
```

1032. 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;
```

1033. 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) *space_factor* = 1000;
        else if (*main_s* < 1000) { **if** (*main_s* > 0) *space_factor* = *main_s*;
        }
        else if (*space_factor* < 1000) *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** *big_reswitch* when a non-character has been fetched 1033 ⟩ ≡
  adjust_space_factor;
  main_f = *cur_font*;
  bchar = *font_bchar*[*main_f*];
  false_bchar = *font_false_bchar*[*main_f*];
  if (*mode* > 0)
    if (*language* ≠ *clang*) *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*) { *cancel_boundary* = *false*;
    main_k = *non_address*;
  }
  else *main_k* = *bchar_label*[*main_f*];
  if (*main_k* ≡ *non_address*) **goto** *main_loop_move2*; /∗ no left boundary processing ∗/
  cur_r = *cur_l*;
  cur_l = *non_char*;
  goto *main_lig_loop1*; /∗ begin with cursor after left boundary ∗/
main_loop_wrapup:
  ⟨ Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1034 ⟩;
main_loop_move: ⟨ If the cursor is immediately followed by the right boundary, **goto** *big_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* 1035 ⟩;
main_loop_lookahead:
  ⟨ Look ahead for another character, or leave *lig_stack* empty if there's none there 1037 ⟩;
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* 1038 ⟩;
main_loop_move_lig:
  ⟨ Move the cursor past a pseudo-ligature, then **goto** *main_loop_lookahead* or *main_lig_loop* 1036 ⟩

This code is used in section 1029.

1034. 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(X)      /* the parameter is either rt_hit or false */
        { main_p = new_ligature(main_f, cur_l, link(cur_q));
          if (lft_hit) { subtype(main_p) = 2;
            lft_hit = false;
          }
          if (X)
            if (lig_stack ≡ null) { incr(subtype(main_p));
              rt_hit = false;
            }
          link(cur_q) = main_p;
          tail = main_p;
          ligature_present = false;
        }
#define wrapup(A)
        if (cur_l < non_char) { if (link(cur_q) > null)
            if (character(tail) ≡ qi(hyphen_char[main_f])) ins_disc = true;
          if (ligature_present) pack_lig(A);
          if (ins_disc) { ins_disc = false;
            if (mode > 0) tail_append(new_disc());
          }
        }
```

⟨Make a ligature node, if *ligature_present*; insert a null discretionary, if appropriate 1034⟩ ≡
  wrapup(*rt_hit*)

This code is used in section 1033.

1035. ⟨If the cursor is immediately followed by the right boundary, **goto** *big_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* 1035⟩ ≡

```
  if (lig_stack ≡ null) goto big_reswitch;
  cur_q = tail;
  cur_l = character(lig_stack);
main_loop_move1:
  if (¬is_char_node(lig_stack)) goto main_loop_move_lig;
main_loop_move2:
  if ((cur_chr < font_bc[main_f]) ∨ (cur_chr > font_ec[main_f])) { char_warning(main_f, cur_chr);
    free_avail(lig_stack);
    goto big_switch;
  }
  main_i = char_info(main_f, cur_l);
  if (¬char_exists(main_i)) { char_warning(main_f, cur_chr);
    free_avail(lig_stack);
    goto big_switch;
  }
  link(tail) = lig_stack; tail = lig_stack      /* main_loop_lookahead is next */
```

This code is used in section 1033.

1036. Here we are at *main_loop_move_lig*. When we begin this code we have $cur\_q \equiv tail$ and $cur\_l \equiv character(lig\_stack)$.

⟨ Move the cursor past a pseudo-ligature, then **goto** *main_loop_lookahead* or *main_lig_loop* 1036 ⟩ ≡

```
main_p = lig_ptr(lig_stack);
if (main_p > null) 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)
  if (main_p > null) goto main_loop_lookahead;
  else cur_r = bchar;
else cur_r = character(lig_stack);
goto main_lig_loop
```

This code is used in section 1033.

1037. 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 1037 ⟩ ≡

```
get_next();   /* set only cur_cmd and cur_chr, for speed */
if (cur_cmd ≡ letter) goto main_loop_lookahead1;
if (cur_cmd ≡ other_char) goto main_loop_lookahead1;
if (cur_cmd ≡ char_given) goto main_loop_lookahead1;
x_token();   /* now expand and set cur_cmd, cur_chr, cur_tok */
if (cur_cmd ≡ letter) goto main_loop_lookahead1;
if (cur_cmd ≡ other_char) goto main_loop_lookahead1;
if (cur_cmd ≡ char_given) goto main_loop_lookahead1;
if (cur_cmd ≡ char_num) { scan_char_num();
  cur_chr = cur_val;
  goto main_loop_lookahead1;
}
if (cur_cmd ≡ no_boundary) bchar = non_char;
cur_r = bchar;
lig_stack = null;
goto main_lig_loop;
main_loop_lookahead1: 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) cur_r = non_char
  /* this prevents spurious ligatures */
```

This code is used in section 1033.

1038. 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 $main\_i \equiv char\_info(main\_f, cur\_l)$.

⟨If there's a ligature/kern command relevant to *cur_l* and *cur_r*, adjust the text appropriately; exit to *main_loop_wrapup* 1038⟩ ≡
  if (*char_tag*(*main_i*) ≠ *lig_tag*) **goto** *main_loop_wrapup*;
  if (*cur_r* ≡ *non_char*) **goto** *main_loop_wrapup*;
  main_k = *lig_kern_start*(*main_f*, *main_i*);
  main_j = *font_info*[*main_k*].*qqqq*;
  if (*skip_byte*(*main_j*) ≤ *stop_flag*) **goto** *main_lig_loop2*;
  main_k = *lig_kern_restart*(*main_f*, *main_j*);
main_lig_loop1: *main_j* = *font_info*[*main_k*].*qqqq*;
main_lig_loop2:
  if (*next_char*(*main_j*) ≡ *cur_r*)
    if (*skip_byte*(*main_j*) ≤ *stop_flag*) ⟨Do ligature or kern command, returning to *main_lig_loop* or *main_loop_wrapup* or *main_loop_move* 1039⟩;
  if (*skip_byte*(*main_j*) ≡ *qi*(0)) *incr*(*main_k*);
  else { **if** (*skip_byte*(*main_j*) ≥ *stop_flag*) **goto** *main_loop_wrapup*;
    main_k = *main_k* + *qo*(*skip_byte*(*main_j*)) + 1;
  }
  goto *main_lig_loop1*

This code is used in section 1033.

1039. 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* 1039⟩ ≡

```
{ if (op_byte(main_j) ≥ kern_flag) { wrapup(rt_hit);
      tail_append(new_kern(char_kern(main_f, main_j)));
      goto main_loop_move;
  }
  if (cur_l ≡ non_char) lft_hit = true;
  else if (lig_stack ≡ null) rt_hit = true;
  check_interrupt;   /* allow a way out in case there's an infinite ligature loop */
  switch (op_byte(main_j)) {
  case qi(1): case qi(5):
    { cur_l = rem_byte(main_j);   /* =:|, =:|> */
      main_i = char_info(main_f, cur_l);
      ligature_present = true;
    } break;
  case qi(2): case qi(6):
    { cur_r = rem_byte(main_j);   /* |=:, |=:> */
      if (lig_stack ≡ null)   /* right boundary character is being consumed */
      { lig_stack = new_lig_item(cur_r);
        bchar = non_char;
      }
      else if (is_char_node(lig_stack))   /* link(lig_stack) ≡ null */
      { main_p = lig_stack;
        lig_stack = new_lig_item(cur_r);
        lig_ptr(lig_stack) = main_p;
      }
      else character(lig_stack) = cur_r;
    } break;
  case qi(3):
    { cur_r = rem_byte(main_j);   /* |=:| */
      main_p = lig_stack;
      lig_stack = new_lig_item(cur_r);
      link(lig_stack) = main_p;
    } break;
  case qi(7): case qi(11):
    { wrapup(false);   /* |=:|>, |=:|>> */
      cur_q = tail;
      cur_l = rem_byte(main_j);
      main_i = char_info(main_f, cur_l);
      ligature_present = true;
    } break;
  default:
    { cur_l = rem_byte(main_j);
      ligature_present = true;   /* =: */
      if (lig_stack ≡ null) goto main_loop_wrapup;
      else goto main_loop_move1;
    }
  }
  if (op_byte(main_j) > qi(4))
    if (op_byte(main_j) ≠ qi(7)) goto main_loop_wrapup;
```

if $(cur\_l < non\_char)$ **goto** *main_lig_loop*;
$main\_k = bchar\_label[main\_f]$;
goto *main_lig_loop1*;
}

This code is used in section 1038.

1040. 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* 1040 ⟩ ≡
if $(space\_skip \equiv zero\_glue)$ {
⟨ Find the glue specification, *main_p*, for text spaces in the current font 1041 ⟩;
$temp\_ptr = new\_glue(main\_p)$;
}
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 1029.

1041. 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 1041 ⟩ ≡
{ $main\_p = font\_glue[cur\_font]$;
if $(main\_p \equiv null)$ { $main\_p = new\_spec(zero\_glue)$;
$main\_k = param\_base[cur\_font] + space\_code$;
$width(main\_p) = font\_info[main\_k].sc$; /* 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 $space\_shrink(cur\_font)$ */
$font\_glue[cur\_font] = main\_p$;
}
}

This code is used in sections 1040 and 1042.

1042. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ ≡

```
static void app_space(void)   /* handle spaces when space_factor ≠ 1000 */
{ pointer q;   /* glue node */
  if ((space_factor ≥ 2000) ∧ (xspace_skip ≠ zero_glue)) q = new_param_glue(xspace_skip_code);
  else { if (space_skip ≠ zero_glue) main_p = space_skip;
    else ⟨ Find the glue specification, main_p, for text spaces in the current font 1041 ⟩;
    main_p = new_spec(main_p);
    ⟨ Modify the glue specification in main_p according to the space factor 1043 ⟩;
    q = new_glue(main_p);
    glue_ref_count(main_p) = null;
  }
  link(tail) = q;
  tail = q;
}
```

See also sections 1046, 1048, 1049, 1050, 1053, 1059, 1060, 1063, 1068, 1069, 1074, 1078, 1083, 1085, 1090, 1092, 1094, 1095, 1098, 1100, 1102, 1104, 1109, 1112, 1116, 1118, 1122, 1126, 1128, 1130, 1134, 1135, 1137, 1141, 1150, 1154, 1158, 1159, 1162, 1164, 1171, 1173, 1175, 1180, 1190, 1193, 1199, 1210, 1269, 1274, 1278, 1287, 1292, 1301, 1347, and 1375.

This code is used in section 1029.

1043. ⟨ Modify the glue specification in *main_p* according to the space factor 1043 ⟩ ≡

```
  if (space_factor ≥ 2000) 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 1042.

1044. Whew—that covers the main loop. We can now proceed at a leisurely pace through the other combinations of possibilities.

```
#define any_mode(A) case vmode + A: case hmode + A: case mmode + A
        /* for mode-independent commands */
```

⟨ Cases of *main_control* that are not part of the inner loop 1044 ⟩ ≡

```
any_mode(relax): case vmode + spacer: case mmode + spacer: case mmode + no_boundary: do_nothing;
any_mode(ignore_spaces):
  { ⟨ Get the next non-blank non-call token 405 ⟩;
    goto big_reswitch;
  }
case vmode + stop:
  if (its_all_over()) { ⟨ record the end of TEX 1759 ⟩
    return;   /* this is the only way out */
  } break;
⟨ Forbidden cases detected in main_control 1047 ⟩ any_mode(mac_param): report_illegal_case(); break;
⟨ Math-only cases in non-math modes, or vice versa 1045 ⟩: insert_dollar_sign(); break;
⟨ Cases of main_control that build boxes and lists 1055 ⟩
⟨ Cases of main_control that don't depend on mode 1209 ⟩
⟨ Cases of main_control that are for extensions to TEX 1346 ⟩
```

This code is used in section 1029.

1045. 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*(*A*) **case** *vmode* + *A*: **case** *hmode* + *A*

⟨ Math-only cases in non-math modes, or vice versa 1045 ⟩ ≡
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*): **case** *mmode* + *endv*: **case** *mmode* + *par_end*: **case** *mmode* + *stop*: **case** *mmode* + *vskip*: **case** *mmode* + *un_vbox*: **case** *mmode* + *valign*: **case** *mmode* + *hrule*

This code is used in section 1044.

1046. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void insert_dollar_sign(void)
{ 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();
}
```

1047. 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* 1047 ⟩ ≡
case *vmode* + *vmove*: **case** *hmode* + *hmove*: **case** *mmode* + *hmove*: *any_mode*(*last_item*):

See also sections 1097, 1110, and 1143.

This code is used in section 1044.

1048. 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* 1042 ⟩ +≡

```
static void you_cant(void)
{ print_err("You␣can't␣use␣`");
  print_cmd_chr(cur_cmd, cur_chr);
  print("'␣in␣");
  print_mode(mode);
}
```

1049. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void report_illegal_case(void)
{ 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();
}
```

1050. 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* 1042 ⟩ +≡

```
static bool privileged(void)
{ if (mode > 0) return true;
  else { report_illegal_case();
    return false;
  }
}
```

1051. 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 225 ⟩ +≡

```
primitive("end", stop, 0);
primitive("dump", stop, 1);
```

1052. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case stop:
  if (chr_code ≡ 1) print_esc("dump"); else print_esc("end"); break;
```

1053. 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* 1042 ⟩ +≡

```
static bool its_all_over(void)   /* do this when \end or \dump occurs */
{ if (privileged()) { if ((page_head ≡ page_tail) ∧ (head ≡ tail) ∧ (dead_cycles ≡ 0)) { return true;
    }
    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 */
  }
  return false;
}
```

1054. 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.

1055. 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 1055 ⟩ ≡
case *vmode* + *hrule*: **case** *hmode* + *vrule*: **case** *mmode* + *vrule*:
{ *tail_append*(*scan_rule_spec*( ));
  if (*abs*(*mode*) ≡ *vmode*) *prev_depth* = *ignore_depth*;
  else if (*abs*(*mode*) ≡ *hmode*) *space_factor* = 1000;
} **break**;

See also sections 1056, 1062, 1066, 1072, 1089, 1091, 1093, 1096, 1101, 1103, 1108, 1111, 1115, 1121, 1125, 1129, 1133, 1136, 1139, 1149, 1153, 1157, 1161, 1163, 1166, 1170, 1174, 1179, 1189, and 1192.
This code is used in section 1044.

1056. 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 1055 ⟩ +≡
case *vmode* + *vskip*: **case** *hmode* + *hskip*: **case** *mmode* + *hskip*: **case** *mmode* + *mskip*: *append_glue*( );
  break;
any_mode(*kern*): **case** *mmode* + *mkern*: *append_kern*( ); **break**;

1057. 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 225 ⟩ +≡
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*);

1058. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case hskip:
  switch (chr_code) {
  case skip_code: print_esc("hskip"); break;
  case fil_code: print_esc("hfil"); break;
  case fill_code: print_esc("hfill"); break;
  case ss_code: print_esc("hss"); break;
  default: print_esc("hfilneg");
  } break;
case vskip:
  switch (chr_code) {
  case skip_code: print_esc("vskip"); break;
  case fil_code: print_esc("vfil"); break;
  case fill_code: print_esc("vfill"); break;
  case ss_code: print_esc("vss"); break;
  default: print_esc("vfilneg");
  } break;
case mskip: print_esc("mskip"); break;
case kern: print_esc("kern"); break;
case mkern: print_esc("mkern"); break;
```

1059. 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* 1042 ⟩ +≡

```
  static void append_glue(void)
  { small_number s;     /* modifier of skip command */
    s = cur_chr;
    switch (s) {
    case fil_code: cur_val = fil_glue; break;
    case fill_code: cur_val = fill_glue; break;
    case ss_code: cur_val = ss_glue; break;
    case fil_neg_code: cur_val = fil_neg_glue; break;
    case skip_code: scan_glue(glue_val); break;
    case mskip_code: scan_glue(mu_val);
    }     /* now cur_val points to the glue specification */
    tail_append(new_glue(cur_val));
    if (s ≥ skip_code) { decr(glue_ref_count(cur_val));
      if (s > skip_code) subtype(tail) = mu_glue;
    }
  }
```

1060. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
  static void append_kern(void)
  { quarterword s;     /* subtype of the kern node */
    s = cur_chr;
    scan_dimen(s ≡ mu_glue, false, false);
    tail_append(new_kern(cur_val));
    subtype(tail) = s;
  }
```

1061. 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.

1062. 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 1055⟩ +≡
non_math(*left_brace*): *new_save_level*(*simple_group*); **break**;
any_mode(*begin_group*): *new_save_level*(*semi_simple_group*); **break**;
any_mode(*end_group*):
  if (*cur_group* ≡ *semi_simple_group*) *unsave*( );
  else *off_save*( ); **break**;

1063. 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* 1042⟩ +≡

```
static void off_save(void)
{ pointer p;     /* inserted token */
  if (cur_group ≡ bottom_level) ⟨Drop current token and complain that it was unmatched 1065⟩
  else { 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 1064⟩;
    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();
  }
}
```

1064. 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 1064 ⟩ ≡

```
switch (cur_group) {
case semi_simple_group:
  { info(p) = cs_token_flag + frozen_end_group;
    print_esc("endgroup");
  } break;
case math_shift_group:
  { info(p) = math_shift_token + '$';
    print_char('$');
  } break;
case math_left_group:
  { info(p) = cs_token_flag + frozen_right;
    link(p) = get_avail();
    p = link(p);
    info(p) = other_token + '.';
    print_esc("right.");
  } break;
default:
  { info(p) = right_brace_token + '}';
    print_char('}');
  }
}
```

This code is used in section 1063.

1065. ⟨ Drop current token and complain that it was unmatched 1065 ⟩ ≡

```
{ print_err("Extra␣");
  print_cmd_chr(cur_cmd, cur_chr);
  help1("Things␣are␣pretty␣mixed␣up,␣but␣I␣think␣the␣worst␣is␣over.");
  error();
}
```

This code is used in section 1063.

1066. 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 1055 ⟩ +≡
any_mode(*right_brace*): *handle_right_brace*( ); **break**;

1067. ⟨ Declare the procedure called *handle_right_brace* 1067 ⟩ ≡

```
static void handle_right_brace(void)
{ pointer p, q;   /* for short-term use */
  scaled d;   /* holds split_max_depth in insert_group */
  int f;   /* holds floating_penalty in insert_group */
  switch (cur_group) {
  case simple_group: unsave(); break;
  case bottom_level:
    { print_err("Too␣many␣}'s");
      help2("You've␣closed␣more␣groups␣than␣you␣opened.",
      "Such␣booboos␣are␣generally␣harmless,␣so␣keep␣going.");
      error();
    } break;
  case semi_simple_group: case math_shift_group: case math_left_group: extra_right_brace(); break;
  ⟨ Cases of handle_right_brace where a right_brace triggers a delayed action 1084 ⟩
  default: confusion("rightbrace");
  }
}
```

This code is used in section 1029.

1068. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void extra_right_brace(void)
{ print_err("Extra␣},␣or␣forgotten␣");
  switch (cur_group) {
  case semi_simple_group: print_esc("endgroup"); break;
  case math_shift_group: print_char('$'); break;
  case math_left_group: print_esc("right");
  }
  help5("I've␣deleted␣a␣group-closing␣symbol␣because␣it␣seems␣to␣be",
  "spurious,␣as␣in␣`$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);
}
```

1069. 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* 1042 ⟩ +≡

```
static void normal_paragraph(void)
{ if (looseness ≠ 0) eq_word_define(int_base + looseness_code, 0);
  if (hang_indent ≠ 0) eq_word_define(dimen_base + hang_indent_code, 0);
  if (hang_after ≠ 1) eq_word_define(int_base + hang_after_code, 1);
  if (par_shape_ptr ≠ null) eq_define(par_shape_loc, shape_ref, null);
  if (inter_line_penalties_ptr ≠ null) eq_define(inter_line_penalties_loc, shape_ref, null);
}
```

1070. 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* $\equiv 2^{30}$. Codes *box_flag* through *global_box_flag* $-$ 1 represent '`\setbox0`' through '`\setbox32767`'; codes *global_box_flag* through *ship_out_flag* $-$ 1 represent '`\global\setbox0`' through '`\global\setbox32767`'; code *ship_out_flag* represents '`\shipout`'; and codes *leader_flag* through *leader_flag* + 2 represent '`\leaders`', '`\cleaders`', and '`\xleaders`'.

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

#define *box_flag* °*10000000000* /* context code for '`\setbox0`' */
#define *global_box_flag* °*10000100000* /* context code for '`\global\setbox0`' */
#define *ship_out_flag* °*10000200000* /* context code for '`\shipout`' */
#define *leader_flag* °*10000200001* /* context code for '`\leaders`' */
#define *box_code* 0 /* *chr_code* for '`\box`' */
#define *copy_code* 1 /* *chr_code* for '`\copy`' */
#define *last_box_code* 2 /* *chr_code* for '`\lastbox`' */
#define *vsplit_code* 3 /* *chr_code* for '`\vsplit`' */
#define *vtop_code* 4 /* *chr_code* for '`\vtop`' */

⟨Put each of TEX's primitives into the hash table 225⟩ +≡
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* $\equiv$ *leader_flag* $-$ 1 */
primitive("`leaders`", *leader_ship*, *a_leaders*);
primitive("`cleaders`", *leader_ship*, *c_leaders*);
primitive("`xleaders`", *leader_ship*, *x_leaders*);

1071. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *hmove*:
  if (*chr_code* ≡ 1) *print_esc*("moveleft"); **else** *print_esc*("moveright"); **break**;
case *vmove*:
  if (*chr_code* ≡ 1) *print_esc*("raise"); **else** *print_esc*("lower"); **break**;
case *make_box*:
  switch (*chr_code*) {
  case *box_code*: *print_esc*("box"); **break**;
  case *copy_code*: *print_esc*("copy"); **break**;
  case *last_box_code*: *print_esc*("lastbox"); **break**;
  case *vsplit_code*: *print_esc*("vsplit"); **break**;
  case *vtop_code*: *print_esc*("vtop"); **break**;
  case *vtop_code* + *vmode*: *print_esc*("vbox"); **break**;
  default: *print_esc*("hbox");
  } **break**;
case *leader_ship*:
  if (*chr_code* ≡ *a_leaders*) *print_esc*("leaders");
  else if (*chr_code* ≡ *c_leaders*) *print_esc*("cleaders");
  else if (*chr_code* ≡ *x_leaders*) *print_esc*("xleaders");
  else *print_esc*("shipout"); **break**;

1072. 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 1055 ⟩ +≡
case *vmode* + *hmove*: **case** *hmode* + *vmove*: **case** *mmode* + *vmove*:
  { *t* = *cur_chr*;
    scan_normal_dimen;
    if (*t* ≡ 0) *scan_box*(*cur_val*); **else** *scan_box*(−*cur_val*);
  } **break**;
any_mode(*leader_ship*): *scan_box*(*leader_flag* − *a_leaders* + *cur_chr*); **break**;
any_mode(*make_box*): *begin_box*(0); **break**;

1073. 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 ⟩ +≡
  static pointer *cur_box*;   /∗ box to be placed into its context ∗/

1074. 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* 1042 ⟩ +≡
  static void *box_end*(**int** *box_context*)
  { **pointer** *p*;   /∗ *ord_noad* for new box in math mode ∗/
    small_number *a*;   /∗ global prefix ∗/
    if (*box_context* < *box_flag*) ⟨ Append box *cur_box* to the current list, shifted by *box_context* 1075 ⟩
    else if (*box_context* < *ship_out_flag*) ⟨ Store *cur_box* in a box register 1076 ⟩
    else if (*cur_box* ≠ *null*)
      if (*box_context* > *ship_out_flag*) ⟨ Append a new leader node that uses *cur_box* 1077 ⟩
      else *ship_out*(*cur_box*);
  }

1075. 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.

$\langle$ Append box *cur_box* to the current list, shifted by *box_context* 1075 $\rangle \equiv$

{ **if** $(cur\_box \neq null)$ { $shift\_amount(cur\_box) = box\_context$;
  if $(abs(mode) \equiv vmode)$ { $append\_to\_vlist(cur\_box)$;
   if $(adjust\_tail \neq null)$ { **if** $(adjust\_head \neq adjust\_tail)$ { $link(tail) = link(adjust\_head)$;
     $tail = adjust\_tail$;
    }
    $adjust\_tail = null$;
   }
   if $(mode > 0)$ $build\_page(\,)$;
  }
  else { **if** $(abs(mode) \equiv hmode)$ $space\_factor = 1000$;
   else { $p = new\_noad(\,)$;
    $math\_type(nucleus(p)) = sub\_box$;
    $info(nucleus(p)) = cur\_box$;
    $cur\_box = p$;
   }
   $link(tail) = cur\_box$;
   $tail = cur\_box$;
  }
 }
}

This code is used in section 1074.

1076. $\langle$ Store *cur_box* in a box register 1076 $\rangle \equiv$

{ **if** $(box\_context < global\_box\_flag)$ { $cur\_val = box\_context - box\_flag$;
  $a = 0$;
 }
 else { $cur\_val = box\_context - global\_box\_flag$;
  $a = 4$;
 }
 if $(cur\_val < 256)$ **define** $(box\_base + cur\_val, box\_ref, cur\_box)$;
 else sa_def_box;
}

This code is used in section 1074.

1077. ⟨Append a new leader node that uses *cur_box* 1077⟩ ≡

```
{ ⟨Get the next non-blank non-relax non-call token 403⟩;
  if (((cur_cmd ≡ hskip) ∧ (abs(mode) ≠ vmode)) ∨
        ((cur_cmd ≡ vskip) ∧ (abs(mode) ≡ vmode))) { append_glue();
    subtype(tail) = box_context − (leader_flag − a_leaders);
    leader_ptr(tail) = cur_box;
  }
  else { 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);
  }
}
```

This code is used in section 1074.

1078. 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* 1042⟩ +≡

```
static void begin_box(int box_context)
{ pointer p, q;    /* run through the current list */
  int m;    /* the length of a replacement list */
  halfword k;    /* 0 or vmode or hmode */
  halfword n;    /* a box number */
  switch (cur_chr) {
  case box_code:
    { scan_register_num();
      fetch_box(cur_box);
      change_box(null);    /* the box becomes void, at the same level */
    } break;
  case copy_code:
    { scan_register_num();
      fetch_box(q);
      cur_box = copy_node_list(q);
    } break;
  case 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 1079⟩ break;
  case vsplit_code: ⟨Split off part of a vertical box, make cur_box point to it 1081⟩ break;
  default: ⟨Initiate the construction of an hbox or vbox, then return 1082⟩
  }
  box_end(box_context);    /* in simple cases, we use the box immediately */
}
```

1079. Note that the condition $\neg is\_char\_node(tail)$ implies that $head \neq tail$, 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 $cur\_box :=null$ 1079 ⟩ ≡

```
{ cur_box = null;
  if (abs(mode) ≡ mmode) { you_cant();
    help1("Sorry;␣this␣\\lastbox␣will␣be␣void.");
    error();
  }
  else if ((mode ≡ vmode) ∧ (head ≡ tail)) { you_cant();
    help2("Sorry...I␣usually␣can't␣take␣things␣from␣the␣current␣page.",
    "This␣\\lastbox␣will␣therefore␣be␣void.");
    error();
  }
  else { if (¬is_char_node(tail))
      if ((type(tail) ≡ hlist_node) ∨ (type(tail) ≡ vlist_node))
        ⟨ Remove the last box, unless it's part of a discretionary 1080 ⟩;
  }
}
```

This code is used in section 1078.

1080. ⟨ Remove the last box, unless it's part of a discretionary 1080 ⟩ ≡

```
{ q = head;
  do { p = q;
    if (¬is_char_node(q))
      if (type(q) ≡ disc_node) { for (m = 1; m ≤ replace_count(q); m++) p = link(p);
        if (p ≡ tail) goto done;
      }
    q = link(p);
  } while (¬(q ≡ tail));
  cur_box = tail;
  shift_amount(cur_box) = 0;
  tail = p;
  link(p) = null;
done: ;
}
```

This code is used in section 1079.

1081. Here we deal with things like '`\vsplit 13 to 100pt`'.

⟨ Split off part of a vertical box, make cur_box point to it 1081 ⟩ ≡

```
{ scan_register_num();
  n = cur_val;
  if (¬scan_keyword("to")) { print_err("Missing␣'to'␣inserted");
    help2("I'm␣working␣on␣'\\vsplit<box␣number>␣to␣<dimen>';",
    "will␣look␣for␣the␣<dimen>␣next.");
    error();
  }
  scan_normal_dimen;
  cur_box = vsplit(n, cur_val);
}
```

This code is used in section 1078.

1082. 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** 1082⟩ ≡

```
{ k = cur_chr − vtop_code;
  saved(0) = box_context;
  if (k ≡ hmode)
    if ((box_context < box_flag) ∧ (abs(mode) ≡ vmode)) scan_spec(adjusted_hbox_group, true);
    else scan_spec(hbox_group, true);
  else { if (k ≡ vmode) scan_spec(vbox_group, true);
    else { scan_spec(vtop_group, true);
      k = vmode;
    }
    normal_paragraph();
  }
  push_nest();
  mode = −k;
  if (k ≡ vmode) { prev_depth = ignore_depth;
    if (every_vbox ≠ null) begin_token_list(every_vbox, every_vbox_text);
  }
  else { space_factor = 1000;
    if (every_hbox ≠ null) begin_token_list(every_hbox, every_hbox_text);
  }
  return;
}
```

This code is used in section 1078.

1083. ⟨Declare action procedures for use by *main_control* 1042⟩ +≡

```
static void scan_box(int box_context)   /* the next input should specify a box or perhaps a rule */
{ ⟨Get the next non-blank non-relax non-call token 403⟩;
  if (cur_cmd ≡ make_box) begin_box(box_context);
  else if ((box_context ≥ leader_flag) ∧ ((cur_cmd ≡ hrule) ∨ (cur_cmd ≡ vrule))) {
    cur_box = scan_rule_spec();
    box_end(box_context);
  }
  else {
    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();
  }
}
```

1084. 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 1084⟩ ≡

```
case hbox_group: package(0); break;
case adjusted_hbox_group:
  { adjust_tail = adjust_head;
    package(0);
  } break;
case vbox_group:
  { end_graf();
    package(0);
  } break;
case vtop_group:
  { end_graf();
    package(vtop_code);
  } break;
```

See also sections 1099, 1117, 1131, 1132, 1167, 1172, and 1185.

This code is used in section 1067.

1085. ⟨Declare action procedures for use by *main_control* 1042⟩ +≡

```
  static void package(small_number c)
  { scaled h;    /* height of box */
    pointer p;    /* first node in a box */
    scaled d;    /* max depth */
    d = box_max_depth;
    unsave();
    save_ptr = save_ptr − 3;
    if (mode ≡ −hmode) cur_box = hpack(link(head), saved(2), saved(1));
    else { cur_box = vpackage(link(head), saved(2), saved(1), d);
      if (c ≡ vtop_code) ⟨Readjust the height and depth of cur_box, for \vtop 1086⟩;
    }
    pop_nest();
    box_end(saved(0));
  }
```

1086. 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` 1086⟩ ≡

```
  { h = 0;
    p = list_ptr(cur_box);
    if (p ≠ null)
      if (type(p) ≤ rule_node) h = height(p);
    depth(cur_box) = depth(cur_box) − h + height(cur_box);
    height(cur_box) = h;
  }
```

This code is used in section 1085.

1087. 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 225 ⟩ +≡
  primitive("indent", *start_par*, 1);
  primitive("noindent", *start_par*, 0);

1088. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *start_par*:
  if (*chr_code* ≡ 0) *print_esc*("noindent"); **else** *print_esc*("indent"); **break**;

1089. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡
case *vmode* + *start_par*: *new_graf*(*cur_chr* > 0); **break**;
case *vmode* + *letter*: **case** *vmode* + *other_char*: **case** *vmode* + *char_num*: **case** *vmode* + *char_given*:
    case *vmode* + *math_shift*: **case** *vmode* + *un_hbox*: **case** *vmode* + *vrule*: **case** *vmode* + *accent*:
    case *vmode* + *discretionary*: **case** *vmode* + *hskip*: **case** *vmode* + *valign*: **case** *vmode* + *ex_space*:
    case *vmode* + *no_boundary*:
  { *back_input*( );
    new_graf(*true*);
  } **break**;

1090. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡
  static small_number *norm_min*(**int** *h*)
  { **if** (*h* ≤ 0) **return** 1; **else if** (*h* ≥ 63) **return** 63; **else return** *h*;
  }
  static void *new_graf*(**bool** *indented*)
  { *prev_graf* = 0;
    if ((*mode* ≡ *vmode*) ∨ (*head* ≠ *tail*)) *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*) { *tail* = *new_null_box*( );
      link(*head*) = *tail*;
      width(*tail*) = *par_indent*; }
    if (*every_par* ≠ *null*) *begin_token_list*(*every_par*, *every_par_text*);
    if (*nest_ptr* ≡ 1) *build_page*( );   /∗ put *par_skip* glue on current page ∗/
  }

1091. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡
case *hmode* + *start_par*: **case** *mmode* + *start_par*: *indent_in_hmode*( ); **break**;

1092. ⟨Declare action procedures for use by *main_control* 1042⟩ +≡

```
static void indent_in_hmode(void)
{ pointer p, q;
  if (cur_chr > 0)   /* \indent */
  { p = new_null_box();
    width(p) = par_indent;
    if (abs(mode) ≡ hmode) space_factor = 1000;
    else { q = new_noad();
      math_type(nucleus(q)) = sub_box;
      info(nucleus(q)) = p;
      p = q;
    }
    tail_append(p);
  }
}
```

1093. 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 1055⟩ +≡

```
case vmode + par_end:
  { normal_paragraph();
    if (mode > 0) build_page();
  } break;
case hmode + par_end:
  { if (align_state < 0) 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) build_page();
  } break;
case hmode + stop: case hmode + vskip: case hmode + hrule: case hmode + un_vbox:
  case hmode + halign: head_for_vmode(); break;
```

1094. ⟨Declare action procedures for use by *main_control* 1042⟩ +≡

```
static void head_for_vmode(void)
{ if (mode < 0)
    if (cur_cmd ≠ hrule) off_save();
    else { 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 ( ) ;
    }
  else { back_input();
    cur_tok = par_token;
    back_input();
    token_type = inserted;
  }
}
```

1095. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void end_graf(void)
{ if (mode ≡ hmode) { if (head ≡ tail) pop_nest();      /* null paragraphs are ignored */
    else line_break(widow_penalty);
    normal_paragraph();
    error_count = 0;
  }
}
```

1096. Insertion and adjustment and mark nodes are constructed by the following pieces of the program.

⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
any_mode(insert): case hmode + vadjust: case mmode + vadjust: begin_insert_or_adjust(); break;
any_mode(mark): make_mark(); break;
```

1097. ⟨ Forbidden cases detected in *main_control* 1047 ⟩ +≡

```
case vmode + vadjust:
```

1098. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void begin_insert_or_adjust(void)
{ if (cur_cmd ≡ vadjust) cur_val = 255;
  else { scan_eight_bit_int();
    if (cur_val ≡ 255) { 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;
    }
  }
  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;
}
```

1099. ⟨Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1084⟩ +≡

```
case insert_group:
  { 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) { 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;
    }
    else { 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);
    }
    free_node(p, box_node_size);
    if (nest_ptr ≡ 0) build_page();
  } break;
case output_group: ⟨Resume the page builder after an output routine has come to an end 1025⟩ break;
```

1100. ⟨Declare action procedures for use by *main_control* 1042⟩ +≡

```
  static void make_mark(void)
  { pointer p;      /* new node */
    halfword c;      /* the mark class */
    if (cur_chr ≡ 0) c = 0;
    else { scan_register_num();
      c = cur_val;
    }
    p = scan_toks(false, true);
    p = get_node(small_node_size);
    mark_class(p) = c;
    type(p) = mark_node;
    subtype(p) = 0;      /* the subtype is not used */
    mark_ptr(p) = def_ref;
    link(tail) = p;
    tail = p;
  }
```

1101. Penalty nodes get into a list via the *break_penalty* command.

⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡
any_mode(*break_penalty*): *append_penalty*( ); **break**;

1102. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void append_penalty(void)
{ scan_int();
  tail_append(new_penalty(cur_val));
  if (mode ≡ vmode) build_page();
}
```

1103. 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 1055 ⟩ +≡
any_mode(*remove_item*): *delete_last*( ); **break**;

1104. 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* 1042 ⟩ +≡

```
static void delete_last(void)
{ pointer p, q;    /* run through the current list */
  int m;    /* the length of a replacement list */
  if ((mode ≡ vmode) ∧ (tail ≡ head))
    ⟨ Apologize for inability to do the operation now, unless \unskip follows non-glue 1105 ⟩
  else { if (¬is_char_node(tail))
      if (type(tail) ≡ cur_chr) { q = head;
        do { p = q;
          if (¬is_char_node(q))
            if (type(q) ≡ disc_node) { for (m = 1; m ≤ replace_count(q); m++) p = link(p);
              if (p ≡ tail) return;
            }
          q = link(p);
        } while (¬(q ≡ tail));
        link(p) = null;
        flush_node_list(tail);
        tail = p;
      }
  }
}
```

1105. ⟨ Apologize for inability to do the operation now, unless `\unskip` follows non-glue 1105 ⟩ ≡

```
{ if ((cur_chr ≠ glue_node) ∨ (last_glue ≠ max_halfword)) { you_cant();
    help2("Sorry...I␣usually␣can't␣take␣things␣from␣the␣current␣page.",
    "Try␣`I\\vskip-\\lastskip'␣instead.");
    if (cur_chr ≡ kern_node) help_line[0] = ("Try␣`I\\kern-\\lastkern'␣instead.");
    else if (cur_chr ≠ glue_node) help_line[0] =
          ("Perhaps␣you␣can␣make␣the␣output␣routine␣do␣it.");
    error ( ) ;
  }
}
```

This code is used in section 1104.

1106. ⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡

```
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);
```

1107. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case remove_item:
  if (chr_code ≡ glue_node) print_esc("unskip");
  else if (chr_code ≡ kern_node) print_esc("unkern");
  else print_esc("unpenalty"); break;
case un_hbox:
  if (chr_code ≡ copy_code) print_esc("unhcopy");
  else print_esc("unhbox"); break; case un_vbox: if (chr_code ≡ copy_code) print_esc("unvcopy")
        ⟨ Cases of un_vbox for print_cmd_chr 1532 ⟩;
  else print_esc("unvbox"); break;
```

1108. The *un_hbox* and *un_vbox* commands unwrap one of the 256 current boxes.

⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
case vmode + un_vbox: case hmode + un_hbox: case mmode + un_hbox: unpackage(); break;
```

1109. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void unpackage(void)
{ pointer p;   /* the box */
  int c;   /* should we copy? */
  if (cur_chr > copy_code) ⟨ Handle saved items and goto done 1533 ⟩;
  c = cur_chr;
  scan_register_num();
  fetch_box(p);
  if (p ≡ null) return;
  if ((abs(mode) ≡ mmode) ∨ ((abs(mode) ≡ vmode) ∧ (type(p) ≠ vlist_node)) ∨
        ((abs(mode) ≡ hmode) ∧ (type(p) ≠ hlist_node))) {
    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;
  }
  if (c ≡ copy_code) link(tail) = copy_node_list(list_ptr(p));
  else { link(tail) = list_ptr(p);
    change_box(null);
    free_node(p, box_node_size);
  }
done:
  while (link(tail) ≠ null) tail = link(tail);
}
```

1110. ⟨ Forbidden cases detected in *main_control* 1047 ⟩ +≡

```
case vmode + ital_corr:
```

1111. 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 1055 ⟩ +≡

```
case hmode + ital_corr: append_italic_correction( ); break;
case mmode + ital_corr: tail_append(new_kern(0)) break;
```

1112. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void append_italic_correction(void){ pointer p;   /* char_node at the tail of the current list */
    internal_font_number f;   /* the font in the char_node */
    if (tail ≠ head) { if (is_char_node(tail)) p = tail;
    else if (type(tail) ≡ ligature_node) p = lig_char(tail);
    else return;
    f = font(p);
    tail_append(new_kern(char_italic(f, char_info(f, character(p))))); subtype(tail) = explicit; } }
```

1113. 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 225 ⟩ +≡

```
primitive("-", discretionary, 1);
primitive("discretionary", discretionary, 0);
```

1114. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *discretionary*:
  if $(chr\_code \equiv 1)$ *print_esc*("-"); **else** *print_esc*("discretionary"); **break**;

1115. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡
case *hmode* + *discretionary*: **case** *mmode* + *discretionary*: *append_discretionary*( ); **break**;

1116. 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* 1042 ⟩ +≡
  static void *append_discretionary*(**void**)
  { **int** *c*; /* hyphen character */
    tail_append(*new_disc*( ));
    if $(cur\_chr \equiv 1)$ { $c = hyphen\_char[cur\_font]$;
      if $(c \geq 0)$
        if $(c < 256)$ $pre\_break(tail) = new\_character(cur\_font, c)$;
    }
    else { *incr*(*save_ptr*);
      $saved(-1) = 0$;
      new_save_level(*disc_group*);
      scan_left_brace( );
      push_nest( );
      $mode = -hmode$;
      $space\_factor = 1000$;
    }
  }

1117. 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 1084 ⟩ +≡
case *disc_group*: *build_discretionary*( ); **break**;

1118. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void build_discretionary(void)
{ pointer p, q;   /* for link manipulation */
  int n;   /* length of discretionary list */
  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 1120 ⟩;
  p = link(head);
  pop_nest();
  switch (saved(−1)) {
  case 0: pre_break(tail) = p; break;
  case 1: post_break(tail) = p; break;
  case 2: ⟨ Attach list p to the current list, and record its length; then finish up and return 1119 ⟩;
  }   /* there are no other cases */
  incr(saved(−1));
  new_save_level(disc_group);
  scan_left_brace();
  push_nest();
  mode = −hmode;
  space_factor = 1000;
}
```

1119. ⟨ Attach list *p* to the current list, and record its length; then finish up and **return** 1119 ⟩ ≡

```
{ if ((n > 0) ∧ (abs(mode) ≡ mmode)) { 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 ( ) ;
  }
  else link(tail) = p;
  if (n ≤ max_quarterword) replace_count(tail) = n;
  else { 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 ( ) ;
  }
  if (n > 0) tail = q;
  decr(save_ptr);
  return;
}
```

This code is used in section 1118.

1120. During this loop, $p \equiv link(q)$ 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 1120 ⟩ ≡

```
q = head;
p = link(q);
n = 0;
while (p ≠ null) { if (¬is_char_node(p))
    if (type(p) > rule_node)
      if (type(p) ≠ kern_node)
        if (type(p) ≠ ligature_node) { 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;
        }
  q = p;
  p = link(q);
  incr(n);
}
done:
```

This code is used in section 1118.

1121. 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 1055 ⟩ +≡
case *hmode* + *accent*: *make_accent*( ); **break**;

1122. 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 1042 ⟩ +≡
  static void make_accent(void)
  { double s, t;    /* amount of slant */
    pointer p, q, r;    /* character, box, and kern nodes */
    internal_font_number f;    /* relevant font */
    scaled a, h, x, w, delta;    /* heights and widths, as explained above */
    four_quarters i;    /* character information */

    scan_char_num();
    f = cur_font;
    p = new_character(f, cur_val);
    if (p ≠ null) { 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 1123 ⟩;
      if (q ≠ null) ⟨ Append the accent with appropriate kerns, then set p: = q 1124 ⟩;
      link(tail) = p;
      tail = p;
      space_factor = 1000;
    }
  }
```

1123. ⟨ Create a character node q for the next character, but set q: $=$ *null* if problems arise 1123 ⟩ ≡

```
  q = null;
  f = cur_font;
  if ((cur_cmd ≡ letter) ∨ (cur_cmd ≡ other_char) ∨ (cur_cmd ≡ char_given))
    q = new_character(f, cur_chr);
  else if (cur_cmd ≡ char_num) { scan_char_num();
    q = new_character(f, cur_val);
  }
  else back_input()
```

This code is used in section 1122.

1124. 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) **double** 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 $p{:}=q$ 1124⟩ ≡

{ $t = slant(f)/float\_constant(65536)$;
  $i = char\_info(f, character(q))$;
  $w = char\_width(f, i)$;
  $h = char\_height(f, height\_depth(i))$;
  if $(h \neq x)$ /* the accent must be shifted up or down */
  { $p = hpack(p, natural)$;
    $shift\_amount(p) = x - h$;
  }
  $delta = round((w - a)/float\_constant(2) + h * t - x * s)$;
  $r = new\_kern(delta)$;
  $subtype(r) = acc\_kern$;
  $link(tail) = r$;
  $link(r) = p$;
  $tail = new\_kern(-a - delta)$;
  $subtype(tail) = acc\_kern$;
  $link(p) = tail$;
  $p = q$;
}

This code is used in section 1122.

1125. 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 $align\_state \equiv 0$. The following program attempts to make an appropriate recovery.

⟨Cases of *main_control* that build boxes and lists 1055⟩ +≡

any_mode(*car_ret*): *any_mode*(*tab_mark*): *align_error*( ); **break**;
any_mode(*no_align*): *no_align_error*( ); **break**;
any_mode(*omit*): *omit_error*( ); **break**;

1126. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void align_error(void)
{ if (abs(align_state) > 2) ⟨ Express consternation over the fact that no alignment is in progress 1127 ⟩
  else { back_input();
    if (align_state < 0) { print_err("Missing␣{␣inserted");
      incr(align_state);
      cur_tok = left_brace_token + '{';
    }
    else { print_err("Missing␣}␣inserted");
      decr(align_state);
      cur_tok = right_brace_token + '}';
    }
    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();
  }
}
```

1127. ⟨ Express consternation over the fact that no alignment is in progress 1127 ⟩ ≡

```
{ print_err("Misplaced␣");
  print_cmd_chr(cur_cmd, cur_chr);
  if (cur_tok ≡ tab_token + '&') {
    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.");
  }
  else { 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.");
  }
  error ( ) ;
}
```

This code is used in section 1126.

1128. 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* 1042 ⟩ +≡

```
static void no_align_error(void)
{ 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 ( ) ;
}
static void omit_error(void)
{ 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 ( ) ;
}
```

1129. 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 1055 ⟩ +≡

```
case vmode + halign: case hmode + valign: init_align( ); break;
case mmode + halign:
  if (privileged( ))
    if (cur_group ≡ math_shift_group) init_align( );
    else off_save( ); break;
case vmode + endv: case hmode + endv: do_endv( ); break;
```

1130. 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* 1042 ⟩ +≡

```
static void do_endv(void)
{ 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)) 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))
    fatal_error("(interwoven␣alignment␣preambles␣are␣not␣allowed)");
  if (cur_group ≡ align_group) { end_graf( );
    if (fin_col( )) fin_row( );
  }
  else off_save( );
}
```

1131. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1084 ⟩ +≡

```
case align_group:
  { 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();
  } break;
```

1132. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1084 ⟩ +≡

```
case no_align_group:
  { end_graf();
    unsave();
    align_peek();
  } break;
```

1133. Finally, `\endcsname` is not supposed to get through to *main_control*.

⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
any_mode(end_cs_name): cs_error(); break;
```

1134. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
  static void cs_error(void)
  { print_err("Extra␣");
    print_esc("endcsname");
    help1("I'm␣ignoring␣this,␣since␣I␣wasn't␣doing␣a␣\\csname.");
    error ( ) ;
  }
```

1135. 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* 1042⟩ +≡

```
static void push_math(group_code c)
{ push_nest();
  mode = −mmode;
  incompleat_noad = null;
  new_save_level(c);
}
```

1136. 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 1055⟩ +≡

```
case hmode + math_shift: init_math(); break;
```

1137. ⟨Declare action procedures for use by *main_control* 1042⟩ +≡

```
static void init_math(void)
{ scaled w;  /∗ new or partial pre_display_size ∗/
  scaled l;  /∗ new display_width ∗/
  scaled s;  /∗ new display_indent ∗/
  pointer p;  /∗ current node when calculating pre_display_size ∗/
  pointer q;  /∗ glue specification when calculating pre_display_size ∗/
  internal_font_number f;  /∗ font in current char_node ∗/
  int n;  /∗ scope of paragraph shape specification ∗/
  scaled v;  /∗ w plus possible glue amount ∗/
  scaled d;  /∗ increment to v ∗/

  get_token();  /∗ get_x_token would fail on \ifmmode! ∗/
  if ((cur_cmd ≡ math_shift) ∧ (mode > 0)) ⟨Go into display math mode 1144⟩
  else { back_input();
    ⟨Go into ordinary math mode 1138⟩;
  }
}
```

1138. ⟨Go into ordinary math mode 1138⟩ ≡

```
{ push_math(math_shift_group);
  eq_word_define(int_base + cur_fam_code, −1);
  if (every_math ≠ null) begin_token_list(every_math, every_math_text);
}
```

This code is used in sections 1137 and 1141.

1139. 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 1055 ⟩ +≡
case *mmode* + *eq_no*:
  if (*privileged*( ))
    if (*cur_group* ≡ *math_shift_group*) *start_eq_no*( );
    else *off_save*( ); **break**;

1140. ⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
  primitive("eqno", *eq_no*, 0);
  primitive("leqno", *eq_no*, 1);

1141. 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* 1042 ⟩ +≡
  static void *start_eq_no*(**void**)
  { *saved*(0) = *cur_chr*;
    incr(*save_ptr*);
    ⟨ Go into ordinary math mode 1138 ⟩;
  }

1142. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *eq_no*:
  if (*chr_code* ≡ 1) *print_esc*("leqno"); **else** *print_esc*("eqno"); **break**;

1143. ⟨ Forbidden cases detected in *main_control* 1047 ⟩ +≡
  non_math(*eq_no*):

1144. 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 1144 ⟩ ≡
  { **if** (*head* ≡ *tail*)   /∗ '\noindent$$' or '$$ $$' ∗/
    { *pop_nest*( );
      $w = -max\_dimen$;
    }
    else { *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 1145 ⟩;
    }   /∗ 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 1148 ⟩;
    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*) *begin_token_list*(*every_display*, *every_display_text*);
    if (*nest_ptr* ≡ 1) *build_page*( );
  }

This code is used in section 1137.

1145. ⟨ 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 1145 ⟩ ≡

$v = shift\_amount(just\_box) + 2 * quad(cur\_font)$;
$w = -max\_dimen$;
$p = list\_ptr(just\_box)$;
while $(p \neq null)$ { ⟨ 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 $v:=max\_dimen$ 1146 ⟩;
  if $(v < max\_dimen)$ $v = v + d$;
  goto *not_found*;
found:
  if $(v < max\_dimen)$ { $v = v + d$;
    $w = v$;
  }
  else { $w = max\_dimen$;
    goto *done*;
  }
not_found: $p = link(p)$;
}
done:

This code is used in section 1144.

1146. ⟨ 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 $v:=max\_dimen$ 1146 ⟩ ≡

reswitch:
  if $(is\_char\_node(p))$ { $f = font(p)$;
    $d = char\_width(f, char\_info(f, character(p)))$;
    goto *found*;
  }
  switch $(type(p))$ {
  case *hlist_node*: **case** *vlist_node*: **case** *rule_node*:
    { $d = width(p)$;
      goto *found*;
    }
  case *ligature_node*: ⟨ Make node p look like a *char_node* and **goto** *reswitch* 651 ⟩
  case *kern_node*: **case** *math_node*: $d = width(p)$; **break**;
  case *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 1147 ⟩ **break**;
  case *whatsit_node*: ⟨ Let d be the width of the whatsit p 1360 ⟩; **break**;
  default: $d = 0$;
  }

This code is used in section 1145.

1147. 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: $=$ *max_dimen*; **goto** *found* in the case of leaders 1147⟩ ≡

```
{ q = glue_ptr(p);
  d = width(q);
  if (glue_sign(just_box) ≡ stretching) { if ((glue_order(just_box) ≡ stretch_order(q)) ∧
          (stretch(q) ≠ 0)) v = max_dimen;
  }
  else if (glue_sign(just_box) ≡ shrinking) { if ((glue_order(just_box) ≡ shrink_order(q)) ∧
          (shrink(q) ≠ 0)) v = max_dimen;
  }
  if (subtype(p) ≥ a_leaders) goto found;
}
```

This code is used in section 1146.

1148. 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 1148⟩ ≡

```
if (par_shape_ptr ≡ null)
  if ((hang_indent ≠ 0) ∧
        (((hang_after ≥ 0) ∧ (prev_graf + 2 > hang_after)) ∨
        (prev_graf + 1 < −hang_after))) { l = hsize − abs(hang_indent);
    if (hang_indent > 0) s = hang_indent; else s = 0;
  }
  else { l = hsize;
    s = 0;
  }
else { n = info(par_shape_ptr);
  if (prev_graf + 2 ≥ n) p = par_shape_ptr + 2 ∗ n;
  else p = par_shape_ptr + 2 ∗ (prev_graf + 2);
  s = mem[p − 1].sc;
  l = mem[p].sc;
}
```

This code is used in section 1144.

1149. 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 1055 ⟩ +≡
case *mmode* + *left_brace*:
  { *tail_append*(*new_noad*( ));
    back_input( );
    scan_math(*nucleus*(*tail*));
  } **break**;

1150. 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* $((cur\_fam \ge 0) \land (cur\_fam < 16))$

⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void scan_math(pointer p)
{ int c;    /* math character code */
restart: ⟨ Get the next non-blank non-relax non-call token 403 ⟩;
reswitch:
  switch (cur_cmd) {
  case letter: case other_char: case char_given:
    { c = ho(math_code(cur_chr));
      if (c ≡ °100000) { ⟨ Treat cur_chr as an active character 1151 ⟩;
        goto restart;
      }
    } break;
  case char_num:
    { scan_char_num();
      cur_chr = cur_val;
      cur_cmd = char_given;
      goto reswitch;
    }
  case math_char_num:
    { scan_fifteen_bit_int();
      c = cur_val;
    } break;
  case math_given: c = cur_chr; break;
  case delim_num:
    { scan_twenty_seven_bit_int();
      c = cur_val/°10000;
    } break;
  default: ⟨ Scan a subformula enclosed in braces and return 1152 ⟩
  }
  math_type(p) = math_char;
  character(p) = qi(c % 256);
  if ((c ≥ var_code) ∧ fam_in_range) fam(p) = cur_fam;
  else fam(p) = (c/256) % 16;
}
```

1151. An active character that is an *outer_call* is allowed here.

⟨ Treat *cur_chr* as an active character 1151 ⟩ ≡

```
{ cur_cs = cur_chr + active_base;
  cur_cmd = eq_type(cur_cs);
  cur_chr = equiv(cur_cs);
  x_token();
  back_input();
}
```

This code is used in sections 1150 and 1154.

1152. The pointer p is placed on *save_stack* while a complex subformula is being scanned.

⟨ Scan a subformula enclosed in braces and **return** 1152 ⟩ ≡
  { *back_input*( );
    scan_left_brace( );
    saved(0) = *p*;
    incr(*save_ptr*);
    push_math(*math_group*);
    return;
  }

This code is used in section 1150.

1153. 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 1055 ⟩ +≡
case *mmode* + *letter*: **case** *mmode* + *other_char*: **case** *mmode* + *char_given*:
  set_math_char(*ho*(*math_code*(*cur_chr*))); **break**;
case *mmode* + *char_num*:
  { *scan_char_num*( );
    cur_chr = *cur_val*;
    set_math_char(*ho*(*math_code*(*cur_chr*)));
  } **break**;
case *mmode* + *math_char_num*:
  { *scan_fifteen_bit_int*( );
    set_math_char(*cur_val*);
  } **break**;
case *mmode* + *math_given*: *set_math_char*(*cur_chr*); **break**;
case *mmode* + *delim_num*:
  { *scan_twenty_seven_bit_int*( );
    set_math_char(*cur_val* /°10000);
  } **break**;

1154. 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* 1042⟩ +≡

```
static void set_math_char(int c)
{ pointer p;      /* the new noad */
  if (c ≥ °100000) ⟨Treat cur_chr as an active character 1151⟩
  else { p = new_noad();
    math_type(nucleus(p)) = math_char;
    character(nucleus(p)) = qi(c % 256);
    fam(nucleus(p)) = (c/256) % 16;
    if (c ≥ var_code) { if (fam_in_range) fam(nucleus(p)) = cur_fam;
      type(p) = ord_noad;
    }
    else type(p) = ord_noad + (c/°10000);
    link(tail) = p;
    tail = p;
  }
}
```

1155. 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 225⟩ +≡

```
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);
```

1156. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case math_comp:
  switch (chr_code) {
  case ord_noad: print_esc("mathord"); break;
  case op_noad: print_esc("mathop"); break;
  case bin_noad: print_esc("mathbin"); break;
  case rel_noad: print_esc("mathrel"); break;
  case open_noad: print_esc("mathopen"); break;
  case close_noad: print_esc("mathclose"); break;
  case punct_noad: print_esc("mathpunct"); break;
  case inner_noad: print_esc("mathinner"); break;
  case under_noad: print_esc("underline"); break;
  default: print_esc("overline");
  } break;
case limit_switch:
  if (chr_code ≡ limits) print_esc("limits");
  else if (chr_code ≡ no_limits) print_esc("nolimits");
  else print_esc("displaylimits"); break;
```

1157. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
case mmode + math_comp:
  { tail_append(new_noad());
    type(tail) = cur_chr;
    scan_math(nucleus(tail));
  } break;
case mmode + limit_switch: math_limit_switch(); break;
```

1158. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void math_limit_switch(void)
{ if (head ≠ tail)
    if (type(tail) ≡ op_noad) { subtype(tail) = cur_chr;
      return;
    }
  print_err("Limit␣controls␣must␣follow␣a␣math␣operator");
  help1("I'm␣ignoring␣this␣misplaced␣\\limits␣or␣\\nolimits␣command.");
  error ( ) ;
}
```

1159. 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* 1042 ⟩ +≡
```
static void scan_delimiter(pointer p, bool r)
{ if (r) scan_twenty_seven_bit_int();
  else { ⟨Get the next non-blank non-relax non-call token 403⟩;
    switch (cur_cmd) {
    case letter: case other_char: cur_val = del_code(cur_chr); break;
    case delim_num: scan_twenty_seven_bit_int(); break;
    default: cur_val = −1;
    }
  }
  if (cur_val < 0)
    ⟨Report that an invalid delimiter code is being changed to null; set cur_val: = 0 1160⟩;
  small_fam(p) = (cur_val/°4000000) % 16;
  small_char(p) = qi((cur_val/°10000) % 256);
  large_fam(p) = (cur_val/256) % 16;
  large_char(p) = qi(cur_val % 256);
}
```

1160. ⟨ Report that an invalid delimiter code is being changed to null; set *cur_val*: = 0 1160 ⟩ ≡
```
{ 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;
}
```
This code is used in section 1159.

1161. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡
```
case mmode + radical: math_radical(); break;
```

1162. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡
```
static void math_radical(void)
{ 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));
}
```

1163. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡
```
case mmode + accent: case mmode + math_accent: math_ac(); break;
```

1164. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void math_ac(void)
{ if (cur_cmd ≡ accent) ⟨ Complain that the user should have said \mathaccent 1165 ⟩;
  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 % 256);
  if ((cur_val ≥ var_code) ∧ fam_in_range) fam(accent_chr(tail)) = cur_fam;
  else fam(accent_chr(tail)) = (cur_val/256) % 16;
  scan_math(nucleus(tail));
}
```

1165. ⟨ Complain that the user should have said `\mathaccent` 1165 ⟩ ≡

```
{ 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 ( ) ;
}
```

This code is used in section 1164.

1166. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
case mmode + vcenter:
  { scan_spec(vcenter_group, false);
    normal_paragraph();
    push_nest();
    mode = −vmode;
    prev_depth = ignore_depth;
    if (every_vbox ≠ null) begin_token_list(every_vbox, every_vbox_text);
  } break;
```

1167. ⟨ Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1084 ⟩ +≡

```
case vcenter_group:
  { 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;
  } break;
```

1168. The routine that inserts a *style_node* holds no surprises.

⟨Put each of TEX's primitives into the hash table 225⟩ +≡
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*);

1169. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 226⟩ +≡
case *math_style*: *print_style*(*chr_code*); **break**;

1170. ⟨Cases of *main_control* that build boxes and lists 1055⟩ +≡
case *mmode* + *math_style*: *tail_append*(*new_style*(*cur_chr*)) **break**;
case *mmode* + *non_script*:
{ *tail_append*(*new_glue*(*zero_glue*));
subtype(*tail*) = *cond_math_glue*;
} **break**;
case *mmode* + *math_choice*: *append_choices*( ); **break**;

1171. 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* 1042⟩ +≡
static void *append_choices*(**void**)
{ *tail_append*(*new_choice*( ));
incr(*save_ptr*);
saved(−1) = 0;
push_math(*math_choice_group*);
scan_left_brace( );
}

1172. ⟨Cases of *handle_right_brace* where a *right_brace* triggers a delayed action 1084⟩ +≡
case *math_choice_group*: *build_choices*( ); **break**;

1173. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡
⟨ Declare the function called *fin_mlist* 1183 ⟩

```
static void build_choices(void)
{ pointer p;    /* the current mlist */
  unsave();
  p = fin_mlist(null);
  switch (saved(−1)) {
  case 0: display_mlist(tail) = p; break;
  case 1: text_mlist(tail) = p; break;
  case 2: script_mlist(tail) = p; break;
  case 3:
    { script_script_mlist(tail) = p;
      decr(save_ptr);
      return;
    }
  }    /* there are no other cases */
  incr(saved(−1));
  push_math(math_choice_group);
  scan_left_brace();
}
```

1174. Subscripts and superscripts are attached to the previous nucleus by the action procedure called *sub_sup*. We use the facts that $sub\_mark \equiv sup\_mark + 1$ and $subscr(p) \equiv supscr(p) + 1$.

⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
case mmode + sub_mark: case mmode + sup_mark: sub_sup(); break;
```

1175. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void sub_sup(void)
{ small_number t;    /* type of previous sub/superscript */
  pointer p;    /* field to be filled by scan_math */
  t = empty;
  p = null;
  if (tail ≠ head)
    if (scripts_allowed(tail)) { p = supscr(tail) + cur_cmd − sup_mark;    /* supscr or subscr */
      t = math_type(p);
    }
  if ((p ≡ null) ∨ (t ≠ empty)) ⟨ Insert a dummy noad to be sub/superscripted 1176 ⟩;
  scan_math(p);
}
```

1176. ⟨ Insert a dummy noad to be sub/superscripted 1176 ⟩ ≡

```
{ tail_append(new_noad());
  p = supscr(tail) + cur_cmd − sup_mark;    /* supscr or subscr */
  if (t ≠ empty) { if (cur_cmd ≡ sup_mark) { print_err("Double superscript");
      help1("I treat `x^1^2' essentially like `x^1{}^2'.");
    }
    else { print_err("Double subscript");
      help1("I treat `x_1_2' essentially like `x_1{}_2'.");
    }
    error();
  }
}
```

This code is used in section 1175.

1177. 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 225 ⟩ +≡

```
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);
```

1178. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case above:
  switch (chr_code) {
  case over_code: print_esc("over"); break;
  case atop_code: print_esc("atop"); break;
  case delimited_code + above_code: print_esc("abovewithdelims"); break;
  case delimited_code + over_code: print_esc("overwithdelims"); break;
  case delimited_code + atop_code: print_esc("atopwithdelims"); break;
  default: print_esc("above");
  } break;
```

1179. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
case mmode + above: math_fraction(); break;
```

1180. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void math_fraction(void)
{ small_number c;    /* the type of generalized fraction we are scanning */
  c = cur_chr;
  if (incompleat_noad ≠ null)
    ⟨ Ignore the fraction operation and complain about this ambiguous case 1182 ⟩
  else { 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 1181 ⟩;
  }
}
```

1181. ⟨ Use code *c* to distinguish between generalized fractions 1181 ⟩ ≡

```
if (c ≥ delimited_code) { scan_delimiter(left_delimiter(incompleat_noad), false);
  scan_delimiter(right_delimiter(incompleat_noad), false);
}
switch (c % delimited_code) {
case above_code:
  { scan_normal_dimen;
    thickness(incompleat_noad) = cur_val;
  } break;
case over_code: thickness(incompleat_noad) = default_code; break;
case atop_code: thickness(incompleat_noad) = 0;
}    /* there are no other cases */
```

This code is used in section 1180.

1182. ⟨ Ignore the fraction operation and complain about this ambiguous case 1182 ⟩ ≡

```
{ if (c ≥ delimited_code) { scan_delimiter(garbage, false);
    scan_delimiter(garbage, false);
  }
  if (c % delimited_code ≡ above_code) 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();
}
```

This code is used in section 1180.

1183. 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* 1183⟩ ≡

```
static pointer fin_mlist(pointer p)
{ pointer q;     /* the mlist to return */
  if (incompleat_noad ≠ null) ⟨Compleat the incompleat noad 1184⟩
  else { link(tail) = p;
    q = link(head);
  }
  pop_nest();
  return q;
}
```

This code is used in section 1173.

1184. ⟨Compleat the incompleat noad 1184⟩ ≡

```
{ math_type(denominator(incompleat_noad)) = sub_mlist;
  info(denominator(incompleat_noad)) = link(head);
  if (p ≡ null) q = incompleat_noad;
  else { q = info(numerator(incompleat_noad));
    if ((type(q) ≠ left_noad) ∨ (delim_ptr ≡ null)) confusion("right");
    info(numerator(incompleat_noad)) = link(delim_ptr);
    link(delim_ptr) = incompleat_noad;
    link(incompleat_noad) = p;
  }
}
```

This code is used in section 1183.

1185. 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 1084⟩ +≡

```
case math_group:
  { unsave();
    decr(save_ptr);
    math_type(saved(0)) = sub_mlist;
    p = fin_mlist(null);
    info(saved(0)) = p;
    if (p ≠ null)
      if (link(p) ≡ null)
        if (type(p) ≡ ord_noad) { if (math_type(subscr(p)) ≡ empty)
            if (math_type(supscr(p)) ≡ empty) { mem[saved(0)].hh = mem[nucleus(p)].hh;
              free_node(p, noad_size);
            }
        }
        else if (type(p) ≡ accent_noad)
          if (saved(0) ≡ nucleus(tail))
            if (type(tail) ≡ ord_noad) ⟨Replace the tail of the list by p 1186⟩;
  } break;
```

1186. ⟨ Replace the tail of the list by p 1186 ⟩ ≡

```
{ q = head;
  while (link(q) ≠ tail) q = link(q);
  link(q) = p;
  free_node(tail, noad_size);
  tail = p;
}
```

This code is used in section 1185.

1187. 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 225 ⟩ +≡

```
primitive("left", left_right, left_noad);
primitive("right", left_right, right_noad);
text(frozen_right) = text(cur_val);
eqtb[frozen_right] = eqtb[cur_val];
```

1188. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case left_right: if (chr_code ≡ left_noad) print_esc("left")
⟨ Cases of left_right for print_cmd_chr 1428 ⟩;
else print_esc("right"); break;
```

1189. ⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
case mmode + left_right: math_left_right(); break;
```

1190. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void math_left_right(void)
{ small_number t;    /* left_noad or right_noad */
  pointer p;    /* new noad */
  pointer q;    /* resulting mlist */
  t = cur_chr;
  if ((t ≠ left_noad) ∧ (cur_group ≠ math_left_group)) ⟨ Try to recover from mismatched \right 1191 ⟩
  else { p = new_noad();
    type(p) = t;
    scan_delimiter(delimiter(p), false);
    if (t ≡ middle_noad) { type(p) = right_noad;
      subtype(p) = middle_noad;
    }
    if (t ≡ left_noad) q = p;
    else { q = fin_mlist(p);
      unsave();    /* end of math_left_group */
    }
    if (t ≠ right_noad) { push_math(math_left_group);
      link(head) = q;
      tail = p;
      delim_ptr = p;
    }
    else { tail_append(new_noad());
      type(tail) = inner_noad;
      math_type(nucleus(tail)) = sub_mlist;
      info(nucleus(tail)) = q;
    }
  }
}
```

1191. ⟨ Try to recover from mismatched `\right` 1191 ⟩ ≡

```
{ if (cur_group ≡ math_shift_group) { scan_delimiter(garbage, false);
    print_err("Extra␣");
    if (t ≡ middle_noad) { print_esc("middle");
      help1("I'm␣ignoring␣a␣\\middle␣that␣had␣no␣matching␣\\left.");
    }
    else { print_esc("right");
      help1("I'm␣ignoring␣a␣\\right␣that␣had␣no␣matching␣\\left.");
    }
    error ( ) ;
  }
  else off_save();
}
```

This code is used in section 1190.

1192. Here is the only way out of math mode.

⟨ Cases of *main_control* that build boxes and lists 1055 ⟩ +≡

```
case mmode + math_shift:
  if (cur_group ≡ math_shift_group) after_math();
  else off_save(); break;
```

1193. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

static void *after_math*(**void**)
{ **bool** *l*; /* '\leqno' instead of '\eqno' */
  bool *danger*; /* not enough symbol fonts are present */
  int *m*; /* *mmode* or −*mmode* */
  pointer *p*; /* the formula */
  pointer *a*; /* box containing equation number */
  ⟨ Local variables for finishing a displayed formula 1197 ⟩
  danger = *false*;
  ⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger*: = *true* 1194 ⟩;
  m = *mode*;
  l = *false*;
  p = *fin_mlist*(*null*); /* this pops the nest */
  if (*mode* ≡ −*m*) /* end of equation number */
  { ⟨ Check that another $ follows 1196 ⟩;
    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) *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* 1194 ⟩;
    m = *mode*;
    p = *fin_mlist*(*null*);
  }
  else *a* = *null*;
  if (*m* < 0) ⟨ Finish math in text 1195 ⟩
  else { **if** (*a* ≡ *null*) ⟨ Check that another $ follows 1196 ⟩;
    ⟨ Finish displayed math 1198 ⟩;
  }
}

1194. ⟨ Check that the necessary fonts for math symbols are present; if not, flush the current math lists and set *danger*: = *true* 1194 ⟩ ≡

```
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)) {
    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;
}
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)) {
    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;
}
```

This code is used in section 1193.

1195. 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 1195 ⟩ ≡

```
{ 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) tail = link(tail);
    tail_append(new_math(math_surround, after));
    space_factor = 1000;
    unsave( );
}
```

This code is used in section 1193.

1196. TEX gets to the following part of the program when the first '$' ending a display has been scanned.

⟨ Check that another $ follows 1196 ⟩ ≡

```
{ get_x_token();
  if (cur_cmd ≠ math_shift) { 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();
  }
}
```

This code is used in sections 1193 and 1205.

1197. 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 1197 ⟩ ≡

```
pointer b; /* box containing the equation */
scaled w; /* width of the equation */
scaled z; /* width of the line */
scaled e; /* width of equation number */
scaled q; /* width of equation number plus space to separate from equation */
scaled d; /* displacement of equation in the line */
scaled s; /* move the line right this much */
small_number g1, g2; /* glue parameter codes for before and after */
pointer r; /* kern node used to position the display */
pointer t; /* tail of adjustment list */
```

This code is used in section 1193.

1198. 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 1198 ⟩ ≡
$cur\_mlist = p$;
$cur\_style = display\_style$;
$mlist\_penalties = false$;
$mlist\_to\_hlist()$;
$p = link(temp\_head)$;
$adjust\_tail = adjust\_head$;
$b = hpack(p, natural)$;
$p = list\_ptr(b)$;
$t = adjust\_tail$;
$adjust\_tail = null$;
$w = width(b)$;
$z = display\_width$;
$s = display\_indent$;
if $((a \equiv null) \lor danger)$ { $e = 0$;
  $q = 0$;
}
else { $e = width(a)$;
  $q = e + math\_quad(text\_size)$;
}
if $(w + q > z)$ ⟨ Squeeze the equation as much as possible; if there is an equation number that should go on a separate line by itself, set $e\colon = 0$ 1200 ⟩;
⟨ Determine the displacement, d, of the left edge of the equation, with respect to the line size z, assuming that $l = false$ 1201 ⟩;
⟨ Append the glue or equation number preceding the display 1202 ⟩;
⟨ Append the display and perhaps also the equation number 1203 ⟩;
⟨ Append the glue or equation number following the display 1204 ⟩;
$resume\_after\_display()$

This code is used in section 1193.

1199. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡
static void *resume_after_display*(**void**)
{ **if** $(cur\_group \neq math\_shift\_group)$ $confusion(\texttt{"display"})$;
  $unsave()$;
  $prev\_graf = prev\_graf + 3$;
  $push\_nest()$;
  $mode = hmode$;
  $space\_factor = 1000$;
  set_cur_lang;
  $clang = cur\_lang$;
  $prev\_graf = (norm\_min(left\_hyphen\_min) * {}^\circ 100 + norm\_min(right\_hyphen\_min)) * {}^\circ 200000 + cur\_lang$;
  ⟨ Scan an optional space 442 ⟩;
  if $(nest\_ptr \equiv 1)$ $build\_page()$;
}

1200. The user can force the equation number to go on a separate line by causing its width to be zero.

⟨ Squeeze the equation as much as possible; if there is an equation number that should go on a separate line by itself, set e: $= 0$ 1200 ⟩ ≡

$\{$ **if** $((e \neq 0) \wedge ((w - \mathit{total\_shrink}[\mathit{normal}] + q \leq z) \vee$
$(\mathit{total\_shrink}[\mathit{fil}] \neq 0) \vee (\mathit{total\_shrink}[\mathit{fill}] \neq 0) \vee (\mathit{total\_shrink}[\mathit{filll}] \neq 0)))$ $\{$
$\mathit{free\_node}(b, \mathit{box\_node\_size})$;
$b = \mathit{hpack}(p, z - q, \mathit{exactly})$;
$\}$
else $\{$ $e = 0$;
if $(w > z)$ $\{$ $\mathit{free\_node}(b, \mathit{box\_node\_size})$;
$b = \mathit{hpack}(p, z, \mathit{exactly})$;
$\}$
$\}$
$w = \mathit{width}(b)$;
$\}$

This code is used in section 1198.

1201. 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 $l = \mathit{false}$ 1201 ⟩ ≡

$d = \mathit{half}(z - w)$;
if $((e > 0) \wedge (d < 2 * e))$ /∗ too close ∗/
$\{$ $d = \mathit{half}(z - w - e)$;
if $(p \neq \mathit{null})$
if $(\neg \mathit{is\_char\_node}(p))$
if $(\mathit{type}(p) \equiv \mathit{glue\_node})$ $d = 0$;
$\}$

This code is used in section 1198.

1202. 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 1202 ⟩ ≡
  tail_append(*new_penalty*(*pre_display_penalty*));
  if $((d + s \le pre\_display\_size) \lor l)$ /* not enough clearance */
  { *g1* = *above_display_skip_code*;
    g2 = *below_display_skip_code*;
  }
  else { *g1* = *above_display_short_skip_code*;
    g2 = *below_display_short_skip_code*;
  }
  if $(l \land (e \equiv 0))$ /* it follows that *type*(*a*) ≡ *hlist_node* */
  { *shift_amount*(*a*) = *s*;
    append_to_vlist(*a*);
    tail_append(*new_penalty*(*inf_penalty*));
  }
  else *tail_append*(*new_param_glue*(*g1*))

This code is used in section 1198.

1203. ⟨ Append the display and perhaps also the equation number 1203 ⟩ ≡
  if $(e \ne 0)$ { $r = new\_kern(z - w - e - d)$;
    if (l) { *link*(*a*) = *r*;
      link(*r*) = *b*;
      b = *a*;
      d = 0;
    }
    else { *link*(*b*) = *r*;
      link(*r*) = *a*;
    }
    b = *hpack*(*b*, *natural*);
  }
  shift_amount(*b*) = *s* + *d*; *append_to_vlist*(*b*)

This code is used in section 1198.

1204. ⟨ Append the glue or equation number following the display 1204 ⟩ ≡
  if $((a \ne null) \land (e \equiv 0) \land \neg l)$ { *tail_append*(*new_penalty*(*inf_penalty*));
    shift_amount(*a*) = *s* + *z* − *width*(*a*);
    append_to_vlist(*a*);
    g2 = 0;
  }
  if $(t \ne adjust\_head)$ /* migrating material comes after equation number */
  { *link*(*tail*) = *link*(*adjust_head*);
    tail = *t*;
  }
  tail_append(*new_penalty*(*post_display_penalty*)); **if** $(g2 > 0)$ *tail_append*(*new_param_glue*(*g2*))

This code is used in section 1198.

1205. 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 1205 ⟩ ≡
{ *do_assignments*( );
  if (*cur_cmd* ≠ *math_shift*) ⟨ Pontificate about improper alignment in display 1206 ⟩
  else ⟨ Check that another `$` follows 1196 ⟩;
  pop_nest( );
  tail_append(*new_penalty*(*pre_display_penalty*));
  tail_append(*new_param_glue*(*above_display_skip_code*));
  link(*tail*) = *p*;
  if (*p* ≠ *null*) *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( );
}

This code is used in section 811.

1206. ⟨ Pontificate about improper alignment in display 1206 ⟩ ≡
{ *print_err*(`"Missing␣$$␣inserted"`);
  help2(`"Displays␣can␣use␣special␣alignments␣(like␣\\eqalignno)"`,
  `"only␣if␣nothing␣but␣the␣alignment␣itself␣is␣between␣$$'s."`);
  back_error( );
}

This code is used in section 1205.

1207. Mode-independent processing. The long *main_control* procedure has now been fully specified, except for certain activities that are independent of the current mode. These activities do not change the current vlist or hlist or mlist; if they change anything, it is the value of a parameter or the meaning of a control sequence.

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

⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
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);

1208. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *prefix*:
if (*chr_code* ≡ 1) *print_esc*("long");
else if (*chr_code* ≡ 2) *print_esc*("outer")
⟨ Cases of *prefix* for *print_cmd_chr* 1453 ⟩;
else *print_esc*("global"); **break**;
case *def*:
if (*chr_code* ≡ 0) *print_esc*("def");
else if (*chr_code* ≡ 1) *print_esc*("gdef");
else if (*chr_code* ≡ 2) *print_esc*("edef");
else *print_esc*("xdef"); **break**;

1209. 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* 1209 ⟩ ≡
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(*internal_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*( ); **break**;

See also sections 1267, 1270, 1273, 1275, 1284, and 1289.

This code is used in section 1044.

1210. If the user says, e.g., '\global\global', the redundancy is silently accepted.

⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡
⟨ Declare subprocedures for *prefixed_command* 1214 ⟩

```
static void prefixed_command(void)
{ small_number a;    /* accumulated prefix codes so far */
  internal_font_number f;    /* identifies a font */
  int j;    /* index into a \parshape specification */
  font_index k;    /* index into font_info */
  pointer p, q;    /* for temporary short-term use */
  int n;    /* ditto */
  bool e;    /* should a definition be expanded? or was \let not done? */
  a = 0;
  while (cur_cmd ≡ prefix) { if (¬odd(a/cur_chr)) a = a + cur_chr;
    ⟨ Get the next non-blank non-relax non-call token 403 ⟩;
    if (cur_cmd ≤ max_non_prefixed_command) ⟨ Discard erroneous prefixes and return 1211 ⟩;
    if (tracing_commands > 2)
      if (eTeX_ex) show_cur_cmd_chr();
  }
  ⟨ Discard the prefixes \long and \outer if they are irrelevant 1212 ⟩;
  ⟨ Adjust for the setting of \globaldefs 1213 ⟩;
  switch (cur_cmd) { ⟨ Assignments 1216 ⟩
  default: confusion("prefix");
  }
done: ⟨ Insert a token saved by \afterassignment, if any 1268 ⟩;
}
```

1211. ⟨ Discard erroneous prefixes and **return** 1211 ⟩ ≡

```
{ print_err("You␣can't␣use␣a␣prefix␣with␣`");
  print_cmd_chr(cur_cmd, cur_chr);
  print_char('\'');
  help1("I'll␣pretend␣you␣didn't␣say␣\\long␣or␣\\outer␣or␣\\global.");
  if (eTeX_ex) help_line[0] =
      "I'll␣pretend␣you␣didn't␣say␣\\long␣or␣\\outer␣or␣\\global␣or␣\\protected.";
  back_error();
  return;
}
```

This code is used in section 1210.

1212. ⟨ Discard the prefixes `\long` and `\outer` if they are irrelevant 1212 ⟩ ≡

```
if (a ≥ 8) { j = protected_token;
    a = a − 8;
}
else j = 0;
if ((cur_cmd ≠ def) ∧ ((a % 4 ≠ 0) ∨ (j ≠ 0))) { print_err("You␣can't␣use␣`");
    print_esc("long");
    print("'␣or␣`");
    print_esc("outer");
    help1("I'll␣pretend␣you␣didn't␣say␣\\long␣or␣\\outer␣here.");
    if (eTeX_ex) { help_line[0] =
            "I'll␣pretend␣you␣didn't␣say␣\\long␣or␣\\outer␣or␣\\protected␣here.";
        print("'␣or␣`");
        print_esc("protected");
    }
    print("'␣with␣`");
    print_cmd_chr(cur_cmd, cur_chr);
    print_char('\'');
    error ( ) ;
}
```

This code is used in section 1210.

1213. The previous routine does not have to adjust a so that $a \% 4 \equiv 0$, since the following routines test for the `\global` prefix as follows.

```
#define global (a ≥ 4)
#define define(A, B, C)
        if (global) geq_define(A, B, C); else eq_define(A, B, C)
#define word_define(A, B)
        if (global) geq_word_define(A, B); else eq_word_define(A, B)
```

⟨ Adjust for the setting of `\globaldefs` 1213 ⟩ ≡

```
if (global_defs ≠ 0)
    if (global_defs < 0) { if (global) a = a − 4;
    }
    else { if (¬global) a = a + 4;
    }
```

This code is used in section 1210.

1214. 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* 1214 ⟩ ≡

```
static void get_r_token(void)
{ restart:
    do { get_token();
    } while (¬(cur_tok ≠ space_token));
    if ((cur_cs ≡ 0) ∨ (cur_cs > frozen_control_sequence)) {
      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) back_input();
      cur_tok = cs_token_flag + frozen_protection;
      ins_error();
      goto restart;
    }
}
```

See also sections 1228, 1235, 1242, 1243, 1244, 1245, 1246, 1256, and 1264.

This code is used in section 1210.

1215. ⟨ Initialize table entries (done by `INITEX` only) 163 ⟩ +≡

```
text(frozen_protection) = s_no("inaccessible");
```

1216. Here's an example of the way many of the following routines operate. (Unfortunately, they aren't all as simple as this.)

⟨ Assignments 1216 ⟩ ≡

```
case set_font: define (cur_font_loc, data, cur_chr); break;
```

See also sections 1217, 1220, 1223, 1224, 1225, 1227, 1231, 1233, 1234, 1240, 1241, 1247, 1251, 1252, 1255, and 1263.

This code is used in section 1210.

1217. 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 1216 ⟩ +≡

```
case def: { uint32_t def_fl;
  if (odd(cur_chr) ∧ ¬global ∧ (global_defs ≥ 0)) a = a + 4;
  e = (cur_chr ≥ 2);
  get_r_token();
  p = cur_cs;
  def_fl = cur_file_line;
  q = scan_toks(true, e);
  if (j ≠ 0) { q = get_avail();
    info(q) = j;
    link(q) = link(def_ref);
    link(def_ref) = q;
  }
  define (p, call + (a % 4), def_ref);
  fl_mem[def_ref] = def_fl;
} break;
```

1218. Both `\let` and `\futurelet` share the command code *let*.

⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡

```
primitive("let", let, normal);
primitive("futurelet", let, normal + 1);
```

1219. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case let:
  if (chr_code ≠ normal) print_esc("futurelet"); else print_esc("let"); break;
```

1220. ⟨ Assignments 1216 ⟩ +≡

```
case let: { n = cur_chr;
  get_r_token();
  p = cur_cs;
  if (n ≡ normal) { do { get_token();
    } while (¬(cur_cmd ≠ spacer));
    if (cur_tok ≡ other_token + '=') { get_token();
      if (cur_cmd ≡ spacer) get_token();
    }
  }
  else { get_token();
    q = cur_tok;
    get_token();
    back_input();
    cur_tok = q;
    back_input();    /* look ahead, then back up */
  }    /* note that back_input doesn't affect cur_cmd, cur_chr */
  if (cur_cmd ≥ call) add_token_ref(cur_chr);
  else if ((cur_cmd ≡ internal_register) ∨ (cur_cmd ≡ toks_register))
    if ((cur_chr < mem_bot) ∨ (cur_chr > lo_mem_stat_max)) add_sa_ref(cur_chr);
  define(p, cur_cmd, cur_chr);
} break;
```

1221. 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 225 ⟩ +≡

```
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);
```

1222. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case shorthand_def:
  switch (chr_code) {
  case char_def_code: print_esc("chardef"); break;
  case math_char_def_code: print_esc("mathchardef"); break;
  case count_def_code: print_esc("countdef"); break;
  case dimen_def_code: print_esc("dimendef"); break;
  case skip_def_code: print_esc("skipdef"); break;
  case mu_skip_def_code: print_esc("muskipdef"); break;
  default: print_esc("toksdef");
  } break;
case char_given:
  { print_esc("char");
    print_hex(chr_code);
  } break;
case math_given:
  { print_esc("mathchar");
    print_hex(chr_code);
  } break;
```

1223. 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 1216 ⟩ +≡

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

1224. ⟨ Assignments 1216 ⟩ +≡

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

1225. 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 1216 ⟩ +≡
case *toks_register*: **case** *assign_toks*:
  { **uint32_t** *def_fl* = *cur_file_line*;
    q = *cur_cs*;
    e = *false*;   /* just in case, will be set *true* for sparse array elements */
    if (*cur_cmd* ≡ *toks_register*)
      if (*cur_chr* ≡ *mem_bot*) { *scan_register_num*( );
        if (*cur_val* > 255) { *find_sa_element*(*tok_val*, *cur_val*, *true*);
          cur_chr = *cur_ptr*;
          e = *true*;
        }
        else *cur_chr* = *toks_base* + *cur_val*;
      }
      else *e* = *true*;
    p = *cur_chr*;   /* *p* ≡ *every_par_loc* or *output_routine_loc* or . . . */
    scan_optional_equals( );
    ⟨ Get the next non-blank non-relax non-call token 403 ⟩;
    if (*cur_cmd* ≠ *left_brace*) ⟨ If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1226 ⟩;
    back_input( );
    cur_cs = *q*;
    q = *scan_toks*(*false*, *false*);
    if (*link*(*def_ref*) ≡ *null*)   /* empty list: revert to the default */
    { *sa_define*(*p*, *null*, *p*, *undefined_cs*, *null*);
      free_avail(*def_ref*);
    }
    else { **if** ((*p* ≡ *output_routine_loc*) ∧ ¬*e*)   /* enclose in curlies */
      { *link*(*q*) = *get_avail*( );
        q = *link*(*q*);
        info(*q*) = *right_brace_token* + '}';
        fl_mem[*q*] = `FILE_LINE`(*system_file*, *system_insert*);
        q = *get_avail*( );
        info(*q*) = *left_brace_token* + '{';
        fl_mem[*q*] = `FILE_LINE`(*system_file*, *system_insert*);
        link(*q*) = *link*(*def_ref*);
        link(*def_ref*) = *q*;
      }
      sa_define(*p*, *def_ref*, *p*, *call*, *def_ref*);
      fl_mem[*def_ref*] = *def_fl*;
    }
  } **break**;

1226. ⟨If the right-hand side is a token parameter or token register, finish the assignment and **goto** *done* 1226⟩ ≡

if $((cur\_cmd \equiv toks\_register) \lor (cur\_cmd \equiv assign\_toks))$ { **if** $(cur\_cmd \equiv toks\_register)$
  if $(cur\_chr \equiv mem\_bot)$ { $scan\_register\_num()$;
   if $(cur\_val < 256)$ $q = equiv(toks\_base + cur\_val)$;
   else { $find\_sa\_element(tok\_val, cur\_val, false)$;
    if $(cur\_ptr \equiv null)$ $q = null$;
    else $q = sa\_ptr(cur\_ptr)$;
   }
  }
  else $q = sa\_ptr(cur\_chr)$;
 else $q = equiv(cur\_chr)$;
 if $(q \equiv null)$ $sa\_define(p, null, p, undefined\_cs, null)$;
 else { $add\_token\_ref(q)$;
  $sa\_define(p, q, p, call, q)$;
 }
 goto *done*;
}

This code is used in section 1225.

1227. Similar routines are used to assign values to the numeric parameters.

⟨Assignments 1216⟩ +≡
case *assign_int*:
 { $p = cur\_chr$;
  $scan\_optional\_equals()$;
  $scan\_int()$;
  $word\_define(p, cur\_val)$;
 } **break**;
case *assign_dimen*:
 { $p = cur\_chr$;
  $scan\_optional\_equals()$;
  $scan\_normal\_dimen$;
  $word\_define(p, cur\_val)$;
 } **break**; **case** *assign_glue*: **case** *assign_mu_glue*: { $p = cur\_chr$;
 $n = cur\_cmd$;
 $scan\_optional\_equals()$;
 if $(n \equiv assign\_mu\_glue)$ $scan\_glue(mu\_val)$; **else** $scan\_glue(glue\_val)$;
 $trap\_zero\_glue()$; **define** $(p, glue\_ref, cur\_val)$;
 } **break**;

1228. 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* 1214⟩ +≡
 static void *trap_zero_glue*(**void**)
 { **if** $((width(cur\_val) \equiv 0) \land (stretch(cur\_val) \equiv 0) \land (shrink(cur\_val) \equiv 0))$ { $add\_glue\_ref(zero\_glue)$;
   $delete\_glue\_ref(cur\_val)$;
   $cur\_val = zero\_glue$;
  }
 }

1229. 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 225 ⟩ +≡
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*);

1230. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *def_code*:
if (*chr_code* ≡ *cat_code_base*) *print_esc*("catcode");
else if (*chr_code* ≡ *math_code_base*) *print_esc*("mathcode");
else if (*chr_code* ≡ *lc_code_base*) *print_esc*("lccode");
else if (*chr_code* ≡ *uc_code_base*) *print_esc*("uccode");
else if (*chr_code* ≡ *sf_code_base*) *print_esc*("sfcode");
else *print_esc*("delcode"); **break**;
case *def_family*: *print_size*(*chr_code* − *math_font_base*); **break**;

1231. The different types of code values have different legal ranges; the following program is careful to check each case properly.

⟨ Assignments 1216 ⟩ +≡
case *def_code*: { ⟨ Let n be the largest legal code value, based on *cur_chr* 1232 ⟩;
$p = cur\_chr$;
scan_char_num( );
$p = p + cur\_val$;
scan_optional_equals( );
scan_int( );
if $(((cur\_val < 0) \land (p < del\_code\_base)) \lor (cur\_val > n))$ { *print_err*("Invalid␣code␣(");
print_int(*cur_val*);
if (*p* < *del_code_base*) *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;
}
if (*p* < *math_code_base*) **define** (*p*, *data*, *cur_val*); **else if** (*p* < *del_code_base*) **define** (*p*, *data*, *hi*(*cur_val*));
else *word_define*(*p*, *cur_val*);
} **break**;

1232. ⟨ Let n be the largest legal code value, based on *cur_chr* 1232 ⟩ ≡

```
if (cur_chr ≡ cat_code_base) n = max_char_code;
else if (cur_chr ≡ math_code_base) n = °100000;
else if (cur_chr ≡ sf_code_base) n = °77777;
else if (cur_chr ≡ del_code_base) n = °77777777;
else n = 255
```

This code is used in section 1231.

1233. ⟨ Assignments 1216 ⟩ +≡

```
case def_family: { p = cur_chr;
scan_four_bit_int();
p = p + cur_val;
scan_optional_equals();
scan_font_ident(); define (p, data, cur_val);
} break;
```

1234. Next we consider changes to TEX's numeric registers.

⟨ Assignments 1216 ⟩ +≡
case *internal_register*: **case** *advance*: **case** *multiply*: **case** *divide*: *do_register_command*(a); **break**;

1235. We use the fact that *internal_register* < *advance* < *multiply* < *divide*.

⟨ Declare subprocedures for *prefixed_command* 1214 ⟩ +≡

```
static void do_register_command(small_number a)
{ pointer l, q, r, s;   /* for list manipulation */
  int p;   /* type of register involved */
  bool e;   /* does l refer to a sparse array element? */
  int w;   /* integer or dimen value of l */

  q = cur_cmd;
  e = false;   /* just in case, will be set true for sparse array elements */
  ⟨ Compute the register location l and its type p; but return if invalid 1236 ⟩;
  if (q ≡ internal_register) scan_optional_equals();
  else if (scan_keyword("by")) do_nothing;   /* optional 'by' */
  arith_error = false;
  if (q < multiply) ⟨ Compute result of register or advance, put it in cur_val 1237 ⟩
  else ⟨ Compute result of multiply or divide, put it in cur_val 1239 ⟩;
  if (arith_error) { print_err("Arithmetic␣overflow");
    help2("I␣can't␣carry␣out␣that␣multiplication␣or␣division,",
    "since␣the␣result␣is␣out␣of␣range.");
    if (p ≥ glue_val) delete_glue_ref(cur_val);
    error ( ) ;
    return;
  }
  if (p < glue_val) sa_word_define(l, cur_val);
  else { trap_zero_glue();
    sa_define(l, cur_val, l, glue_ref, cur_val);
  }
}
```

1236. 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 1236⟩ ≡

```
{ if (q ≠ internal_register) { get_x_token();
    if ((cur_cmd ≥ assign_int) ∧ (cur_cmd ≤ assign_mu_glue)) { l = cur_chr;
      p = cur_cmd − assign_int;
      goto found;
    }
    if (cur_cmd ≠ internal_register) { 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;
    }
  }
  if ((cur_chr < mem_bot) ∨ (cur_chr > lo_mem_stat_max)) { l = cur_chr;
    p = sa_type(l);
    e = true;
  }
  else { p = cur_chr − mem_bot;
    scan_register_num();
    if (cur_val > 255) { find_sa_element(p, cur_val, true);
      l = cur_ptr;
      e = true;
    }
    else
      switch (p) {
      case int_val: l = cur_val + count_base; break;
      case dimen_val: l = cur_val + scaled_base; break;
      case glue_val: l = cur_val + skip_base; break;
      case mu_val: l = cur_val + mu_skip_base;
      }   /* there are no other cases */
  }
}
found:
if (p < glue_val) if (e) w = sa_int(l); else w = eqtb[l].i;
else if (e) s = sa_ptr(l); else s = equiv(l)
```

This code is used in section 1235.

1237. ⟨Compute result of **register** or *advance*, put it in *cur_val* 1237⟩ ≡

```
if (p < glue_val) { if (p ≡ int_val) scan_int(); else scan_normal_dimen;
  if (q ≡ advance) cur_val = cur_val + w;
}
else { scan_glue(p);
  if (q ≡ advance) ⟨Compute the sum of two glue specs 1238⟩;
}
```

This code is used in section 1235.

1238. ⟨Compute the sum of two glue specs 1238⟩ ≡

$\{\ q = \mathit{new\_spec}(\mathit{cur\_val});$
$\quad r = s;$
$\quad \mathit{delete\_glue\_ref}(\mathit{cur\_val});$
$\quad \mathit{width}(q) = \mathit{width}(q) + \mathit{width}(r);$
$\quad$ **if** $(\mathit{stretch}(q) \equiv 0)\ \mathit{stretch\_order}(q) = \mathit{normal};$
$\quad$ **if** $(\mathit{stretch\_order}(q) \equiv \mathit{stretch\_order}(r))\ \mathit{stretch}(q) = \mathit{stretch}(q) + \mathit{stretch}(r);$
$\quad$ **else if** $((\mathit{stretch\_order}(q) < \mathit{stretch\_order}(r)) \land (\mathit{stretch}(r) \neq 0))\ \{\ \mathit{stretch}(q) = \mathit{stretch}(r);$
$\qquad \mathit{stretch\_order}(q) = \mathit{stretch\_order}(r);$
$\quad \}$
$\quad$ **if** $(\mathit{shrink}(q) \equiv 0)\ \mathit{shrink\_order}(q) = \mathit{normal};$
$\quad$ **if** $(\mathit{shrink\_order}(q) \equiv \mathit{shrink\_order}(r))\ \mathit{shrink}(q) = \mathit{shrink}(q) + \mathit{shrink}(r);$
$\quad$ **else if** $((\mathit{shrink\_order}(q) < \mathit{shrink\_order}(r)) \land (\mathit{shrink}(r) \neq 0))\ \{\ \mathit{shrink}(q) = \mathit{shrink}(r);$
$\qquad \mathit{shrink\_order}(q) = \mathit{shrink\_order}(r);$
$\quad \}$
$\quad \mathit{cur\_val} = q;$
$\}$

This code is used in section 1237.

1239. ⟨Compute result of *multiply* or *divide*, put it in *cur_val* 1239⟩ ≡

$\{\ \mathit{scan\_int}(\,);$
$\quad$ **if** $(p < \mathit{glue\_val})$
$\qquad$ **if** $(q \equiv \mathit{multiply})$
$\qquad\quad$ **if** $(p \equiv \mathit{int\_val})\ \mathit{cur\_val} = \mathit{mult\_integers}(w, \mathit{cur\_val});$
$\qquad\quad$ **else** $\mathit{cur\_val} = \mathit{nx\_plus\_y}(w, \mathit{cur\_val}, 0);$
$\qquad$ **else** $\mathit{cur\_val} = \mathit{x\_over\_n}(w, \mathit{cur\_val});$
$\quad$ **else** $\{\ r = \mathit{new\_spec}(s);$
$\qquad$ **if** $(q \equiv \mathit{multiply})\ \{\ \mathit{width}(r) = \mathit{nx\_plus\_y}(\mathit{width}(s), \mathit{cur\_val}, 0);$
$\qquad\quad \mathit{stretch}(r) = \mathit{nx\_plus\_y}(\mathit{stretch}(s), \mathit{cur\_val}, 0);$
$\qquad\quad \mathit{shrink}(r) = \mathit{nx\_plus\_y}(\mathit{shrink}(s), \mathit{cur\_val}, 0);$
$\qquad \}$
$\qquad$ **else** $\{\ \mathit{width}(r) = \mathit{x\_over\_n}(\mathit{width}(s), \mathit{cur\_val});$
$\qquad\quad \mathit{stretch}(r) = \mathit{x\_over\_n}(\mathit{stretch}(s), \mathit{cur\_val});$
$\qquad\quad \mathit{shrink}(r) = \mathit{x\_over\_n}(\mathit{shrink}(s), \mathit{cur\_val});$
$\qquad \}$
$\qquad \mathit{cur\_val} = r;$
$\quad \}$
$\}$

This code is used in section 1235.

1240. 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 1216 ⟩ +≡
```
case set_box:
  { scan_register_num();
    if (global) n = global_box_flag + cur_val; else n = box_flag + cur_val;
    scan_optional_equals();
    if (set_box_allowed) scan_box(n);
    else { 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 ( ) ;
    }
  } break;
```

1241. 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 1216 ⟩ +≡
```
case set_aux: alter_aux(); break;
case set_prev_graf: alter_prev_graf(); break;
case set_page_dimen: alter_page_so_far(); break;
case set_page_int: alter_integer(); break;
case set_box_dimen: alter_box_dimen(); break;
```

1242. ⟨ Declare subprocedures for *prefixed_command* 1214 ⟩ +≡
```
static void alter_aux(void)
{ halfword c;    /* hmode or vmode */
  if (cur_chr ≠ abs(mode)) report_illegal_case();
  else { c = cur_chr;
    scan_optional_equals();
    if (c ≡ vmode) { scan_normal_dimen;
      prev_depth = cur_val;
    }
    else { scan_int();
      if ((cur_val ≤ 0) ∨ (cur_val > 32767)) { print_err("Bad␣space␣factor");
        help1("I␣allow␣only␣values␣in␣the␣range␣1..32767␣here.");
        int_error(cur_val);
      }
      else space_factor = cur_val;
    }
  }
}
```

1243. ⟨ Declare subprocedures for *prefixed_command* 1214 ⟩ +≡

```
static void alter_prev_graf(void)
{ int p;   /* index into nest */
  nest[nest_ptr] = cur_list;
  p = nest_ptr;
  while (abs(nest[p].mode_field) ≠ vmode) decr(p);
  scan_optional_equals();
  scan_int();
  if (cur_val < 0) { print_err("Bad␣");
    print_esc("prevgraf");
    help1("I␣allow␣only␣nonnegative␣values␣here.");
    int_error(cur_val);
  }
  else { nest[p].pg_field = cur_val;
    cur_list = nest[nest_ptr];
  }
}
```

1244. ⟨ Declare subprocedures for *prefixed_command* 1214 ⟩ +≡

```
static void alter_page_so_far(void)
{ int c;   /* index into page_so_far */
  c = cur_chr;
  scan_optional_equals();
  scan_normal_dimen;
  page_so_far[c] = cur_val;
}
```

1245. ⟨ Declare subprocedures for *prefixed_command* 1214 ⟩ +≡

```
static void alter_integer(void){ small_number c;
      /* 0 for \deadcycles, 1 for \insertpenalties, etc. */
    c = cur_chr;
    scan_optional_equals();
    scan_int(); if (c ≡ 0) dead_cycles = cur_val
    ⟨ Cases for alter_integer 1426 ⟩
    else insert_penalties = cur_val;
    }
```

1246. ⟨ Declare subprocedures for *prefixed_command* 1214 ⟩ +≡

```
static void alter_box_dimen(void)
{ small_number c;   /* width_offset or height_offset or depth_offset */
  pointer b;   /* box register */
  c = cur_chr;
  scan_register_num();
  fetch_box(b);
  scan_optional_equals();
  scan_normal_dimen;
  if (b ≠ null) mem[b + c].sc = cur_val;
}
```

1247. Paragraph shapes are set up in the obvious way.

⟨ Assignments 1216 ⟩ +≡
```
case set_shape: { q = cur_chr;
  scan_optional_equals();
  scan_int();
  n = cur_val;
  if (n ≤ 0) p = null;
  else if (q > par_shape_loc) { n = (cur_val/2) + 1;
    p = get_node(2 * n + 1);
    info(p) = n;
    n = cur_val;
    mem[p + 1].i = n;    /* number of penalties */
    for (j = p + 2; j ≤ p + n + 1; j++) { scan_int();
      mem[j].i = cur_val;    /* penalty values */
    }
    if (¬odd(n)) mem[p + n + 2].i = 0;    /* unused */
  }
  else { p = get_node(2 * n + 1);
    info(p) = n;
    for (j = 1; j ≤ n; j++) { scan_normal_dimen;
      mem[p + 2 * j − 1].sc = cur_val;    /* indentation */
      scan_normal_dimen;
      mem[p + 2 * j].sc = cur_val;    /* width */
    }
  }
  define(q, shape_ref, p);
} break;
```

1248. 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) bad = 41;
```

1249. New hyphenation data is loaded by the *hyph_data* command.

⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
```
primitive("hyphenation", hyph_data, 0);
primitive("patterns", hyph_data, 1);
```

1250. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
```
case hyph_data:
  if (chr_code ≡ 1) print_esc("patterns");
  else print_esc("hyphenation"); break;
```

1251. ⟨ Assignments 1216 ⟩ +≡
case *hyph_data*:
  if (*cur_chr* ≡ 1) {
#**ifdef** INIT
    new_patterns( );
    goto *done*;
#**endif**
    print_err("Patterns␣can␣be␣loaded␣only␣by␣INITEX");
    help0;
    error ( ) ;
    do { *get_token*( );
    } **while** (¬(*cur_cmd* ≡ *right_brace*)); /∗ flush the patterns ∗/
    return;
  }
  else { *new_hyph_exceptions*( );
    goto *done*;
  } **break**;

1252. 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 1216 ⟩ +≡
case *assign_font_dimen*:
  { *find_font_dimen*(*true*);
    k = *cur_val*;
    scan_optional_equals( );
    scan_normal_dimen;
    font_info[*k*].*sc* = *cur_val*;
  } **break**;
case *assign_font_int*:
  { *n* = *cur_chr*;
    scan_font_ident( );
    f = *cur_val*;
    scan_optional_equals( );
    scan_int( );
    if (*n* ≡ 0) *hyphen_char*[*f*] = *cur_val*; **else** *skew_char*[*f*] = *cur_val*;
  } **break**;

1253. ⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
  primitive("hyphenchar", *assign_font_int*, 0);
  primitive("skewchar", *assign_font_int*, 1);

1254. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *assign_font_int*:
  if (*chr_code* ≡ 0) *print_esc*("hyphenchar");
  else *print_esc*("skewchar"); **break**;

1255. Here is where the information for a new font gets loaded.

⟨ Assignments 1216 ⟩ +≡
case *def_font*: *new_font*(*a*); **break**;

1256. ⟨ Declare subprocedures for *prefixed_command* 1214 ⟩ +≡

```
static void new_font(small_number a){ pointer u;   /* user's font identifier */
    scaled s;   /* stated "at" size, or negative of scaled magnification */
    int f;   /* runs through existing fonts */
    str_number t;   /* name for the frozen font identifier */
    int old_setting;   /* holds selector setting */
    str_number flushable_string;   /* string not yet referenced */
    if (job_name ≡ 0) open_log_file( );   /* avoid confusing texput with the font name */
    get_r_token( );
    u = cur_cs;
    if (u ≥ hash_base) t = text(u);
    else if (u ≥ single_base)
      if (u ≡ null_cs) t = s_no("FONT"); else t = u − single_base;
    else { old_setting = selector;
      selector = new_string;
      print("FONT");
      printn(u − active_base);
      selector = old_setting;
      str_room(1);
      t = make_string( );
    }
    define(u, set_font, null_font);
    scan_optional_equals( );
    scan_file_name( );
    ⟨ Scan the font size specification 1257 ⟩;
    ⟨ If this font has already been loaded, set f to the internal font number and goto
        common_ending 1259 ⟩;
    f = read_font_info(u, cur_name, cur_area, s); common_ending: define(u, set_font, f);
    eqtb[font_id_base + f] = eqtb[u];
    font_id_text(f) = t;
    }
```

1257. ⟨ Scan the font size specification 1257 ⟩ ≡

```
  name_in_progress = true;   /* this keeps cur_name from being changed */
  if (scan_keyword("at")) ⟨ Put the (positive) 'at' size into s 1258 ⟩
  else if (scan_keyword("scaled")) { scan_int( );
    s = −cur_val;
    if ((cur_val ≤ 0) ∨ (cur_val > 32768)) {
      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;
    }
  }
  else s = −1000;
  name_in_progress = false
```

This code is used in section 1256.

1258. ⟨Put the (positive) 'at' size into s 1258⟩ ≡

```
{ scan_normal_dimen;
  s = cur_val;
  if ((s ≤ 0) ∨ (s ≥ °1000000000)) { 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;
  }
}
```

This code is used in section 1257.

1259. 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* 1259⟩ ≡

```
flushable_string = str_ptr − 1;
for (f = font_base + 1; f ≤ font_ptr; f++)
  if (str_eq_str(font_name[f], cur_name) ∧ str_eq_str(font_area[f], cur_area)) {
      if (cur_name ≡ flushable_string) { flush_string;
      cur_name = font_name[f];
    }
    if (s > 0) { if (s ≡ font_size[f]) goto common_ending;
    }
    else if (font_size[f] ≡ xn_over_d(font_dsize[f], −s, 1000)) goto common_ending;
  }
```

This code is used in section 1256.

1260. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 226⟩ +≡

```
case set_font:
  { print("select␣font␣");
    slow_print(font_name[chr_code]);
    if (font_size[chr_code] ≠ font_dsize[chr_code]) { print("␣at␣");
      print_scaled(font_size[chr_code]);
      print("pt");
    }
  } break;
```

1261. ⟨Put each of TEX's primitives into the hash table 225⟩ +≡

```
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);
```

1262. ⟨Cases of *print_cmd_chr* for symbolic printing of primitives 226⟩ +≡
case *set_interaction*:
  switch (*chr_code*) {
  case *batch_mode*: *print_esc*("batchmode"); **break**;
  case *nonstop_mode*: *print_esc*("nonstopmode"); **break**;
  case *scroll_mode*: *print_esc*("scrollmode"); **break**;
  default: *print_esc*("errorstopmode");
  } **break**;

1263. ⟨Assignments 1216⟩ +≡
case *set_interaction*: *new_interaction*( ); **break**;

1264. ⟨Declare subprocedures for *prefixed_command* 1214⟩ +≡
  static void *new_interaction*(**void**)
  { *print_ln*( );
    interaction = *cur_chr*;
    ⟨Initialize the print *selector* based on *interaction* 74⟩;
    if (*log_opened*) *selector* = *selector* + 2;
  }

1265. 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⟩ +≡
  static halfword *after_token*;   /∗ zero, or a saved token ∗/

1266. ⟨Set initial values of key variables 21⟩ +≡
  after_token = 0;

1267. ⟨Cases of *main_control* that don't depend on *mode* 1209⟩ +≡
any_mode(*after_assignment*):
  { *get_token*( );
    after_token = *cur_tok*;
  } **break**;

1268. ⟨Insert a token saved by \afterassignment, if any 1268⟩ ≡
  if (*after_token* ≠ 0) { *cur_tok* = *after_token*;
    back_input( );
    after_token = 0;
  }

This code is used in section 1210.

1269. 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* 1042⟩ +≡
  static void *do_assignments*(**void**)
  { **loop** { ⟨Get the next non-blank non-relax non-call token 403⟩;
      if (*cur_cmd* ≤ *max_non_prefixed_command*) **return**;
      set_box_allowed = *false*;
      prefixed_command( );
      set_box_allowed = *true*;
    }
  }

1270. ⟨ Cases of *main_control* that don't depend on *mode* 1209 ⟩ +≡

```
any_mode(after_group):
  { get_token();
    save_for_after(cur_tok);
  } break;
```

1271. Files for `\read` are opened and closed by the *in_stream* command.

⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡

```
  primitive("openin", in_stream, 1);
  primitive("closein", in_stream, 0);
```

1272. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case in_stream:
  if (chr_code ≡ 0) print_esc("closein");
  else print_esc("openin"); break;
```

1273. ⟨ Cases of *main_control* that don't depend on *mode* 1209 ⟩ +≡

```
any_mode(in_stream): open_or_close_in(); break;
```

1274. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
  static void open_or_close_in(void)
  { int c;      /* 1 for \openin, 0 for \closein */
    int n;      /* stream number */
    c = cur_chr;
    scan_four_bit_int();
    n = cur_val;
    if (read_open[n] ≠ closed) { a_close(&read_file[n]);
      read_open[n] = closed;
    }
    if (c ≠ 0) { scan_optional_equals();
      scan_file_name();
      pack_cur_name(".tex");
      if (a_open_in(&read_file[n])) { read_open[n] = just_open;
        ⟨ Set new read_file_num[n] 1745 ⟩
      }
    }
  }
```

1275. The user can issue messages to the terminal, regardless of the current mode.

⟨ Cases of *main_control* that don't depend on *mode* 1209 ⟩ +≡

```
any_mode(message): issue_message(); break;
```

1276. ⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡

```
  primitive("message", message, 0);
  primitive("errmessage", message, 1);
```

1277. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case message:
  if (chr_code ≡ 0) print_esc("message");
  else print_esc("errmessage"); break;
```

1278. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void issue_message(void)
{ int old_setting;   /* holds selector setting */
  int c;   /* identifies \message and \errmessage */
  str_number s;   /* the message */
  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) ⟨ Print string s on the terminal 1279 ⟩
  else ⟨ Print string s as an error message 1282 ⟩;
  flush_string;
}
```

1279. ⟨ Print string *s* on the terminal 1279 ⟩ ≡

```
{ if (term_offset + length(s) > max_print_line − 2) print_ln();
  else if ((term_offset > 0) ∨ (file_offset > 0)) print_char('␣');
  slow_print(s);
  update_terminal;
}
```

This code is used in section 1278.

1280. 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 ⟩ +≡

```
static bool long_help_seen;   /* has the long \errmessage help been used? */
```

1281. ⟨ Set initial values of key variables 21 ⟩ +≡

```
long_help_seen = false;
```

1282. ⟨ Print string *s* as an error message 1282 ⟩ ≡

```
{ print_err("");
  slow_print(s);
  if (err_help ≠ null) use_err_help = true;
  else if (long_help_seen) help1("(That␣was␣another␣\\errmessage.)")
  else { if (interaction < error_stop_mode) 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.");
  }
  error();
  use_err_help = false;
}
```

This code is used in section 1278.

1283. The **error** routine calls on *give_err_help* if help is requested from the *err_help* parameter.

```
static void give_err_help(void)
{ token_show(err_help);
}
```

1284. 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* 1209 ⟩ +≡
any_mode(*case_shift*): *shift_case*( ); **break**;

1285. ⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡

```
primitive("lowercase", case_shift, lc_code_base);
primitive("uppercase", case_shift, uc_code_base);
```

1286. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case case_shift:
  if (chr_code ≡ lc_code_base) print_esc("lowercase");
  else print_esc("uppercase"); break;
```

1287. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void shift_case(void)
{ pointer b;    /* lc_code_base or uc_code_base */
  pointer p;    /* runs through the token list */
  halfword t;   /* token */
  eight_bits c;   /* character code */
  b = cur_chr;
  p = scan_toks(false, false);
  p = link(def_ref);
  while (p ≠ null) { ⟨ Change the case of the token in p, if a change is appropriate 1288 ⟩;
    p = link(p);
  }
  back_list(link(def_ref));
  free_avail(def_ref);    /* omit reference count */
}
```

1288. 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 1288 ⟩ ≡

```
t = info(p);
if (t < cs_token_flag + single_base) { c = t % 256;
  if (equiv(b + c) ≠ 0) info(p) = t − c + equiv(b + c);
}
```

This code is used in section 1287.

1289. 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* 1209 ⟩ +≡
any_mode(*xray*): *show_whatever*( ); **break**;

1290. **#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 225 ⟩ +≡
primitive("show", *xray*, *show_code*);
primitive("showbox", *xray*, *show_box_code*);
primitive("showthe", *xray*, *show_the_code*);
primitive("showlists", *xray*, *show_lists_code*);

1291. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
case *xray*:
switch (*chr_code*) {
case *show_box_code*: *print_esc*("showbox"); **break**;
case *show_the_code*: *print_esc*("showthe"); **break**;
case *show_lists_code*: *print_esc*("showlists"); **break**;
⟨ Cases of *xray* for *print_cmd_chr* 1406 ⟩
default: *print_esc*("show");
} **break**;

1292. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡

```
static void show_whatever(void)
{ pointer p;  /* tail of a token list to show */
  small_number t;  /* type of conditional being shown */
  int m;  /* upper bound on fi_or_else codes */
  int l;  /* line where that conditional began */
  int n;  /* level of \if...\fi nesting */
  switch (cur_chr) {
  case show_lists_code:
    { begin_diagnostic();
      show_activities();
    } break;
  case show_box_code: ⟨ Show the current contents of a box 1295 ⟩ break;
  case show_code: ⟨ Show the current meaning of a token, then goto common_ending 1293 ⟩
    ⟨ Cases for show_whatever 1407 ⟩
  default: ⟨ Show the current value of some parameter or register, then goto common_ending 1296 ⟩
  }
  ⟨ Complete a potentially long \show command 1297 ⟩;
common_ending:
  if (interaction < error_stop_mode) { help0;
    decr(error_count);
  }
  else if (tracing_online > 0) {
    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).");
  }
  else {
    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.");
  }
  error();
}
```

1293. ⟨ Show the current meaning of a token, then **goto** *common_ending* 1293 ⟩ ≡

```
{ get_token();
  if (interaction ≡ error_stop_mode) wake_up_terminal;
  print_nl(">␣");
  if (cur_cs ≠ 0) { sprint_cs(cur_cs);
    print_char('=');
  }
  print_meaning();
  goto common_ending;
}
```

This code is used in section 1292.

1294. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡

```
case undefined_cs: print("undefined"); break;
case call: case long_call: case outer_call: case long_outer_call:
  { n = cmd − call;
    if (info(link(chr_code)) ≡ protected_token) n = n + 4;
    if (odd(n/4)) print_esc("protected");
    if (odd(n)) print_esc("long");
    if (odd(n/2)) print_esc("outer");
    if (n > 0) print_char('␣');
    print("macro");
  } break;
case end_template: print_esc("outer␣endtemplate"); break;
```

1295. ⟨ Show the current contents of a box 1295 ⟩ ≡

```
  { scan_register_num();
    fetch_box(p);
    begin_diagnostic();
    print_nl(">␣\\box");
    print_int(cur_val);
    print_char('=');
    if (p ≡ null) print("void"); else show_box(p);
  }
```

This code is used in section 1292.

1296. ⟨ Show the current value of some parameter or register, then **goto** *common_ending* 1296 ⟩ ≡

```
  { the_toks();
    if (interaction ≡ error_stop_mode) wake_up_terminal;
    print_nl(">␣");
    token_show(temp_head);
    flush_list(link(temp_head));
    goto common_ending;
  }
```

This code is used in section 1292.

1297. ⟨ Complete a potentially long `\show` command 1297 ⟩ ≡

```
  end_diagnostic(true);
  print_err("OK");
  if (selector ≡ term_and_log)
    if (tracing_online ≤ 0) { selector = term_only;
      print("␣(see␣the␣transcript␣file)");
      selector = term_and_log;
    }
```

This code is used in section 1292.

1298. 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⟩ +≡
 static str_number *format_ident*, *frozen_format_ident*;

1299. ⟨Set initial values of key variables 21⟩ +≡
 format_ident = *frozen_format_ident* = 0;

1300. We keep a copy of the initial value, be able to test for it later.

⟨Initialize table entries (done by `INITEX` only) 163⟩ +≡
 format_ident = *frozen_format_ident* = *s_no*("␣(INITEX)");

1301. ⟨Declare action procedures for use by *main_control* 1042⟩ +≡

```
#ifdef INIT
  static void store_fmt_file(void)
  { int j, k, l;      /* all-purpose indices */
    int p, q;      /* all-purpose pointers */
    int x;      /* something to dump */
    four_quarters w;      /* four ASCII codes */
    ⟨If dumping is not allowed, abort 1303⟩;
    ⟨Create the format_ident, open the format file, and inform the user that dumping has begun 1327⟩;
    ⟨Dump constants for consistency check 1306⟩;
    ⟨Dump the string pool 1308⟩;
    ⟨Dump the dynamic memory 1310⟩;
    ⟨Dump the table of equivalents 1312⟩;
    ⟨Dump the font information 1319⟩;
    ⟨Dump the hyphenation tables 1323⟩;
    ⟨Dump a couple more things and the closing check word 1325⟩;
    ⟨Close the format file 1328⟩;
  }
#endif
```

1302. 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 too_small(X)
        { wake_up_terminal;
          wterm_ln("---!␣Must␣increase␣the␣%s", X);
          goto bad_fmt;
        }
⟨Declare the function called open_fmt_file 523⟩
  static bool load_fmt_file(void)
  { int j, k;    /* all-purpose indices */
    int p, q;    /* all-purpose pointers */
    int x;    /* something undumped */
    four_quarters w;    /* four ASCII codes */
    ⟨Undump constants for consistency check 1307⟩;
    ⟨Undump the string pool 1309⟩;
    ⟨Undump the dynamic memory 1311⟩;
    ⟨Undump the table of equivalents 1313⟩;
    ⟨Undump the font information 1320⟩;
    ⟨Undump the hyphenation tables 1324⟩;
    ⟨Undump a couple more things and the closing check word 1326⟩;
    return true;    /* it worked! */
  bad_fmt: wake_up_terminal;
    wterm_ln("(Fatal␣format␣file␣error;␣I'm␣stymied)");
    return false;
  }
```

1303. 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 1303⟩ ≡
  if (save_ptr ≠ 0) { print_err("You␣can't␣dump␣inside␣a␣group");
    help1("`{...\\dump}'␣is␣a␣no-no.");
    succumb;
  }
```

This code is used in section 1301.

1304. Format files consist of **memory_word** items, and we use the following macros to dump words of different types:

```
#define dump_wd(A)
        { fmt_file.d = A;
          put(fmt_file); }
#define dump_int(A)
        { fmt_file.d.i = A;
          put(fmt_file); }
#define dump_hh(A)
        { fmt_file.d.hh = A;
          put(fmt_file); }
#define dump_qqqq(A)
        { fmt_file.d.qqqq = A;
          put(fmt_file); }
```

⟨ Global variables 13 ⟩ +≡
  static word_file *fmt_file*;   /∗ for input or output of format information ∗/

1305. 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(A)
        { get(fmt_file);
          A = fmt_file.d; }
#define undump_int(A)
        { get(fmt_file);
          A = fmt_file.d.i; }
#define undump_hh(A)
        { get(fmt_file);
          A = fmt_file.d.hh; }
#define undump_qqqq(A)
        { get(fmt_file);
          A = fmt_file.d.qqqq; }
#define undump(A, B, C)
        { undump_int(x);
          if ((x < A) ∨ (x > B)) goto bad_fmt; else C = x; }
#define undump_size(A, B, C, D)
        { undump_int(x);
          if (x < A) goto bad_fmt;
          if (x > B) too_small(C) else D = x; }
```

1306. The next few sections of the program should make it clear how we use the dump/undump macros.

⟨ Dump constants for consistency check 1306 ⟩ ≡
dump_int(0);
⟨ Dump the ε-TEX state 1384 ⟩
⟨ Dump the PRoTE state 1543 ⟩
⟨ Dump the `ROM` array 1584 ⟩
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 1301.

1307. 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 1307 ⟩ ≡
$x = fmt\_file.d.i$;
if ($x \neq 0$) **goto** *bad_fmt*; /∗ check that strings are the same ∗/
⟨ Undump the ε-TEX state 1385 ⟩
⟨ Undump the PRoTE state 1544 ⟩
⟨ Undump the `ROM` array 1585 ⟩
undump_int(x);
if ($x \neq mem\_bot$) **goto** *bad_fmt*;
undump_int(x);
if ($x \neq mem\_top$) **goto** *bad_fmt*;
undump_int(x);
if ($x \neq eqtb\_size$) **goto** *bad_fmt*;
undump_int(x);
if ($x \neq hash\_prime$) **goto** *bad_fmt*;
undump_int(x); **if** ($x \neq hyph\_size$) **goto** *bad_fmt*

This code is used in section 1302.

1308. **#define** *dump_four_ASCII* $w.b0 = qi(so(str\_pool[k]))$;
$w.b1 = qi(so(str\_pool[k+1]))$;
$w.b2 = qi(so(str\_pool[k+2]))$;
$w.b3 = qi(so(str\_pool[k+3]))$; *dump_qqqq*($w$)

⟨ Dump the string pool 1308 ⟩ ≡
dump_int(*pool_ptr*);
dump_int(*str_ptr*);
for ($k = 0$; $k \leq str\_ptr$; $k{+}{+}$) *dump_int*(*str_start*[k]);
$k = 0$;
while ($k + 4 < pool\_ptr$) { *dump_four_ASCII*;
$k = k + 4$;
}
$k = pool\_ptr - 4$;
dump_four_ASCII;
print_ln( );
print_int(*str_ptr*);
print("␣strings␣of␣total␣length␣"); *print_int*(*pool_ptr*)

This code is used in section 1301.

1309. **#define** *undump_four_ASCII* $\;$ *undump_qqqq*(w);
$\quad$ *str_pool*$[k]$ = *si*(*qo*($w.b0$));
$\quad$ *str_pool*$[k+1]$ = *si*(*qo*($w.b1$));
$\quad$ *str_pool*$[k+2]$ = *si*(*qo*($w.b2$)); *str_pool*$[k+3]$ = *si*(*qo*($w.b3$))

⟨ Undump the string pool 1309 ⟩ ≡
undump_size(0, *pool_size*, "string␣pool␣size", *pool_ptr*);
undump_size(0, *max_strings*, "max␣strings", *str_ptr*);
for ($k = 0$; $k \le$ *str_ptr*; k++) *undump*(0, *pool_ptr*, *str_start*$[k]$);
$k = 0$;
while ($k + 4 <$ *pool_ptr*) { *undump_four_ASCII*;
$\quad k = k + 4$;
}
$k =$ *pool_ptr* $- 4$;
undump_four_ASCII;
init_str_ptr = *str_ptr*; *init_pool_ptr* = *pool_ptr*

This code is used in section 1302.

1310. 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 1310 ⟩ ≡

```
sort_avail();
var_used = 0;
dump_int(lo_mem_max);
dump_int(rover);
if (eTeX_ex)
  for (k = int_val; k ≤ tok_val; k++) dump_int(sa_root[k]);
p = mem_bot;
q = rover;
x = 0;
do {
  for (k = p; k ≤ q + 1; k++) dump_wd(mem[k]);
  x = x + q + 2 − p;
  var_used = var_used + q − p;
  p = q + node_size(q);
  q = rlink(q);
} while (¬(q ≡ rover));
var_used = var_used + lo_mem_max − p;
dyn_used = mem_end + 1 − hi_mem_min;
for (k = p; k ≤ lo_mem_max; k++) dump_wd(mem[k]);
x = x + lo_mem_max + 1 − p;
dump_int(hi_mem_min);
dump_int(avail);
for (k = hi_mem_min; k ≤ mem_end; k++) dump_wd(mem[k]);
x = x + mem_end + 1 − hi_mem_min;
p = avail;
while (p ≠ null) { decr(dyn_used);
  p = link(p);
}
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 1301.

1311. ⟨ Undump the dynamic memory 1311 ⟩ ≡

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

This code is used in section 1302.

1312. ⟨ Dump the table of equivalents 1312 ⟩ ≡

```
⟨ Dump regions 1 to 4 of eqtb 1314 ⟩;
⟨ Dump regions 5 and 6 of eqtb 1315 ⟩;
dump_int(par_loc);
dump_int(write_loc);
dump_int(input_loc);
⟨ Dump the hash table 1317 ⟩
```

This code is used in section 1301.

1313. ⟨ Undump the table of equivalents 1313 ⟩ ≡

```
⟨ Undump regions 1 to 6 of eqtb 1316 ⟩;
undump(hash_base, frozen_control_sequence, par_loc);
par_token = cs_token_flag + par_loc;
undump(hash_base, frozen_control_sequence, write_loc);
undump(hash_base, frozen_control_sequence, input_loc);
input_token = cs_token_flag + input_loc;
⟨ Undump the hash table 1318 ⟩
```

This code is used in section 1302.

1314. 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* 1314⟩ ≡
  $k = active\_base$;
  do { $j = k$;
    while $(j < int\_base - 1)$ { **if** $((equiv(j) \equiv equiv(j+1)) \wedge (eq\_type(j) \equiv eq\_type(j+1)) \wedge (eq\_level(j) \equiv eq\_level(j+1)))$ **goto** *found1*;
      $incr(j)$;
    }
    $l = int\_base$;
    goto *done1*; /* $j \equiv int\_base - 1$ */
  found1: $incr(j)$;
    $l = j$;
    while $(j < int\_base - 1)$ { **if** $((equiv(j) \neq equiv(j+1)) \vee (eq\_type(j) \neq eq\_type(j+1)) \vee (eq\_level(j) \neq eq\_level(j+1)))$ **goto** *done1*;
      $incr(j)$;
    }
  done1: $dump\_int(l - k)$;
    while $(k < l)$ { $dump\_wd(eqtb[k])$;
      $incr(k)$;
    }
    $k = j + 1$;
    $dump\_int(k - l)$;
  } **while** $(\neg(k \equiv int\_base))$

This code is used in section 1312.

1315. ⟨Dump regions 5 and 6 of *eqtb* 1315⟩ ≡
  do { $j = k$;
    while $(j < eqtb\_size)$ { **if** $(eqtb[j].i \equiv eqtb[j+1].i)$ **goto** *found2*;
      $incr(j)$;
    }
    $l = eqtb\_size + 1$;
    goto *done2*; /* $j \equiv eqtb\_size$ */
  found2: $incr(j)$;
    $l = j$;
    while $(j < eqtb\_size)$ { **if** $(eqtb[j].i \neq eqtb[j+1].i)$ **goto** *done2*;
      $incr(j)$;
    }
  done2: $dump\_int(l - k)$;
    while $(k < l)$ { $dump\_wd(eqtb[k])$;
      $incr(k)$;
    }
    $k = j + 1$;
    $dump\_int(k - l)$;
  } **while** $(\neg(k > eqtb\_size))$

This code is used in section 1312.

1316. ⟨ Undump regions 1 to 6 of *eqtb* 1316 ⟩ ≡

```
k = active_base;
do { undump_int(x);
  if ((x < 1) ∨ (k + x > eqtb_size + 1)) goto bad_fmt;
  for (j = k; j ≤ k + x − 1; j++) undump_wd(eqtb[j]);
  k = k + x;
  undump_int(x);
  if ((x < 0) ∨ (k + x > eqtb_size + 1)) goto bad_fmt;
  for (j = k; j ≤ k + x − 1; j++) eqtb[j] = eqtb[k − 1];
  k = k + x;
} while (¬(k > eqtb_size))
```

This code is used in section 1313.

1317. A different scheme is used to compress the hash table, since its lower region is usually sparse. When $text(p) \neq 0$ for $p \le hash\_used$, we output two words, p and $hash[p]$. The hash table is, of course, densely packed for $p \ge hash\_used$, so the remaining entries are output in a block.

⟨ Dump the hash table 1317 ⟩ ≡

```
dump_int(hash_used);
cs_count = frozen_control_sequence − 1 − hash_used;
for (p = hash_base; p ≤ hash_used; p++)
  if (text(p) ≠ 0) { dump_int(p);
    dump_hh(hash[p]);
    incr(cs_count);
  }
for (p = hash_used + 1; p ≤ undefined_control_sequence − 1; p++) dump_hh(hash[p]);
dump_int(cs_count);
print_ln();
print_int(cs_count); print("␣multiletter␣control␣sequences")
```

This code is used in section 1312.

1318. ⟨ Undump the hash table 1318 ⟩ ≡

```
undump(hash_base, frozen_control_sequence, hash_used);
p = hash_base − 1;
do { undump(p + 1, hash_used, p);
  undump_hh(hash[p]);
} while (¬(p ≡ hash_used));
for (p = hash_used + 1; p ≤ undefined_control_sequence − 1; p++) undump_hh(hash[p]);
undump_int(cs_count)
```

This code is used in section 1313.

1319. ⟨ Dump the font information 1319 ⟩ ≡

```
dump_int(fmem_ptr);
for (k = 0; k ≤ fmem_ptr − 1; k++) dump_wd(font_info[k]);
dump_int(font_ptr);
for (k = null_font; k ≤ font_ptr; k++) ⟨ Dump the array info for internal font number k 1321 ⟩;
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 ≠ font_base + 1) print_char('s')
```

This code is used in section 1301.

1320. ⟨ Undump the font information 1320 ⟩ ≡

```
undump_size(7, font_mem_size, "font␣mem␣size", fmem_ptr);
for (k = 0; k ≤ fmem_ptr − 1; k++) undump_wd(font_info[k]);
undump_size(font_base, font_max, "font␣max", font_ptr);
for (k = null_font; k ≤ font_ptr; k++) ⟨ Undump the array info for internal font number k 1322 ⟩
```

This code is used in section 1302.

1321. ⟨ Dump the array info for internal font number k 1321 ⟩ ≡

```
{ 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");
  printn_esc(font_id_text(k));
  print_char('=');
  print_file_name(font_name[k], font_area[k], empty_string);
  if (font_size[k] ≠ font_dsize[k]) { print("␣at␣");
    print_scaled(font_size[k]);
    print("pt");
  }
}
```

This code is used in section 1319.

1322. ⟨Undump the array info for internal font number k 1322⟩ ≡

{ *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]);
}

This code is used in section 1320.

1323. ⟨ Dump the hyphenation tables 1323 ⟩ ≡

```
dump_int(hyph_count);
for (k = 0; k ≤ hyph_size; k++)
  if (hyph_word[k] ≠ 0) { dump_int(k);
    dump_int(hyph_word[k]);
    dump_int(hyph_list[k]);
  }
print_ln();
print_int(hyph_count);
print("␣hyphenation␣exception");
if (hyph_count ≠ 1) print_char('s');
if (trie_not_ready) init_trie();
dump_int(trie_max);
dump_int(hyph_start);
for (k = 0; k ≤ trie_max; k++) dump_hh(trie[k]);
dump_int(trie_op_ptr);
for (k = 1; k ≤ trie_op_ptr; k++) { dump_int(hyf_distance[k]);
  dump_int(hyf_num[k]);
  dump_int(hyf_next[k]);
}
print_nl("Hyphenation␣trie␣of␣length␣");
print_int(trie_max);
print("␣has␣");
print_int(trie_op_ptr);
print("␣op");
if (trie_op_ptr ≠ 1) print_char('s');
print("␣out␣of␣");
print_int(trie_op_size);
for (k = 255; k ≥ 0; k−−)
  if (trie_used[k] > min_quarterword) { print_nl("␣␣");
    print_int(qo(trie_used[k]));
    print("␣for␣language␣");
    print_int(k);
    dump_int(k);
    dump_int(qo(trie_used[k]));
  }
```

This code is used in section 1301.

1324. Only "nonempty" parts of *op_start* need to be restored.

⟨ Undump the hyphenation tables 1324 ⟩ ≡

```
  undump(0, hyph_size, hyph_count);
  for (k = 1; k ≤ hyph_count; k++) { undump(0, hyph_size, j);
    undump(0, str_ptr, hyph_word[j]);
    undump(min_halfword, max_halfword, hyph_list[j]);
  }
  undump_size(0, trie_size, "trie␣size", j);
#ifdef INIT
  trie_max = j;
#endif
  undump(0, j, hyph_start);
  for (k = 0; k ≤ j; k++) undump_hh(trie[k]);
  undump_size(0, trie_op_size, "trie␣op␣size", j);
#ifdef INIT
  trie_op_ptr = j;
#endif
  for (k = 1; k ≤ j; k++) { undump(0, 63, hyf_distance[k]);    /* a small_number */
    undump(0, 63, hyf_num[k]);
    undump(min_quarterword, max_quarterword, hyf_next[k]);
  }
#ifdef INIT
  for (k = 0; k ≤ 255; k++) trie_used[k] = min_quarterword;
#endif
  k = 256;
  while (j > 0) { undump(0, k − 1, k);
    undump(1, j, x);
#ifdef INIT
    trie_used[k] = qi(x);
#endif
    j = j − x;
    op_start[k] = qo(j);
  }
#ifdef INIT
  trie_not_ready = false
#endif
```

This code is used in section 1302.

1325. 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 1325 ⟩ ≡

```
  dump_int(interaction);
  dump_int(format_ident);
  dump_int(69069); tracing_stats = 0
```

This code is used in section 1301.

1326. ⟨Undump a couple more things and the closing check word 1326⟩ ≡
undump(*batch_mode*, *error_stop_mode*, *interaction*);
if (*interaction_option* ≥ 0) *interaction* = *interaction_option*; /* TEX Live */
undump(0, *str_ptr*, *format_ident*);
undump_int(x); **if** (($x \ne 69069$) $\lor$ *eof*(*fmt_file*)) **goto** *bad_fmt*

This code is used in section 1302.

1327. ⟨Create the *format_ident*, open the format file, and inform the user that dumping has begun 1327⟩ ≡
selector = *new_string*;
print("␣(preloaded␣format=");
printn(*job_name*);
print_char('␣');
print_int(*year*);
print_char('.');
print_int(*month*);
print_char('.');
print_int(*day*);
print_char(')');
if (*interaction* ≡ *batch_mode*) *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*)) *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 1301.

1328. ⟨Close the format file 1328⟩ ≡
w_close(&*fmt_file*)

This code is used in section 1301.

1329. 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 1332 ⟩

1330. We have noted that there are two versions of TEX82. One, called `INITEX`, has to be run first; it initializes everything from scratch, without reading a format file, and it has the capability of dumping a format file. The other one is called '`VIRTEX`'; it is a "virgin" program that needs to input a format file in order to get started. `VIRTEX` typically has more memory capacity than `INITEX`, because it does not need the space consumed by the auxiliary hyphenation tables and the numerous calls on *primitive*, etc.

The `VIRTEX` program cannot read a format file instantaneously, of course; the best implementations therefore allow for production versions of TEX that not only avoid the loading routine for Pascal object code, they also have a format file pre-loaded. This is impossible to do if we stick to standard Pascal; but there is a simple way to fool many systems into avoiding the initialization, as follows: (1) We declare a global integer variable called *ready_already*. The probability is negligible that this variable holds any particular value like 314159 when `VIRTEX` is first loaded. (2) After we have read in a format file and initialized everything, we set $ready\_already = 314159$. (3) Soon `VIRTEX` will print '`*`', waiting for more input; and at this point we interrupt the program and save its core image in some form that the operating system can reload speedily. (4) When that core image is activated, the program starts again at the beginning; but now $ready\_already \equiv 314159$ 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 $ready\_already \equiv 314159$, 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 ⟩ +≡
  static int *ready_already*; /∗ a sacrifice of purity for economy ∗/

1331. 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."

```
int main(int argc, char *argv[])
{   /* start_here */
  main_init(argc, argv);   /* TEX Live */
  history = fatal_error_stop;   /* in case we quit during initialization */
  t_open_out;   /* open the terminal for output */
  if (ready_already ≡ 314159) goto start_of_TEX;
  ⟨Check the "constant" values for consistency 14⟩
  if (bad > 0) {
    wterm_ln("Ouch---my␣internal␣constants␣have␣been␣clobbered!""---case␣%d", bad);
    exit(0);
  }
  get_strings_started();
  initialize();   /* set global variables to their starting values */
#ifdef INIT
  if (iniversion)   /* TEX Live */
  { init_prim();   /* call primitive for each primitive */
    init_str_ptr = str_ptr;
    init_pool_ptr = pool_ptr;
    fix_date_and_time();
  }
#endif
  ready_already = 314159;
start_of_TEX: ⟨Initialize the output routines 54⟩;
  ⟨Get the first line of input and prepare to start 1336⟩;
  history = spotless;   /* ready to go! */
  main_control();   /* come to life */
  final_cleanup();   /* prepare for death */
  close_files_and_terminate();
  ready_already = 0;
  return 0;
}
```

1332. 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 1332⟩ ≡
  static void close_files_and_terminate(void)
  { int k;    /* all-purpose index */
    ⟨Finish the extensions 1377⟩
    new_line_char = −1;
#ifdef STAT
    ⟨Output statistics about this job 1333⟩;
#endif
    wake_up_terminal;
    ⟨Finish the DVI file 641⟩;
    if (log_opened) { wlog_cr;
      a_close(&log_file);
      selector = selector − 2;
      if (selector ≡ term_only) { print_nl("Transcript␣written␣on␣");
        slow_print(log_name);
        print_char('.');
        print_nl("");
      }
    }
  }
```

See also sections 1334, 1335, 1337, and 1545.

This code is used in section 1329.

1333. 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 1333 ⟩ ≡

```
if (log_opened) { wlog_ln("␣");
  wlog_ln("Here␣is␣how␣much␣of␣TeX's␣memory␣you␣used:");
  wlog("␣%d␣string", str_ptr − init_str_ptr);
  if (str_ptr ≠ init_str_ptr + 1) wlog("s");
  wlog_ln("␣out␣of␣%d", max_strings − init_str_ptr);
  wlog_ln("␣%d␣string␣characters␣out␣of␣%d", pool_ptr − init_pool_ptr, pool_size − init_pool_ptr);
  wlog_ln("␣%d␣words␣of␣memory␣out␣of␣%d", lo_mem_max − mem_min + mem_end − hi_mem_min + 2,
      mem_end + 1 − mem_min);
  wlog_ln("␣%d␣multiletter␣control␣sequences␣out␣of␣%d", cs_count, hash_size);
  wlog("␣%d␣words␣of␣font␣info␣for␣%d␣font", fmem_ptr, font_ptr − font_base);
  if (font_ptr ≠ font_base + 1) wlog("s");
  wlog_ln(",␣out␣of␣%d␣for␣%d", font_mem_size, font_max − font_base);
  wlog("␣%d␣hyphenation␣exception", hyph_count);
  if (hyph_count ≠ 1) wlog("s");
  wlog_ln("␣out␣of␣%d", hyph_size);
  wlog_ln("␣%di,%dn,%dp,%db,%ds␣stack␣positions␣out␣of␣%di,%dn,%dp,%db,%ds", max_in_stack,
      max_nest_stack,
      max_param_stack,
      max_buf_stack + 1,
      max_save_stack + 6,
      stack_size, nest_size, param_size, buf_size, save_size);
}
```

This code is used in section 1332.

1334. We get to the *final_cleanup* routine when `\end` or `\dump` has been scanned and *its_all_over*.

⟨Last-minute procedures 1332⟩ +≡

```
static void final_cleanup(void)
{ int c;      /* 0 for \end, 1 for \dump */
  c = cur_chr;
  if (c ≠ 1) new_line_char = −1;
  if (job_name ≡ 0) open_log_file();
  while (input_ptr > 0)
    if (state ≡ token_list) end_token_list(); else end_file_reading();
  while (open_parens > 0) { print("␣)");
    decr(open_parens);
  }
  if (cur_level > level_one) { print_nl("(");
    print_esc("end␣occurred␣");
    print("inside␣a␣group␣at␣level␣");
    print_int(cur_level − level_one);
    print_char(')');
    if (eTeX_ex) show_save_groups();
  }
  while (cond_ptr ≠ null) { print_nl("(");
    print_esc("end␣occurred␣");
    print("when␣");
    print_cmd_chr(if_test, cur_if);
    if (if_line ≠ 0) { print("␣on␣line␣");
      print_int(if_line);
    }
    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);
  }
  if (history ≠ spotless)
    if (((history ≡ warning_issued) ∨ (interaction < error_stop_mode)))
      if (selector ≡ term_and_log) { selector = term_only;
        print_nl("(see␣the␣transcript␣file␣for␣additional␣information)");
        selector = term_and_log;
      }
  if (c ≡ 1) {
#ifdef INIT
    for (c = top_mark_code; c ≤ split_bot_mark_code; c++)
      if (cur_mark[c] ≠ null) delete_token_ref(cur_mark[c]);
    if (sa_mark ≠ null)
      if (do_marks(destroy_marks, 0, sa_mark)) sa_mark = null;
    for (c = last_box_code; c ≤ vsplit_code; c++) flush_node_list(disc_ptr[c]);
    if (last_glue ≠ max_halfword) delete_glue_ref(last_glue);
    store_fmt_file();
    return;
#endif
    print_nl("(\\dump␣is␣performed␣only␣by␣INITEX)");
    return;
```

```
    }
  }
```

1335. ⟨ Last-minute procedures 1332 ⟩ +≡

```
#ifdef INIT
  static void init_prim(void)    /* initialize all the primitives */
  { no_new_control_sequence = false;
    first = 0;
    ⟨ Put each of TeX's primitives into the hash table 225 ⟩;
    no_new_control_sequence = true;
  }
#endif
```

1336. 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 1336 ⟩ ≡

```
  { ⟨ Initialize the input routines 330 ⟩;
    ⟨ Enable ε-TeX and furthermore Prote, if requested 1378 ⟩
    if ((format_ident ≡ 0) ∨ (buffer[loc] ≡ '&')) { if (format_ident ≠ 0) initialize();
          /* erase preloaded format */
      if (¬open_fmt_file()) exit(0);
      if (¬load_fmt_file()) { w_close(&fmt_file);
        exit(0);
      }
      w_close(&fmt_file);
      while ((loc < limit) ∧ (buffer[loc] ≡ '␣')) incr(loc);
    }
    if (eTeX_ex) wterm_ln("entering␣extended␣mode");
    if (Prote_ex) { Prote_initialize();
    }
    if (end_line_char_inactive) decr(limit);
    else buffer[limit] = end_line_char;
    fix_date_and_time();
    ⟨ Initialize the print selector based on interaction 74 ⟩;
    if ((loc < limit) ∧ (cat_code(buffer[loc]) ≠ escape)) start_input();    /* \input assumed */
  }
```

This code is used in section 1331.

1337. 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 $m \equiv 13$, there is an additional argument, l.)

```
⟨Last-minute procedures 1332⟩ +≡
#ifdef DEBUG
  static void debug_help(void)    /* routine to display various things */
  { int k, l, m, n;
    clear_terminal;
    loop { wake_up_terminal;
      print_nl("debug␣#␣(-1␣to␣exit):");
      update_terminal;
      if (fscanf(term_in.f, "␣%d", &m) < 1 ∨ m < 0) return;
      else if (m ≡ 0) { goto breakpoint;    /* go to every declared label at least once */
      breakpoint: m = 0;    /* 'BREAKPOINT' */
      }
      else { fscanf(term_in.f, "␣%d", &n);
        switch (m) { ⟨Numbered cases for debug_help 1338⟩
        default: print("?");
        }
      }
    }
  }
#endif
```

1338. ⟨Numbered cases for *debug_help* 1338⟩ ≡
case 1: *print_word*(*mem*[*n*]); **break**; /* display *mem*[*n*] in all forms */
case 2: *print_int*(*info*(*n*)); **break**;
case 3: *print_int*(*link*(*n*)); **break**;
case 4: *print_word*(*eqtb*[*n*]); **break**;
case 5: *print_word*(*font_info*[*n*]); **break**;
case 6: *print_word*(*save_stack*[*n*]); **break**;
case 7: *show_box*(*n*); **break**; /* show a box, abbreviated by *show_box_depth* and *show_box_breadth* */
case 8:
  { *breadth_max* = 10000;
    depth_threshold = *pool_size* − *pool_ptr* − 10;
    show_node_list(*n*); /* show a box in its entirety */
  } **break**;
case 9: *show_token_list*(*n*, *null*, 1000); **break**;
case 10: *slow_print*(*n*); **break**;
case 11: *check_mem*($n > 0$); **break**; /* check wellformedness; print new busy locations if $n > 0$ */
case 12: *search_mem*(*n*); **break**; /* look for pointers to *n* */
case 13:
  { *fscanf*(*term_in*.*f*, "␣%d", &*l*);
    print_cmd_chr(*n*, *l*);
  } **break**;
case 14:
  for ($k = 0$; $k \le n$; *k*++) *printn*(*buffer*[*k*]); **break**;
case 15:
  { *font_in_short_display* = *null_font*;
    short_display(*n*);
  } **break**;
case 16: *panicking* = ¬*panicking*; **break**;

This code is used in section 1337.

1339. 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.

1340. 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 $type \equiv whatsit\_node$, 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*(A) *link*$(A+1)$ /* language number, in the range 0 . . 255 */
#**define** *what_lhm*(A) *type*$(A+1)$ /* minimum left fragment, in the range 1 . . 63 */
#**define** *what_rhm*(A) *subtype*$(A+1)$ /* minimum right fragment, in the range 1 . . 63 */
#**define** *write_tokens*(A) *link*$(A+1)$ /* reference count of token list to write */
#**define** *write_stream*(A) *info*$(A+1)$ /* stream number (0 to 17) */
#**define** *open_name*(A) *link*$(A+1)$ /* string number of file name to open */
#**define** *open_area*(A) *info*$(A+2)$ /* string number of file area for *open_name* */
#**define** *open_ext*(A) *link*$(A+2)$ /* string number of file extension for *open_name* */

1341. The sixteen possible `\write` streams are represented by the *write_file* array. The jth file is open if and only if *write_open*[j] $\equiv$ *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 ⟩ +≡
  static alpha_file *write_file*[16];
  static bool *write_open*[18];

1342. ⟨ Set initial values of key variables 21 ⟩ +≡
  for ($k = 0$; $k \le 17$; k++) *write_open*[k] = *false*;

1343. 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 *latex_first_extension_code* 5
#define *latespecial_node* (*latex_first_extension_code* + 0)
  /* *subtype* in whatsits that represent `\special` things expanded during output */
#define *set_language_code* (*latex_first_extension_code* + 1) /* command modifier for `\setlanguage` */
#define *TeX_last_extension_cmd_mod* *set_language_code*

⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
  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*);

1344. The variable *write_loc* just introduced is used to provide an appropriate error message in case of "runaway" write texts.

⟨ Global variables 13 ⟩ +≡
  static pointer *write_loc*; /* *eqtb* address of `\write` */

1345. ⟨ Cases of *print_cmd_chr* for symbolic printing of primitives 226 ⟩ +≡
  case *extension*: **switch** (*chr_code*) {
case *open_node*: *print_esc*("`openout`"); **break**;
case *write_node*: *print_esc*("`write`"); **break**;
case *close_node*: *print_esc*("`closeout`"); **break**;
case *special_node*: *print_esc*("`special`"); **break**;
case *immediate_code*: *print_esc*("`immediate`"); **break**;
case *set_language_code*: *print_esc*("`setlanguage`"); **break**;
  ⟨ Cases of *extension* for *print_cmd_chr* 1604 ⟩
default: *print*("`[unknown␣extension!]`");
  } **break**;

1346. 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 1346 ⟩ ≡
any_mode(*extension*): *do_extension*( );
This code is used in section 1044.

1347. ⟨ Declare action procedures for use by *main_control* 1042 ⟩ +≡
⟨ Declare procedures needed in *do_extension* 1348 ⟩

```
static void do_extension(void){ int k;   /* all-purpose integer */
    pointer p;   /* all-purpose pointer */
    switch (cur_chr) {
  case open_node: ⟨ Implement \openout 1350 ⟩ break;
  case write_node: ⟨ Implement \write 1351 ⟩ break;
  case close_node: ⟨ Implement \closeout 1352 ⟩ break;
  case special_node: ⟨ Implement \special 1353 ⟩ break;
  case immediate_code: ⟨ Implement \immediate 1374 ⟩ break;
  case set_language_code: ⟨ Implement \setlanguage 1376 ⟩ break;
    ⟨ Cases for do_extension 1607 ⟩
  default: confusion("ext1");
    }
    }
```

1348. 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* 1348 ⟩ ≡

```
static void new_whatsit(small_number s, small_number w)
{ pointer p;   /* the new node */
  p = get_node(w);
  type(p) = whatsit_node;
  subtype(p) = s;
  link(tail) = p;
  tail = p;
}
```

See also section 1349.

This code is used in section 1347.

1349. 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* 1348 ⟩ +≡

```
static void new_write_whatsit(small_number w)
{ new_whatsit(cur_chr, w);
  if (w ≠ write_node_size) scan_four_bit_int();
  else { scan_int();
    if (cur_val < 0) cur_val = 17;
    else if (cur_val > 15) cur_val = 16;
  }
  write_stream(tail) = cur_val;
}
```

1350. ⟨ Implement `\openout` 1350 ⟩ ≡
{ *new_write_whatsit*(*open_node_size*);
scan_optional_equals( );
scan_file_name( );
open_name(*tail*) = *cur_name*;
open_area(*tail*) = *cur_area*;
open_ext(*tail*) = *cur_ext*;
}
This code is used in section 1347.

1351. 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` 1351 ⟩ ≡
{ $k = cur\_cs$;
new_write_whatsit(*write_node_size*);
$cur\_cs = k$;
$p = scan\_toks(false, false)$;
write_tokens(*tail*) = *def_ref*;
}
This code is used in section 1347.

1352. ⟨ Implement `\closeout` 1352 ⟩ ≡
{ *new_write_whatsit*(*write_node_size*);
write_tokens(*tail*) = *null*;
}
This code is used in section 1347.

1353. When '`\special{...}`' appears, we expand the macros in the token list as in `\xdef` and `\mark`. When marked with `shipout`, we keep tokens unexpanded for now.

⟨ Implement `\special` 1353 ⟩ ≡
{ **if** (*scan_keyword*("`shipout`")) { *new_whatsit*(*latespecial_node*, *write_node_size*);
write_stream(*tail*) = *null*;
$p = scan\_toks(false, false)$;
write_tokens(*tail*) = *def_ref*;
}
else { *new_whatsit*(*special_node*, *write_node_size*);
write_stream(*tail*) = *null*;
$p = scan\_toks(false, true)$;
write_tokens(*tail*) = *def_ref*;
}
}
This code is used in section 1347.

1354. 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 55 ⟩ +≡

```
static void print_write_whatsit(char *s, pointer p)
{ print_esc(s);
  if (write_stream(p) < 16) print_int(write_stream(p));
  else if (write_stream(p) ≡ 16) print_char('*');
  else print_char('-');
}
```

1355. ⟨ Display the whatsit node p 1355 ⟩ ≡

```
switch (subtype(p)) {
case open_node:
  { print_write_whatsit("openout", p);
    print_char('=');
    print_file_name(open_name(p), open_area(p), open_ext(p));
  } break;
case write_node:
  { print_write_whatsit("write", p);
    print_mark(write_tokens(p));
  } break;
case close_node: print_write_whatsit("closeout", p); break;
case latespecial_node:
  { print_esc("special");
    print("␣shipout");
    print_mark(write_tokens(p));
  } break;
case special_node:
  { print_esc("special");
    print_mark(write_tokens(p));
  } break;
case language_node:
  { print_esc("setlanguage");
    print_int(what_lang(p));
    print("␣(hyphenmin␣");
    print_int(what_lhm(p));
    print_char(',');
    print_int(what_rhm(p));
    print_char(')');
  } break;
  ⟨ Cases for displaying the whatsit node 1675 ⟩
default: print("whatsit?");
}
```

This code is used in section 182.

1356. ⟨ 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 1356 ⟩ ≡

switch (*subtype*(p)) {
case *open_node*:
{ $r = get\_node(open\_node\_size)$;
words = *open_node_size*;
} **break**;
case *write_node*: **case** *special_node*: **case** *latespecial_node*:
{ $r = get\_node(write\_node\_size)$;
add_token_ref(*write_tokens*(p));
words = *write_node_size*;
} **break**;
case *close_node*: **case** *language_node*:
{ $r = get\_node(small\_node\_size)$;
words = *small_node_size*;
} **break**;
⟨ Cases for making a partial copy of the whatsit node 1676 ⟩
default: *confusion*("ext2");
}

This code is used in section 205.

1357. ⟨ Wipe out the whatsit node p and **goto** *done* 1357 ⟩ ≡

{ **switch** (*subtype*(p)) {
case *open_node*: *free_node*(p, *open_node_size*); **break**;
case *write_node*: **case** *special_node*: **case** *latespecial_node*:
{ *delete_token_ref*(*write_tokens*(p));
free_node(p, *write_node_size*);
goto *done*;
}
case *close_node*: **case** *language_node*: *free_node*(p, *small_node_size*); **break**;
⟨ Cases for wiping out the whatsit node 1677 ⟩
default: *confusion*("ext3");
}
goto *done*;
}

This code is used in section 201.

1358. ⟨ Incorporate a whatsit node into a vbox 1358 ⟩ ≡

do_nothing

This code is used in section 668.

1359. ⟨ Incorporate a whatsit node into an hbox 1359 ⟩ ≡

do_nothing

This code is used in section 650.

1360. ⟨ Let d be the width of the whatsit p 1360 ⟩ ≡

$d = 0$

This code is used in section 1146.

1361. **#define** *adv_past*(*A*) **if** (*subtype*(*A*) ≡ *language_node*) { *cur_lang* = *what_lang*(*A*);
l_hyf = *what_lhm*(*A*);
r_hyf = *what_rhm*(*A*);
set_hyph_index;
}

⟨ Advance past a whatsit node in the *line_break* loop 1361 ⟩ ≡ *adv_past*(*cur_p*)

This code is used in section 865.

1362. ⟨ Advance past a whatsit node in the pre-hyphenation loop 1362 ⟩ ≡ *adv_past*(*s*)

This code is used in section 895.

1363. ⟨ Prepare to move whatsit *p* to the current page, then **goto** *contribute* 1363 ⟩ ≡
goto *contribute*

This code is used in section 999.

1364. ⟨ Process whatsit *p* in *vert_break* loop, **goto** *not_found* 1364 ⟩ ≡
goto *not_found*

This code is used in section 972.

1365. ⟨ Output the whatsit node *p* in a vlist 1365 ⟩ ≡
out_what(*p*)

This code is used in section 630.

1366. ⟨ Output the whatsit node *p* in an hlist 1366 ⟩ ≡
out_what(*p*)

This code is used in section 621.

1367. 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* 1367 ⟩ ≡
static void *special_out*(**pointer** *p*)
{ **pointer** *q*, *r*; /* temporary variables for list manipulation */
int *old_mode*; /* saved *mode* */
if (*subtype*(*p*) ≡ *latespecial_node*) {
⟨ Expand macros in the token list and make *link*(*def_ref*) point to the result 1370 ⟩;
write_tokens(*p*) = *def_ref*;
}
}

See also sections 1369 and 1372.

This code is used in section 618.

1368. 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) 163 ⟩ +≡
text(*end_write*) = *s_no*("`endwrite`");
eq_level(*end_write*) = *level_one*;
eq_type(*end_write*) = *outer_call*;
equiv(*end_write*) = *null*;

1369. ⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1367 ⟩ +≡

```
static void write_out(pointer p)
{ int old_setting;    /* holds print selector */
  int old_mode;    /* saved mode */
  small_number j;    /* write stream number */
  pointer q, r;    /* temporary variables for list manipulation */
  ⟨ Expand macros in the token list and make link(def_ref) point to the result 1370 ⟩;
  old_setting = selector;
  j = write_stream(p);
  if (write_open[j]) selector = j;
  else {    /* write to the terminal if file isn't open */
    if ((j ≡ 17) ∧ (selector ≡ term_and_log)) selector = log_only;
    print_nl("");
  }
  token_show(def_ref);
  print_ln();
  flush_list(def_ref);
  selector = old_setting;
}
```

1370. 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 1370 ⟩ ≡

```
  q = get_avail();
  info(q) = right_brace_token + '}';
  fl_mem[q] = FILE_LINE(system_file, system_insert);
  r = get_avail();
  link(q) = r;
  info(r) = end_write_token;
  ins_list(q);
  fl_mem[r] = FILE_LINE(system_file, system_insert);
  begin_token_list(write_tokens(p), write_text);
  q = get_avail();
  info(q) = left_brace_token + '{';
  ins_list(q);
  fl_mem[q] = FILE_LINE(system_file, system_insert);
    /* 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) ⟨ Recover from an unbalanced write command 1371 ⟩;
  mode = old_mode; end_token_list()    /* conserve stack space */
```

This code is used in sections 1367 and 1369.

1371. ⟨ Recover from an unbalanced write command 1371 ⟩ ≡

```
{ 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 ( ) ;
  do { get_token( );
  } while (¬(cur_tok ≡ end_write_token));
}
```

This code is used in section 1370.

1372. The *out_what* procedure takes care of outputting whatsit nodes for *vlist_out* and *hlist_out*.

⟨ Declare procedures needed in *hlist_out*, *vlist_out* 1367 ⟩ +≡
⟨ Declare procedures needed in *out_what* 1678 ⟩

```
static void out_what(pointer p){ small_number j;    /* write stream number */
    switch (subtype(p)) {
  case open_node: case write_node: case close_node:
    ⟨ Do some work that has been queued up for \write 1373 ⟩ break;
  case special_node: special_out(p); break;
  case language_node: do_nothing; break;
    ⟨ Cases for out_what 1679 ⟩
  default: confusion("ext4");
    }
    }
```

1373. 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` 1373 ⟩ ≡

```
if (¬doing_leaders) { j = write_stream(p);
  if (subtype(p) ≡ write_node) write_out(p);
  else { if (write_open[j]) a_close(&write_file[j]);
    if (subtype(p) ≡ close_node) write_open[j] = false;
    else if (j < 16) { cur_name = open_name(p);
      cur_area = open_area(p);
      cur_ext = open_ext(p);
      pack_cur_name(".tex");
      while (¬a_open_out(&write_file[j])) prompt_file_name("output␣file␣name", ".tex");
      write_open[j] = true;
    }
  }
}
```

This code is used in section 1372.

1374. 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` 1374⟩ ≡

```
{ get_x_token();
  if ((cur_cmd ≡ extension) ∧ (cur_chr ≤ close_node)) { p = tail;
    do_extension();   /* append a whatsit node */
    out_what(tail);   /* do the action immediately */
    flush_node_list(tail);
    tail = p;
    link(p) = null;
  }
  else back_input();
}
```

This code is used in section 1347.

1375. 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* 1042⟩ +≡

```
static void fix_language(void)
{ ASCII_code l;   /* the new current language */
  if (language ≤ 0) l = 0;
  else if (language > 255) l = 0;
  else l = language;
  if (l ≠ clang) { new_whatsit(language_node, small_node_size);
    what_lang(tail) = l;
    clang = l;
    what_lhm(tail) = norm_min(left_hyphen_min);
    what_rhm(tail) = norm_min(right_hyphen_min);
  }
}
```

1376. ⟨Implement `\setlanguage` 1376⟩ ≡

```
if (abs(mode) ≠ hmode) report_illegal_case();
else { new_whatsit(language_node, small_node_size);
  scan_int();
  if (cur_val ≤ 0) clang = 0;
  else if (cur_val > 255) clang = 0;
  else clang = cur_val;
  what_lang(tail) = clang;
  what_lhm(tail) = norm_min(left_hyphen_min);
  what_rhm(tail) = norm_min(right_hyphen_min);
}
```

This code is used in section 1347.

1377. ⟨Finish the extensions 1377⟩ ≡

```
for (k = 0; k ≤ 15; k++)
  if (write_open[k]) a_close(&write_file[k]);
```

See also section 1777.

This code is used in section 1332.

1378. The extended features of ε-TEX. The program has three modes of operation: (1) In TEX compatibility mode it fully deserves the name TEX and there are neither extended features nor additional primitive commands. There are, however, a few modifications that would be legitimate in any implementation of TEX such as, e.g., preventing inadequate results of the glue to DVI unit conversion during *ship_out*. (2) In extended mode there are additional primitive commands and the extended features of ε-TEX are available. (3) In PRōTE mode there are supplementary primitive commands that will be discussed in the section below.

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

The code below is designed to work for cases where '**#ifdef** `INIT` ... **#endif**' is a run-time switch.

⟨ Enable ε-TEX and furthermore Prote, if requested 1378 ⟩ ≡

```
#ifdef INIT
  if (iniversion ∧ (buffer[loc] ≡ '*' ∨ etexp))    /* TEX Live */
  { no_new_control_sequence = false;
    ⟨ Generate all ε-TEX primitives 1379 ⟩
    if (buffer[loc] ≡ '*') incr(loc);    /* TEX Live */
    eTeX_mode = 1;    /* enter extended mode */
    ⟨ Initialize variables for ε-TEX extended mode 1495 ⟩
    if (buffer[loc] ≡ '*' ∨ ltxp) { ⟨ Check PRōTE "constant" values for consistency 1566 ⟩
      ⟨ Generate all PRōTE primitives 1553 ⟩
      if (buffer[loc] ≡ '*') incr(loc);
      Prote_mode = 1;    /* enter PRōTE mode */
    }
  }
#endif
  if (¬no_new_control_sequence)    /* just entered extended mode ? */
    no_new_control_sequence = true; else
```

This code is used in section 1336.

1379. The ε-TEX features available in extended mode are grouped into two categories: (1) Some of them are permanently enabled and have no semantic effect as long as none of the additional primitives are executed. (2) The remaining ε-TEX features are optional and can be individually enabled and disabled. For each optional feature there is an ε-TEX state variable named `\...state`; the feature is enabled, resp. disabled by assigning a positive, resp. non-positive value to that integer.

```
#define eTeX_state_base  (int_base + eTeX_state_code)
#define eTeX_state(A)  eqtb[eTeX_state_base + A].i    /* an ε-TEX state variable */
#define eTeX_version_code  eTeX_int    /* code for \eTeXversion */
```

⟨ Generate all ε-TEX primitives 1379 ⟩ ≡

```
  primitive("lastnodetype", last_item, last_node_type_code);
  primitive("eTeXversion", last_item, eTeX_version_code);
  primitive("eTeXrevision", convert, eTeX_revision_code);
```

See also sections 1387, 1393, 1396, 1399, 1402, 1405, 1414, 1416, 1419, 1422, 1427, 1429, 1441, 1444, 1452, 1460, 1483, 1487, 1491, 1531, 1534, and 1538.

This code is used in section 1378.

1380. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ ≡

```
case last_node_type_code: print_esc("lastnodetype"); break;
case eTeX_version_code: print_esc("eTeXversion"); break;
```

See also sections 1394, 1397, 1400, 1403, 1461, 1484, 1488, 1554, 1569, 1603, 1640, and 1667.

This code is used in section 416.

1381. ⟨ Cases for fetching an integer value 1381 ⟩ ≡
case *eTeX_version_code*: *cur_val* = *eTeX_version*; **break**;
See also sections 1395, 1398, and 1485.
This code is used in section 423.

1382. #**define** *eTeX_ex* (*eTeX_mode* ≡ 1) /∗ is this extended mode? ∗/
⟨ Global variables 13 ⟩ +≡
static int *eTeX_mode*; /∗ identifies compatibility and extended mode ∗/

1383. ⟨ Initialize table entries (done by INITEX only) 163 ⟩ +≡
eTeX_mode = 0; /∗ initially we are in compatibility mode ∗/
⟨ Initialize variables for ε-TEX compatibility mode 1494 ⟩

1384. ⟨ Dump the ε-TEX state 1384 ⟩ ≡
dump_int(*eTeX_mode*);
for ($j = 0$; $j \le eTeX\_states - 1$; $j{+}{+}$) $eTeX\_state(j) = 0$; /∗ disable all enhancements ∗/
See also section 1440.
This code is used in section 1306.

1385. ⟨ Undump the ε-TEX state 1385 ⟩ ≡
undump(0, 1, *eTeX_mode*);
if (*eTeX_ex*) { ⟨ Initialize variables for ε-TEX extended mode 1495 ⟩;
}
else { ⟨ Initialize variables for ε-TEX compatibility mode 1494 ⟩;
}
This code is used in section 1307.

1386. The *eTeX_enabled* function simply returns its first argument as result. This argument is *true* if an optional ε-TEX feature is currently enabled; otherwise, if the argument is *false*, the function gives an error message.

⟨ Declare ε-TEX procedures for use by *main_control* 1386 ⟩ ≡
```
static bool eTeX_enabled(bool b, quarterword j, halfword k)
{ if (¬b) { print_err("Improper␣");
      print_cmd_chr(j, k);
      help1("Sorry,␣this␣optional␣e-TeX␣feature␣has␣been␣disabled.");
      error ( ) ;
    }
    return b;
}
```
See also sections 1409 and 1425.
This code is used in section 814.

1387. First we implement the additional ε-TEX parameters in the table of equivalents.

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
primitive("everyeof", *assign_toks*, *every_eof_loc*);
primitive("tracingassigns", *assign_int*, *int_base* + *tracing_assigns_code*);
primitive("tracinggroups", *assign_int*, *int_base* + *tracing_groups_code*);
primitive("tracingifs", *assign_int*, *int_base* + *tracing_ifs_code*);
primitive("tracingscantokens", *assign_int*, *int_base* + *tracing_scan_tokens_code*);
primitive("tracingnesting", *assign_int*, *int_base* + *tracing_nesting_code*);
primitive("savingvdiscards", *assign_int*, *int_base* + *saving_vdiscards_code*);
primitive("savinghyphcodes", *assign_int*, *int_base* + *saving_hyph_codes_code*);

1388. **#define** *every_eof* *equiv*(*every_eof_loc*)

⟨ Cases of *assign_toks* for *print_cmd_chr* 1388 ⟩ ≡
case *every_eof_loc*: *print_esc*("everyeof"); **break**;

This code is used in section 230.

1389. ⟨ Cases for *print_param* 1389 ⟩ ≡
case *tracing_assigns_code*: *print_esc*("tracingassigns"); **break**;
case *tracing_groups_code*: *print_esc*("tracinggroups"); **break**;
case *tracing_ifs_code*: *print_esc*("tracingifs"); **break**;
case *tracing_scan_tokens_code*: *print_esc*("tracingscantokens"); **break**;
case *tracing_nesting_code*: *print_esc*("tracingnesting"); **break**;
case *saving_vdiscards_code*: *print_esc*("savingvdiscards"); **break**;
case *saving_hyph_codes_code*: *print_esc*("savinghyphcodes"); **break**;

See also section 1539.

This code is used in section 236.

1390. In order to handle \everyeof we need an array *eof_seen* of boolean variables.

⟨ Global variables 13 ⟩ +≡
static bool *eof_seen0*[*max_in_open*], ∗**const** *eof_seen* = *eof_seen0* − 1; /∗ has eof been seen? ∗/

1391. The *print_group* procedure prints the current level of grouping and the name corresponding to *cur_group*.

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡

```
static void print_group(bool e)
{ switch (cur_group) {
  case bottom_level:
    { print("bottom␣level");
      return;
    }
  case simple_group: case semi_simple_group:
    { if (cur_group ≡ semi_simple_group) print("semi␣");
      print("simple");
    } break;
  case hbox_group: case adjusted_hbox_group:
    { if (cur_group ≡ adjusted_hbox_group) print("adjusted␣");
      print("hbox");
    } break;
  case vbox_group: print("vbox"); break;
  case vtop_group: print("vtop"); break;
  case align_group: case no_align_group:
    { if (cur_group ≡ no_align_group) print("no␣");
      print("align");
    } break;
  case output_group: print("output"); break;
  case disc_group: print("disc"); break;
  case insert_group: print("insert"); break;
  case vcenter_group: print("vcenter"); break;
  case math_group: case math_choice_group: case math_shift_group: case math_left_group:
    { print("math");
      if (cur_group ≡ math_choice_group) print("␣choice");
      else if (cur_group ≡ math_shift_group) print("␣shift");
      else if (cur_group ≡ math_left_group) print("␣left");
    }
  }   /* there are no other cases */
  print("␣group␣(level␣");
  print_int(qo(cur_level));
  print_char(')');
  if (saved(−1) ≠ 0) { if (e) print("␣entered␣at␣line␣");
    else print("␣at␣line␣");
    print_int(saved(−1));
  }
}
```

1392. The *group_trace* procedure is called when a new level of grouping begins ($e \equiv false$) or ends ($e \equiv true$) with *saved*(−1) containing the line number.

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡
```
#ifdef STAT
  static void group_trace(bool e)
  { begin_diagnostic();
    print_char('{');
    if (e) print("leaving␣");
    else print("entering␣");
    print_group(e);
    print_char('}');
    end_diagnostic(false);
  }
#endif
```

1393. The `\currentgrouplevel` and `\currentgrouptype` commands return the current level of grouping and the type of the current group respectively.

```
#define current_group_level_code (eTeX_int + 1)   /* code for \currentgrouplevel */
#define current_group_type_code (eTeX_int + 2)   /* code for \currentgrouptype */
```

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
```
  primitive("currentgrouplevel", last_item, current_group_level_code);
  primitive("currentgrouptype", last_item, current_group_type_code);
```

1394. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡
```
case current_group_level_code: print_esc("currentgrouplevel"); break;
case current_group_type_code: print_esc("currentgrouptype"); break;
```

1395. ⟨ Cases for fetching an integer value 1381 ⟩ +≡
```
case current_group_level_code: cur_val = cur_level − level_one; break;
case current_group_type_code: cur_val = cur_group; break;
```

1396. The `\currentiflevel`, `\currentiftype`, and `\currentifbranch` commands return the current level of conditionals and the type and branch of the current conditional.

```
#define current_if_level_code (eTeX_int + 3)   /* code for \currentiflevel */
#define current_if_type_code (eTeX_int + 4)   /* code for \currentiftype */
#define current_if_branch_code (eTeX_int + 5)   /* code for \currentifbranch */
```

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
```
  primitive("currentiflevel", last_item, current_if_level_code);
  primitive("currentiftype", last_item, current_if_type_code);
  primitive("currentifbranch", last_item, current_if_branch_code);
```

1397. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡
```
case current_if_level_code: print_esc("currentiflevel"); break;
case current_if_type_code: print_esc("currentiftype"); break;
case current_if_branch_code: print_esc("currentifbranch"); break;
```

1398. ⟨ Cases for fetching an integer value 1381 ⟩ +≡

case *current_if_level_code*:
  { $q = cond\_ptr$;
    $cur\_val = 0$;
    while $(q \neq null)$ { $incr(cur\_val)$;
      $q = link(q)$;
    }
  } **break**;
case *current_if_type_code*:
  if $(cond\_ptr \equiv null)$ $cur\_val = 0$;
  else if $(cur\_if < unless\_code)$ $cur\_val = cur\_if + 1$;
  else $cur\_val = -(cur\_if - unless\_code + 1)$; **break**;
case *current_if_branch_code*:
  if $((if\_limit \equiv or\_code) \lor (if\_limit \equiv else\_code))$ $cur\_val = 1$;
  else if $(if\_limit \equiv fi\_code)$ $cur\_val = -1$;
  else $cur\_val = 0$; **break**;

1399. The `\fontcharwd`, `\fontcharht`, `\fontchardp`, and `\fontcharic` commands return information about a character in a font.

#define *font_char_wd_code* *eTeX_dim*   /* code for `\fontcharwd` */
#define *font_char_ht_code* $(eTeX\_dim + 1)$ /* code for `\fontcharht` */
#define *font_char_dp_code* $(eTeX\_dim + 2)$ /* code for `\fontchardp` */
#define *font_char_ic_code* $(eTeX\_dim + 3)$ /* code for `\fontcharic` */

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
  primitive("fontcharwd", *last_item*, *font_char_wd_code*);
  primitive("fontcharht", *last_item*, *font_char_ht_code*);
  primitive("fontchardp", *last_item*, *font_char_dp_code*);
  primitive("fontcharic", *last_item*, *font_char_ic_code*);

1400. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡

case *font_char_wd_code*: *print_esc*("fontcharwd"); **break**;
case *font_char_ht_code*: *print_esc*("fontcharht"); **break**;
case *font_char_dp_code*: *print_esc*("fontchardp"); **break**;
case *font_char_ic_code*: *print_esc*("fontcharic"); **break**;

1401. ⟨ Cases for fetching a dimension value 1401 ⟩ ≡

case *font_char_wd_code*: **case** *font_char_ht_code*: **case** *font_char_dp_code*: **case** *font_char_ic_code*:
  { *scan_font_ident*( );
    $q = cur\_val$;
    scan_char_num( );
    if $((font\_bc[q] \leq cur\_val) \land (font\_ec[q] \geq cur\_val))$ { $i = char\_info(q, qi(cur\_val))$;
      switch (m) {
      case *font_char_wd_code*: $cur\_val = char\_width(q, i)$; **break**;
      case *font_char_ht_code*: $cur\_val = char\_height(q, height\_depth(i))$; **break**;
      case *font_char_dp_code*: $cur\_val = char\_depth(q, height\_depth(i))$; **break**;
      case *font_char_ic_code*: $cur\_val = char\_italic(q, i)$;
      }   /* there are no other cases */
    }
    else $cur\_val = 0$;
  } **break**;

See also sections 1404 and 1486.

This code is used in section 423.

1402. The `\parshapedimen`, `\parshapeindent`, and `\parshapelength` commands return the indent and length parameters of the current `\parshape` specification.

#define *par_shape_length_code* $(eTeX\_dim + 4)$ /* code for `\parshapelength` */
#define *par_shape_indent_code* $(eTeX\_dim + 5)$ /* code for `\parshapeindent` */
#define *par_shape_dimen_code* $(eTeX\_dim + 6)$ /* code for `\parshapedimen` */

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
primitive("parshapelength", *last_item*, *par_shape_length_code*);
primitive("parshapeindent", *last_item*, *par_shape_indent_code*);
primitive("parshapedimen", *last_item*, *par_shape_dimen_code*);

1403. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡
case *par_shape_length_code*: *print_esc*("parshapelength"); **break**;
case *par_shape_indent_code*: *print_esc*("parshapeindent"); **break**;
case *par_shape_dimen_code*: *print_esc*("parshapedimen"); **break**;

1404. ⟨ Cases for fetching a dimension value 1401 ⟩ +≡

```
case par_shape_length_code: case par_shape_indent_code: case par_shape_dimen_code:
  { q = cur_chr − par_shape_length_code;
    scan_int();
    if ((par_shape_ptr ≡ null) ∨ (cur_val ≤ 0)) cur_val = 0;
    else { if (q ≡ 2) { q = cur_val % 2;
          cur_val = (cur_val + q)/2;
        }
        if (cur_val > info(par_shape_ptr)) cur_val = info(par_shape_ptr);
        cur_val = mem[par_shape_ptr + 2 ∗ cur_val − q].sc;
    }
    cur_val_level = dimen_val;
  } break;
```

1405. The `\showgroups` command displays all currently active grouping levels.

#define *show_groups* 4 /* `\showgroups` */

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
primitive("showgroups", *xray*, *show_groups*);

1406. ⟨ Cases of *xray* for *print_cmd_chr* 1406 ⟩ ≡
case *show_groups*: *print_esc*("showgroups"); **break**;

See also sections 1415 and 1420.

This code is used in section 1291.

1407. ⟨ Cases for *show_whatever* 1407 ⟩ ≡

```
case show_groups:
  { begin_diagnostic();
    show_save_groups();
  } break;
```

See also section 1421.

This code is used in section 1292.

1408. ⟨ Types in the outer block 18 ⟩ +≡
typedef int32_t save_pointer; /* index into *save_stack* */

1409. The modifications of TEX required for the display produced by the *show_save_groups* procedure were first discussed by Donald E. Knuth in *TUGboat* **11**, 165–170 and 499–511, 1990.

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

⟨Declare ε-TEX procedures for use by *main_control* 1386⟩ +≡

```
static void show_save_groups(void)
{ int p;   /* index into nest */
  int m;   /* mode */
  save_pointer v;   /* saved value of save_ptr */
  quarterword l;   /* saved value of cur_level */
  group_code c;   /* saved value of cur_group */
  int a;   /* to keep track of alignments */
  int i;
  quarterword j;
  char *s;

  p = nest_ptr;
  nest[p] = cur_list;   /* put the top level into the array */
  v = save_ptr;
  l = cur_level;
  c = cur_group;
  save_ptr = cur_boundary;
  decr(cur_level);
  a = 1;
  print_nl("");
  print_ln();
  loop { print_nl("###␣");
    print_group(true);
    if (cur_group ≡ bottom_level) goto done;
    do { m = nest[p].mode_field;
      if (p > 0) decr(p);
      else m = vmode;
    } while (¬(m ≠ hmode));
    print("␣(");
    switch (cur_group) {
    case simple_group:
      { incr(p);
        goto found2;
      }
    case hbox_group: case adjusted_hbox_group: s = "hbox"; break;
    case vbox_group: s = "vbox"; break;
    case vtop_group: s = "vtop"; break;
    case align_group:
      if (a ≡ 0) { if (m ≡ −vmode) s = "halign";
        else s = "valign";
        a = 1;
        goto found1;
      }
      else { if (a ≡ 1) print("align␣entry");
        else print_esc("cr");
        if (p ≥ a) p = p − a;
        a = 0;
        goto found;
```

```
    } break;
  case no_align_group:
    { incr(p);
      a = −1;
      print_esc("noalign");
      goto found2;
    }
  case output_group:
    { print_esc("output");
      goto found;
    }
  case math_group: goto found2;
  case disc_group: case math_choice_group:
    { if (cur_group ≡ disc_group) print_esc("discretionary");
      else print_esc("mathchoice");
      for (i = 1; i ≤ 3; i++)
        if (i ≤ saved(−2)) print("{}");
      goto found2;
    }
  case insert_group:
    { if (saved(−2) ≡ 255) print_esc("vadjust");
      else { print_esc("insert");
        print_int(saved(−2));
      }
      goto found2;
    }
  case vcenter_group:
    { s = "vcenter";
      goto found1;
    }
  case semi_simple_group:
    { incr(p);
      print_esc("begingroup");
      goto found;
    }
  case math_shift_group:
    { if (m ≡ mmode) print_char('$');
      else if (nest[p].mode_field ≡ mmode) { print_cmd_chr(eq_no, saved(−2));
        goto found;
      }
      print_char('$');
      goto found;
    }
  case math_left_group:
    { if (type(nest[p + 1].eTeX_aux_field) ≡ left_noad) print_esc("left");
      else print_esc("middle");
      goto found;
    }
  }    /* there are no other cases */
  ⟨Show the box context 1411⟩;
found1: print_esc(s);
  ⟨Show the box packaging info 1410⟩;
```

```
    found2: print_char('{');
    found: print_char(')');
        decr(cur_level);
        cur_group = save_level(save_ptr);
        save_ptr = save_index(save_ptr);
    }
  done: save_ptr = v;
    cur_level = l;
    cur_group = c;
  }
```

1410. ⟨Show the box packaging info 1410⟩ ≡

```
  if (saved(−2) ≠ 0) { print_char('␣');
    if (saved(−3) ≡ exactly) print("to");
    else print("spread");
    print_scaled(saved(−2));
    print("pt");
  }
```

This code is used in section 1409.

1411. ⟨Show the box context 1411⟩ ≡

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

This code is used in section 1409.

1412. The *scan_general_text* procedure is much like *scan_toks*(*false*, *false*), but will be invoked via *expand*, i.e., recursively.

⟨Declare ε-TEX procedures for scanning 1412⟩ ≡

```
  static void scan_general_text(void);
```

See also sections 1454, 1463, and 1468.

This code is used in section 408.

1413. The token list (balanced text) created by *scan_general_text* begins at *link*(*temp_head*) and ends at *cur_val*. (If *cur_val* ≡ *temp_head*, the list is empty.)

⟨ Declare ε-TEX procedures for token lists 1413 ⟩ ≡

```
static void scan_general_text(void)
{ int s;   /* to save scanner_status */
  pointer w;   /* to save warning_index */
  pointer d;   /* to save def_ref */
  pointer p;   /* tail of the token list being built */
  pointer q;   /* new node being added to the token list via store_new_token */
  halfword unbalance;   /* number of unmatched left braces */

  s = scanner_status;
  w = warning_index;
  d = def_ref;
  scanner_status = absorbing;
  warning_index = cur_cs;
  def_ref = get_avail();
  token_ref_count(def_ref) = null;
  p = def_ref;
  scan_left_brace();   /* remove the compulsory left brace */
  unbalance = 1;
  loop { get_token();
    if (cur_tok < right_brace_limit)
      if (cur_cmd < right_brace) incr(unbalance);
      else { decr(unbalance);
        if (unbalance ≡ 0) goto found;
      }
    store_new_token(cur_tok);
  }
found: q = link(def_ref);
  free_avail(def_ref);   /* discard reference count */
  if (q ≡ null) cur_val = temp_head; else cur_val = p;
  link(temp_head) = q;
  scanner_status = s;
  warning_index = w;
  def_ref = d;
}
```

See also section 1435.

This code is used in section 463.

1414. The `\showtokens` command displays a token list.

#define *show_tokens* 5 /* `\showtokens` , must be odd! */

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
primitive("showtokens", *xray*, *show_tokens*);

1415. ⟨ Cases of *xray* for *print_cmd_chr* 1406 ⟩ +≡
case *show_tokens*: *print_esc*("showtokens"); **break**;

1416. The `\unexpanded` primitive prevents expansion of tokens much as the result from `\the` applied to a token variable. The `\detokenize` primitive converts a token list into a list of character tokens much as if the token list were written to a file. We use the fact that the command modifiers for `\unexpanded` and `\detokenize` are odd whereas those for `\the` and `\showthe` are even.

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
  primitive("unexpanded", *the*, 1);
  primitive("detokenize", *the*, *show_tokens*);

1417. ⟨ Cases of *the* for *print_cmd_chr* 1417 ⟩ ≡
  ; **else**
  if (*chr_code* ≡ 1) *print_esc*("unexpanded");
  else *print_esc*("detokenize")

This code is used in section 265.

1418. ⟨ Handle `\unexpanded` or `\detokenize` and **return** 1418 ⟩ ≡
  if (*odd*(*cur_chr*)) { *c* = *cur_chr*;
    scan_general_text( );
    if (*c* ≡ 1) **return** *cur_val*;
    else { *old_setting* = *selector*;
      selector = *new_string*;
      b = *pool_ptr*;
      p = *get_avail*( );
      link(*p*) = *link*(*temp_head*);
      token_show(*p*);
      flush_list(*p*);
      selector = *old_setting*;
      return *str_toks*(*b*);
    }
  }

This code is used in section 464.

1419. The `\showifs` command displays all currently active conditionals.

#define *show_ifs* 6 /* `\showifs` */

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
  primitive("showifs", *xray*, *show_ifs*);

1420. ⟨ Cases of *xray* for *print_cmd_chr* 1406 ⟩ +≡
case *show_ifs*: *print_esc*("showifs"); **break**;

1421. **#define** *print_if_line*(*A*)
  if ($A \neq 0$) { *print*("␣entered␣on␣line␣");
    print_int(*A*);
  }

⟨ Cases for *show_whatever* 1407 ⟩ +≡
case *show_ifs*:
  { *begin_diagnostic*( );
    print_nl("");
    print_ln( );
    if ($cond\_ptr \equiv null$) { *print_nl*("###␣");
      print("no␣active␣conditionals");
    }
    else { $p = cond\_ptr$;
      $n = 0$;
      do { *incr*(*n*);
        $p = link(p)$; } **while** $(\neg(p \equiv null))$;
      $p = cond\_ptr$;
      $t = cur\_if$;
      $l = if\_line$;
      $m = if\_limit$;
      do { *print_nl*("###␣level␣");
        print_int(*n*);
        print(":␣");
        print_cmd_chr(*if_test*, *t*);
        if ($m \equiv fi\_code$) *print_esc*("else");
        print_if_line(*l*);
        decr(*n*);
        $t = subtype(p)$;
        $l = if\_line\_field(p)$;
        $m = type(p)$;
        $p = link(p)$;
      } **while** $(\neg(p \equiv null))$;
    }
  } **break**;

1422. The \interactionmode primitive allows to query and set the interaction mode.

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
  primitive("interactionmode", *set_page_int*, 2);

1423. ⟨ Cases of *set_page_int* for *print_cmd_chr* 1423 ⟩ ≡
  ; **else if** ($chr\_code \equiv 2$) *print_esc*("interactionmode")

This code is used in section 416.

1424. ⟨ Cases for 'Fetch the *dead_cycles* or the *insert_penalties*' 1424 ⟩ ≡
  ; **else if** ($m \equiv 2$) $cur\_val = interaction$

This code is used in section 418.

1425. ⟨ Declare ε-TEX procedures for use by *main_control* 1386 ⟩ +≡
  static void *new_interaction*(**void**);

1426. ⟨ Cases for *alter_integer* 1426 ⟩ ≡

```
;
else
  if (c ≡ 2) { if ((cur_val < batch_mode) ∨ (cur_val > error_stop_mode)) {
      print_err("Bad␣interaction␣mode");
      help2("Modes␣are␣0=batch,␣1=nonstop,␣2=scroll,␣and",
      "3=errorstop.␣Proceed,␣and␣I'll␣ignore␣this␣case.");
      int_error(cur_val);
    }
    else { cur_chr = cur_val;
      new_interaction();
    }
  }
```

This code is used in section 1245.

1427. The *middle* feature of ε-TEX allows one ore several `\middle` delimiters to appear between `\left` and `\right`.

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
primitive("middle", *left_right*, *middle_noad*);

1428. ⟨ Cases of *left_right* for *print_cmd_chr* 1428 ⟩ ≡
; **else if** (*chr_code* ≡ *middle_noad*) *print_esc*("middle")

This code is used in section 1188.

1429. The *scan_tokens* feature of ε-TEX defines the `\scantokens` primitive.

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
primitive("scantokens", *input*, 2);

1430. ⟨ Cases of *input* for *print_cmd_chr* 1430 ⟩ ≡
; **else if** (*chr_code* ≡ 2) *print_esc*("scantokens")

This code is used in section 376.

1431. ⟨ Cases for *input* 1431 ⟩ ≡
; **else if** (*cur_chr* ≡ 2) *pseudo_start*( )

This code is used in section 377.

1432. The global variable *pseudo_files* is used to maintain a stack of pseudo files. The *info* field of each pseudo file points to a linked list of variable size nodes representing lines not yet processed: the *info* field of the first word contains the size of this node, all the following words contain ASCII codes.

⟨ Global variables 13 ⟩ +≡
static pointer *pseudo_files*; /∗ stack of pseudo files ∗/

1433. ⟨ Set initial values of key variables 21 ⟩ +≡
pseudo_files = *null*;

1434. The *pseudo_start* procedure initiates reading from a pseudo file.

⟨ Declare ε-TEX procedures for expanding 1434 ⟩ ≡
static void *pseudo_start*(**void**);

See also sections 1492, 1497, and 1501.

This code is used in section 365.

1435. ⟨ Declare ε-TEX procedures for token lists 1413 ⟩ +≡

```
static void pseudo_start(void)
{ int old_setting;    /* holds selector setting */
  str_number s;    /* string to be converted into a pseudo file */
  pool_pointer l, m;    /* indices into str_pool */
  pointer p, q, r;    /* for list construction */
  four_quarters w;    /* four ASCII codes */
  int nl, sz;

  scan_general_text();
  old_setting = selector;
  selector = new_string;
  token_show(temp_head);
  selector = old_setting;
  flush_list(link(temp_head));
  str_room(1);
  s = make_string();
  ⟨ Convert string s into a new pseudo file 1436 ⟩;
  flush_string;
  ⟨ Initiate input from new pseudo file 1437 ⟩;
}
```

1436. ⟨Convert string s into a new pseudo file 1436⟩ ≡

```
str_pool[pool_ptr] = si('␣');
l = str_start[s];
nl = si(new_line_char);
p = get_avail();
q = p;
while (l < pool_ptr) { m = l;
  while ((l < pool_ptr) ∧ (str_pool[l] ≠ nl)) incr(l);
  sz = (l − m + 7)/4;
  if (sz ≡ 1) sz = 2;
  r = get_node(sz);
  link(q) = r;
  q = r;
  info(q) = hi(sz);
  while (sz > 2) { decr(sz);
    incr(r);
    w.b0 = qi(so(str_pool[m]));
    w.b1 = qi(so(str_pool[m + 1]));
    w.b2 = qi(so(str_pool[m + 2]));
    w.b3 = qi(so(str_pool[m + 3]));
    mem[r].qqqq = w;
    m = m + 4;
  }
  w.b0 = qi('␣');
  w.b1 = qi('␣');
  w.b2 = qi('␣');
  w.b3 = qi('␣');
  if (l > m) { w.b0 = qi(so(str_pool[m]));
    if (l > m + 1) { w.b1 = qi(so(str_pool[m + 1]));
      if (l > m + 2) { w.b2 = qi(so(str_pool[m + 2]));
        if (l > m + 3) w.b3 = qi(so(str_pool[m + 3]));
      }
    }
  }
  mem[r + 1].qqqq = w;
  if (str_pool[l] ≡ nl) incr(l);
}
info(p) = link(p);
link(p) = pseudo_files; pseudo_files = p
```

This code is used in section 1435.

1437. ⟨ Initiate input from new pseudo file 1437 ⟩ ≡

```
begin_file_reading();     /* set up cur_file and new level of input */
line = 0;
limit = start;
loc = limit + 1;     /* force line read */
if (tracing_scan_tokens > 0) { if (term_offset > max_print_line − 3) print_ln();
  else if ((term_offset > 0) ∨ (file_offset > 0)) print_char('␣');
  name = 19;
  print("(␣");
  incr(open_parens);
  update_terminal;
}
else name = 18
```

This code is used in section 1435.

1438. Here we read a line from the current pseudo file into *buffer*.

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡

```
static bool pseudo_input(void)     /* inputs the next line or returns false */
{ pointer p;     /* current line from pseudo file */
  int sz;     /* size of node p */
  four_quarters w;     /* four ASCII codes */
  int r;     /* loop index */
  last = first;     /* cf. Matthew 19 : 30 */
  p = info(pseudo_files);
  if (p ≡ null) return false;
  else { info(pseudo_files) = link(p);
    sz = ho(info(p));
    if (4 * sz − 3 ≥ buf_size − last) ⟨ Report overflow of the input buffer, and abort 35 ⟩;
    last = first;
    for (r = p + 1; r ≤ p + sz − 1; r++) { w = mem[r].qqqq;
      buffer[last] = w.b0;
      buffer[last + 1] = w.b1;
      buffer[last + 2] = w.b2;
      buffer[last + 3] = w.b3;
      last = last + 4;
    }
    if (last ≥ max_buf_stack) max_buf_stack = last + 1;
    while ((last > first) ∧ (buffer[last − 1] ≡ '␣')) decr(last);
    free_node(p, sz);
    return true;
  }
}
```

1439. When we are done with a pseudo file we 'close' it.

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡
```
static void pseudo_close(void)   /* close the top level pseudo file */
{ pointer p, q;
  p = link(pseudo_files);
  q = info(pseudo_files);
  free_avail(pseudo_files);
  pseudo_files = p;
  while (q ≠ null) { p = q;
    q = link(p);
    free_node(p, ho(info(p)));
  }
}
```

1440. ⟨ Dump the ε-TEX state 1384 ⟩ +≡
```
while (pseudo_files ≠ null) pseudo_close();   /* flush pseudo files */
```

1441. ⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
```
primitive("readline", read_to_cs, 1);
```

1442. ⟨ Cases of *read* for *print_cmd_chr* 1442 ⟩ ≡
```
; else print_esc("readline")
```
This code is used in section 265.

1443. ⟨ Handle `\readline` and **goto** *done* 1443 ⟩ ≡
```
if (j ≡ 1) { while (loc ≤ limit)   /* current line not yet finished */
  { cur_chr = buffer[loc];
    incr(loc);
    if (cur_chr ≡ '␣') cur_tok = space_token; else cur_tok = cur_chr + other_token;
    store_new_token(cur_tok);
  }
  goto done;
}
```
This code is used in section 482.

1444. Here we define the additional conditionals of ε-TEX as well as the `\unless` prefix.

```
#define if_def_code 17   /* '\ifdefined' */
#define if_cs_code 18   /* '\ifcsname' */
#define if_font_char_code 19   /* '\iffontchar' */
#define eTeX_last_if_test_cmd_mod if_font_char_code
#define eTeX_last_expand_after_cmd_mod 1
```

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
```
primitive("unless", expand_after, 1);
primitive("ifdefined", if_test, if_def_code);
primitive("ifcsname", if_test, if_cs_code);
primitive("iffontchar", if_test, if_font_char_code);
```

1445. ⟨ Cases of *expandafter* for *print_cmd_chr* 1445 ⟩ ≡
```
case 1: print_esc("unless"); break;
```
See also sections 1579 and 1589.

This code is used in section 265.

1446. ⟨ Cases of *if_test* for *print_cmd_chr* 1446 ⟩ ≡
case *if_def_code*: *print_esc*(`"ifdefined"`); **break**;
case *if_cs_code*: *print_esc*(`"ifcsname"`); **break**;
case *if_font_char_code*: *print_esc*(`"iffontchar"`); **break**;

See also section 1572.
This code is used in section 487.

1447. The result of a boolean condition is reversed when the conditional is preceded by `\unless`.

⟨ Negate a boolean conditional and **goto** *reswitch* 1447 ⟩ ≡
{ *get_token*( );
  if $((cur\_cmd \equiv if\_test) \land (cur\_chr \neq if\_case\_code))$ { $cur\_chr = cur\_chr + unless\_code$;
    goto *reswitch*;
  }
  print_err(`"You␣can't␣use␣`"`);
  print_esc(`"unless"`);
  print(`"'␣before␣`"`);
  print_cmd_chr(*cur_cmd*, *cur_chr*);
  print_char(`'\''`);
  help1(`"Continue,␣and␣I'll␣forget␣that␣it␣ever␣happened."`);
  back_error( );
}

This code is used in section 366.

1448. The conditional `\ifdefined` tests if a control sequence is defined.

We need to reset *scanner_status*, since `\outer` control sequences are allowed, but we might be scanning a macro definition or preamble.

⟨ Cases for *conditional* 1448 ⟩ ≡
case *if_def_code*:
  { *save_scanner_status* = *scanner_status*;
    scanner_status = *normal*;
    get_next( );
    $b = (cur\_cmd \neq undefined\_cs)$;
    scanner_status = *save_scanner_status*;
  } **break**;

See also sections 1449, 1451, 1574, and 1576.
This code is used in section 500.

1449. The conditional `\ifcsname` is equivalent to `{\expandafter }\expandafter \ifdefined \csname`, except that no new control sequence will be entered into the hash table (once all tokens preceding the mandatory `\endcsname` have been expanded).

⟨ Cases for *conditional* 1448 ⟩ +≡
case *if_cs_code*:
  { $n = get\_avail()$;
    $p = n$; /* head of the list of characters */
    do { *get_x_token*( );
      if $(cur\_cs \equiv 0)$ *store_new_token*(*cur_tok*);
    } **while** $(\neg(cur\_cs \neq 0))$;
    if $(cur\_cmd \neq end\_cs\_name)$ ⟨ Complain about missing `\endcsname` 372 ⟩;
    ⟨ Look up the characters of list n in the hash table, and set *cur_cs* 1450 ⟩;
    flush_list(n);
    $b = (eq\_type(cur\_cs) \neq undefined\_cs)$;
  } **break**;

1450. ⟨ Look up the characters of list n in the hash table, and set *cur_cs* 1450 ⟩ ≡
  $m = first$;
  $p = link(n)$;
  while $(p \neq null)$ { **if** $(m \geq max\_buf\_stack)$ { $max\_buf\_stack = m + 1$;
      if $(max\_buf\_stack \equiv buf\_size)$ *overflow*("buffer␣size", *buf_size*);
    }
    $buffer[m] = info(p) \,\%\, {}^\circ 400$;
    incr(m);
    $p = link(p)$;
  }
  if $(m \equiv first)$ $cur\_cs = null\_cs$; /* the list is empty */
  else if $(m > first + 1)$ $cur\_cs = id\_lookup(first, m - first)$; /* *no_new_control_sequence* is *true* */
  else $cur\_cs = single\_base + buffer[first]$ /* the list has length one */

This code is used in section 1449.

1451. The conditional `\iffontchar` tests the existence of a character in a font.

⟨ Cases for *conditional* 1448 ⟩ +≡
case *if_font_char_code*:
  { *scan_font_ident*( );
    $n = cur\_val$;
    scan_char_num( );
    if $((font\_bc[n] \leq cur\_val) \land (font\_ec[n] \geq cur\_val))$ $b = char\_exists(char\_info(n, qi(cur\_val)))$;
    else $b = false$;
  } **break**;

1452. The `protected` feature of ε-TEX defines the `\protected` prefix command for macro definitions. Such macros are protected against expansions when lists of expanded tokens are built, e.g., for `\edef` or during `\write`.

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
  primitive("protected", *prefix*, 8);

1453. ⟨ Cases of *prefix* for *print_cmd_chr* 1453 ⟩ ≡
  ; **else if** $(chr\_code \equiv 8)$ *print_esc*("protected")

This code is used in section 1208.

1454. The *get_x_or_protected* procedure is like *get_x_token* except that protected macros are not expanded.

⟨ Declare ε-TEX procedures for scanning 1412 ⟩ +≡

```
static void get_x_or_protected(void)
    /* sets cur_cmd, cur_chr, cur_tok, and expands non-protected macros */
{ loop { get_token();
    if (cur_cmd ≤ max_command) return;
    if ((cur_cmd ≥ call) ∧ (cur_cmd < end_template))
      if (info(link(cur_chr)) ≡ protected_token) return;
    expand();
  }
}
```

1455. A group entered (or a conditional started) in one file may end in a different file. Such slight anomalies, although perfectly legitimate, may cause errors that are difficult to locate. In order to be able to give a warning message when such anomalies occur, ε-TEX uses the *grp_stack* and *if_stack* arrays to record the initial *cur_boundary* and *cond_ptr* values for each input file.

⟨ Global variables 13 ⟩ +≡

```
static save_pointer grp_stack[max_in_open + 1];    /* initial cur_boundary */
static pointer if_stack[max_in_open + 1];    /* initial cond_ptr */
```

1456. When a group ends that was apparently entered in a different input file, the *group_warning* procedure is invoked in order to update the *grp_stack*. If moreover `\tracingnesting` is positive we want to give a warning message. The situation is, however, somewhat complicated by two facts: (1) There may be *grp_stack* elements without a corresponding `\input` file or `\scantokens` pseudo file (e.g., error insertions from the terminal); and (2) the relevant information is recorded in the *name_field* of the *input_stack* only loosely synchronized with the *in_open* variable indexing *grp_stack*.

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡

```
static void group_warning(void)
{ int i;    /* index into grp_stack */
  bool w;    /* do we need a warning? */

  base_ptr = input_ptr;
  input_stack[base_ptr] = cur_input;    /* store current state */
  i = in_open;
  w = false;
  while ((grp_stack[i] ≡ cur_boundary) ∧ (i > 0)) {
    ⟨ Set variable w to indicate if this case should be reported 1457 ⟩;
    grp_stack[i] = save_index(save_ptr);
    decr(i);
  }
  if (w) { print_nl("Warning:␣end␣of␣");
    print_group(true);
    print("␣of␣a␣different␣file");
    print_ln();
    if (tracing_nesting > 1) show_context();
    if (history ≡ spotless) history = warning_issued;
  }
}
```

1457. This code scans the input stack in order to determine the type of the current input file.

⟨ Set variable w to indicate if this case should be reported 1457 ⟩ ≡

```
if (tracing_nesting > 0) { while ((input_stack[base_ptr].state_field ≡ token_list) ∨
        (input_stack[base_ptr].index_field > i)) decr(base_ptr);
    if (input_stack[base_ptr].name_field > 17) w = true;
}
```

This code is used in sections 1456 and 1458.

1458. When a conditional ends that was apparently started in a different input file, the *if_warning* procedure is invoked in order to update the *if_stack*. If moreover `\tracingnesting` is positive we want to give a warning message (with the same complications as above).

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡

```
static void if_warning(void)
{ int i;    /* index into if_stack */
  bool w;    /* do we need a warning? */
  base_ptr = input_ptr;
  input_stack[base_ptr] = cur_input;    /* store current state */
  i = in_open;
  w = false;
  while (if_stack[i] ≡ cond_ptr) { ⟨ Set variable w to indicate if this case should be reported 1457 ⟩;
    if_stack[i] = link(cond_ptr);
    decr(i);
  }
  if (w) { print_nl("Warning:␣end␣of␣");
    print_cmd_chr(if_test, cur_if);
    print_if_line(if_line);
    print("␣of␣a␣different␣file");
    print_ln();
    if (tracing_nesting > 1) show_context();
    if (history ≡ spotless) history = warning_issued;
  }
}
```

1459. Conversely, the *file_warning* procedure is invoked when a file ends and some groups entered or conditionals started while reading from that file are still incomplete.

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡

```
static void file_warning(void)
{ pointer p;   /* saved value of save_ptr or cond_ptr */
  quarterword l;   /* saved value of cur_level or if_limit */
  quarterword c;   /* saved value of cur_group or cur_if */
  int i;   /* saved value of if_line */

  p = save_ptr;
  l = cur_level;
  c = cur_group;
  save_ptr = cur_boundary;
  while (grp_stack[in_open] ≠ save_ptr) { decr(cur_level);
    print_nl("Warning:␣end␣of␣file␣when␣");
    print_group(true);
    print("␣is␣incomplete");
    cur_group = save_level(save_ptr);
    save_ptr = save_index(save_ptr);
  }
  save_ptr = p;
  cur_level = l;
  cur_group = c;   /* restore old values */
  p = cond_ptr;
  l = if_limit;
  c = cur_if;
  i = if_line;
  while (if_stack[in_open] ≠ cond_ptr) { print_nl("Warning:␣end␣of␣file␣when␣");
    print_cmd_chr(if_test, cur_if);
    if (if_limit ≡ fi_code) print_esc("else");
    print_if_line(if_line);
    print("␣is␣incomplete");
    if_line = if_line_field(cond_ptr);
    cur_if = subtype(cond_ptr);
    if_limit = type(cond_ptr);
    cond_ptr = link(cond_ptr);
  }
  cond_ptr = p;
  if_limit = l;
  cur_if = c;
  if_line = i;   /* restore old values */
  print_ln();
  if (tracing_nesting > 1) show_context();
  if (history ≡ spotless) history = warning_issued;
}
```

1460. Here are the additional ε-TEX primitives for expressions.

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡

```
primitive("numexpr", last_item, eTeX_expr − int_val + int_val);
primitive("dimexpr", last_item, eTeX_expr − int_val + dimen_val);
primitive("glueexpr", last_item, eTeX_expr − int_val + glue_val);
primitive("muexpr", last_item, eTeX_expr − int_val + mu_val);
```

1461. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡

```
case eTeX_expr − int_val + int_val: print_esc("numexpr"); break;
case eTeX_expr − int_val + dimen_val: print_esc("dimexpr"); break;
case eTeX_expr − int_val + glue_val: print_esc("glueexpr"); break;
case eTeX_expr − int_val + mu_val: print_esc("muexpr"); break;
```

1462. This code for reducing *cur_val_level* and/or negating the result is similar to the one for all the other cases of *scan_something_internal*, with the difference that *scan_expr* has already increased the reference count of a glue specification.

⟨ Process an expression and **return** 1462 ⟩ ≡

```
{ if (m < eTeX_mu) { switch (m) {
      ⟨ Cases for fetching a glue value 1489 ⟩
      }   /* there are no other cases */
      cur_val_level = glue_val;
   }
   else if (m < eTeX_expr) { switch (m) {
      ⟨ Cases for fetching a mu value 1490 ⟩
      }   /* there are no other cases */
      cur_val_level = mu_val;
   }
   else { cur_val_level = m − eTeX_expr + int_val;
      scan_expr();
   }
   while (cur_val_level > level) { if (cur_val_level ≡ glue_val) { m = cur_val;
         cur_val = width(m);
         delete_glue_ref(m);
      }
      else if (cur_val_level ≡ mu_val) mu_error();
      decr(cur_val_level);
   }
   if (negative)
      if (cur_val_level ≥ glue_val) { m = cur_val;
         cur_val = new_spec(m);
         delete_glue_ref(m);
         ⟨ Negate all three glue components of cur_val 430 ⟩;
      }
      else negate(cur_val);
   return;
}
```

This code is used in section 423.

1463. ⟨ Declare ε-TEX procedures for scanning 1412 ⟩ +≡

```
static void scan_expr(void);
```

1464. The *scan_expr* procedure scans and evaluates an expression.

⟨ Declare procedures needed for expressions 1464 ⟩ ≡
⟨ Declare subprocedures for *scan_expr* 1475 ⟩

```
static void scan_expr(void)    /* scans and evaluates an expression */
{ bool a, b;    /* saved values of arith_error */
  small_number l;    /* type of expression */
  small_number r;    /* state of expression so far */
  small_number s;    /* state of term so far */
  small_number o;    /* next operation or type of next factor */
  int e;    /* expression so far */
  int t;    /* term so far */
  int f;    /* current factor */
  int n;    /* numerator of combined multiplication and division */
  pointer p;    /* top of expression stack */
  pointer q;    /* for stack manipulations */
  l = cur_val_level;
  a = arith_error;
  b = false;
  p = null;
  ⟨Scan and evaluate an expression e of type l 1465⟩;
  if (b) { print_err("Arithmetic␣overflow");
    help2("I␣can't␣evaluate␣this␣expression,",
    "since␣the␣result␣is␣out␣of␣range.");
    error ( ) ;
    if (l ≥ glue_val) { delete_glue_ref(e);
      e = zero_glue;
      add_glue_ref(e);
    }
    else e = 0;
  }
  arith_error = a;
  cur_val = e;
  cur_val_level = l;
}
```

See also section 1469.

This code is used in section 460.

1465. Evaluating an expression is a recursive process: When the left parenthesis of a subexpression is scanned we descend to the next level of recursion; the previous level is resumed with the matching right parenthesis.

```
#define expr_none 0   /* ( seen, or ( ⟨expr⟩ ) seen */
#define expr_add 1   /* ( ⟨expr⟩ + seen */
#define expr_sub 2   /* ( ⟨expr⟩ - seen */
#define expr_mult 3   /* ⟨term⟩ * seen */
#define expr_div 4   /* ⟨term⟩ / seen */
#define expr_scale 5   /* ⟨term⟩ * ⟨factor⟩ / seen */
```

⟨ Scan and evaluate an expression e of type l 1465 ⟩ ≡

```
restart: r = expr_none;
  e = 0;
  s = expr_none;
  t = 0;
  n = 0;
resume:
  if (s ≡ expr_none) o = l; else o = int_val;
  ⟨ Scan a factor f of type o or start a subexpression 1467 ⟩;
found: ⟨ Scan the next operator and set o 1466 ⟩;
  arith_error = b;
  ⟨ Make sure that f is in the proper range 1472 ⟩;
  switch (s) { ⟨ Cases for evaluation of the current term 1473 ⟩
  }   /* there are no other cases */
  if (o > expr_sub) s = o; else ⟨ Evaluate the current expression 1474 ⟩;
  b = arith_error;
  if (o ≠ expr_none) goto resume;
  if (p ≠ null) ⟨ Pop the expression stack and goto found 1471 ⟩
```

This code is used in section 1464.

1466. ⟨ Scan the next operator and set o 1466 ⟩ ≡

```
  ⟨ Get the next non-blank non-call token 405 ⟩;
  if (cur_tok ≡ other_token + '+') o = expr_add;
  else if (cur_tok ≡ other_token + '-') o = expr_sub;
  else if (cur_tok ≡ other_token + '*') o = expr_mult;
  else if (cur_tok ≡ other_token + '/') o = expr_div;
  else { o = expr_none;
    if (p ≡ null) { if (cur_cmd ≠ relax) back_input();
    }
    else if (cur_tok ≠ other_token + ')') { print_err("Missing␣)␣inserted␣for␣expression");
      help1("I␣was␣expecting␣to␣see␣‘+’,␣‘-’,␣‘*’,␣‘/’,␣or␣‘)’.␣Didn’t.");
      back_error();
    }
  }
```

This code is used in section 1465.

1467. ⟨ Scan a factor f of type o or start a subexpression 1467 ⟩ ≡
⟨ Get the next non-blank non-call token 405 ⟩;
if (*cur_tok* ≡ *other_token* + '(') ⟨ Push the expression stack and **goto** *restart* 1470 ⟩;
back_input( );
if (o ≡ *int_val*) *scan_int*( );
else if (o ≡ *dimen_val*) *scan_normal_dimen*;
else if (o ≡ *glue_val*) *scan_normal_glue*( );
else *scan_mu_glue*( );
f = *cur_val*

This code is used in section 1465.

1468. ⟨ Declare ε-TEX procedures for scanning 1412 ⟩ +≡
static void *scan_normal_glue*(**void**);
static void *scan_mu_glue*(**void**);

1469. Here we declare two trivial procedures in order to avoid mutually recursive procedures with parameters.

⟨ Declare procedures needed for expressions 1464 ⟩ +≡
static void *scan_normal_glue*(**void**)
{ *scan_glue*(*glue_val*);
}
static void *scan_mu_glue*(**void**)
{ *scan_glue*(*mu_val*);
}

1470. Parenthesized subexpressions can be inside expressions, and this nesting has a stack. Seven local variables represent the top of the expression stack: p points to pushed-down entries, if any; l specifies the type of expression currently beeing evaluated; e is the expression so far and r is the state of its evaluation; t is the term so far and s is the state of its evaluation; finally n is the numerator for a combined multiplication and division, if any.

#**define** *expr_node_size* 4 /* number of words in stack entry for subexpressions */
#**define** *expr_e_field*(A) *mem*[$A + 1$].i /* saved expression so far */
#**define** *expr_t_field*(A) *mem*[$A + 2$].i /* saved term so far */
#**define** *expr_n_field*(A) *mem*[$A + 3$].i /* saved numerator */

⟨ Push the expression stack and **goto** *restart* 1470 ⟩ ≡
{ q = *get_node*(*expr_node_size*);
link(q) = p;
type(q) = l;
subtype(q) = $4 * s + r$;
expr_e_field(q) = e;
expr_t_field(q) = t;
expr_n_field(q) = n;
p = q;
l = o;
goto *restart*;
}

This code is used in section 1467.

1471. ⟨ Pop the expression stack and **goto** *found* 1471 ⟩ ≡

```
{ f = e;
  q = p;
  e = expr_e_field(q);
  t = expr_t_field(q);
  n = expr_n_field(q);
  s = subtype(q)/4;
  r = subtype(q) % 4;
  l = type(q);
  p = link(q);
  free_node(q, expr_node_size);
  goto found;
}
```

This code is used in section 1465.

1472. We want to make sure that each term and (intermediate) result is in the proper range. Integer values must not exceed *infinity* ($2^{31} - 1$) in absolute value, dimensions must not exceed *max_dimen* ($2^{30} - 1$). We avoid the absolute value of an integer, because this might fail for the value -2^{31} using 32-bit arithmetic.

```
#define num_error(A)    /* clear a number or dimension and set arith_error */
        { arith_error = true;
          A = 0;
        }
#define glue_error(A)    /* clear a glue spec and set arith_error */
        { arith_error = true;
          delete_glue_ref(A);
          A = new_spec(zero_glue);
        }
```

⟨ Make sure that f is in the proper range 1472 ⟩ ≡

```
if ((l ≡ int_val) ∨ (s > expr_sub)) { if ((f > infinity) ∨ (f < −infinity)) num_error(f);
}
else if (l ≡ dimen_val) { if (abs(f) > max_dimen) num_error(f);
}
else { if ((abs(width(f)) > max_dimen) ∨
          (abs(stretch(f)) > max_dimen) ∨
          (abs(shrink(f)) > max_dimen)) glue_error(f);
}
```

This code is used in section 1465.

1473. Applying the factor f to the partial term t (with the operator s) is delayed until the next operator o has been scanned. Here we handle the first factor of a partial term. A glue spec has to be copied unless the next operator is a right parenthesis; this allows us later on to simply modify the glue components.

```
#define normalize_glue(A)
        if (stretch(A) ≡ 0) stretch_order(A) = normal;
        if (shrink(A) ≡ 0) shrink_order(A) = normal
```

⟨ Cases for evaluation of the current term 1473 ⟩ ≡

```
case expr_none:
  if ((l ≥ glue_val) ∧ (o ≠ expr_none)) { t = new_spec(f);
    delete_glue_ref(f);
    normalize_glue(t);
  }
  else t = f; break;
```

See also sections 1477, 1478, and 1480.

This code is used in section 1465.

1474. When a term t has been completed it is copied to, added to, or subtracted from the expression e.

```
#define expr_add_sub(A, B, C) add_or_sub(A, B, C, r ≡ expr_sub)
#define expr_a(A, B) expr_add_sub(A, B, max_dimen)
```

⟨ Evaluate the current expression 1474 ⟩ ≡

```
{ s = expr_none;
  if (r ≡ expr_none) e = t;
  else if (l ≡ int_val) e = expr_add_sub(e, t, infinity);
  else if (l ≡ dimen_val) e = expr_a(e, t);
  else ⟨ Compute the sum or difference of two glue specs 1476 ⟩;
  r = o;
}
```

This code is used in section 1465.

1475. The function $add\_or\_sub(x, y, max\_answer, negative)$ computes the sum (for $negative \equiv false$) or difference (for $negative \equiv true$) of x and y, provided the absolute value of the result does not exceed max_answer.

⟨ Declare subprocedures for *scan_expr* 1475 ⟩ ≡

```
static int add_or_sub(int x, int y, int max_answer, bool negative)
{ int a;   /* the answer */
  if (negative) negate(y);
  if (x ≥ 0)
    if (y ≤ max_answer − x) a = x + y; else num_error(a)
  else if (y ≥ −max_answer − x) a = x + y; else num_error(a);
  return a;
}
```

See also sections 1479 and 1481.

This code is used in section 1464.

1476. We know that *stretch_order*$(e) >$ *normal* implies *stretch*$(e) \neq 0$ and *shrink_order*$(e) >$ *normal* implies *shrink*$(e) \neq 0$.

⟨ Compute the sum or difference of two glue specs 1476 ⟩ ≡

```
  { width(e) = expr_a(width(e), width(t));
    if (stretch_order(e) ≡ stretch_order(t)) stretch(e) = expr_a(stretch(e), stretch(t));
    else if ((stretch_order(e) < stretch_order(t)) ∧ (stretch(t) ≠ 0)) { stretch(e) = stretch(t);
      stretch_order(e) = stretch_order(t);
    }
    if (shrink_order(e) ≡ shrink_order(t)) shrink(e) = expr_a(shrink(e), shrink(t));
    else if ((shrink_order(e) < shrink_order(t)) ∧ (shrink(t) ≠ 0)) { shrink(e) = shrink(t);
      shrink_order(e) = shrink_order(t);
    }
    delete_glue_ref(t);
    normalize_glue(e);
  }
```

This code is used in section 1474.

1477. If a multiplication is followed by a division, the two operations are combined into a 'scaling' operation. Otherwise the term t is multiplied by the factor f.

```
#define expr_m(A)  A = nx_plus_y(A, f, 0)
```

⟨ Cases for evaluation of the current term 1473 ⟩ +≡

```
case expr_mult:
  if (o ≡ expr_div) { n = f;
    o = expr_scale;
  }
  else if (l ≡ int_val) t = mult_integers(t, f);
  else if (l ≡ dimen_val) expr_m(t);
  else { expr_m(width(t));
    expr_m(stretch(t));
    expr_m(shrink(t));
  } break;
```

1478. Here we divide the term t by the factor f.

```
#define expr_d(A)  A = quotient(A, f)
```

⟨ Cases for evaluation of the current term 1473 ⟩ +≡

```
case expr_div:
  if (l < glue_val) expr_d(t);
  else { expr_d(width(t));
    expr_d(stretch(t));
    expr_d(shrink(t));
  } break;
```

1479. The function *quotient*(n, d) computes the rounded quotient $q = \lfloor n/d + \frac{1}{2} \rfloor$, when n and d are positive.

⟨ Declare subprocedures for *scan_expr* 1475 ⟩ +≡

```
static int quotient(int n, int d)
{ bool negative;   /* should the answer be negated? */
  int a;   /* the answer */
  if (d ≡ 0) num_error(a)
  else { if (d > 0) negative = false;
    else { negate(d);
      negative = true;
    }
    if (n < 0) { negate(n);
      negative = ¬negative;
    }
    a = n/d;
    n = n − a * d;
    d = n − d;   /* avoid certain compiler optimizations! */
    if (d + n ≥ 0) incr(a);
    if (negative) negate(a);
  }
  return a;
}
```

1480. Here the term t is multiplied by the quotient n/f.

#**define** *expr_s*(A) $A =$ *fract*(A, n, f, max_dimen)

⟨ Cases for evaluation of the current term 1473 ⟩ +≡

```
case expr_scale:
  if (l ≡ int_val) t = fract(t, n, f, infinity);
  else if (l ≡ dimen_val) expr_s(t);
  else { expr_s(width(t));
    expr_s(stretch(t));
    expr_s(shrink(t));
  }
```

1481. Finally, the function *fract*(x, n, d, max_answer) computes the integer $q = \lfloor xn/d + \frac{1}{2} \rfloor$, when x, n, and d are positive and the result does not exceed *max_answer*. We can't use floating point arithmetic since the routine must produce identical results in all cases; and it would be too dangerous to multiply by n and then divide by d, in separate operations, since overflow might well occur. Hence this subroutine simulates double precision arithmetic, somewhat analogous to METAFONT's *make_fraction* and *take_fraction* routines.

⟨Declare subprocedures for *scan_expr* 1475⟩ +≡

```
static int fract(int x, int n, int d, int max_answer)
{ bool negative;    /* should the answer be negated? */
  int a;    /* the answer */
  int f;    /* a proper fraction */
  int h;    /* smallest integer such that 2 * h ≥ d */
  int r;    /* intermediate remainder */
  int t;    /* temp variable */

  if (d ≡ 0) goto too_big;
  a = 0;
  if (d > 0) negative = false;
  else { negate(d);
    negative = true;
  }
  if (x < 0) { negate(x);
    negative = ¬negative;
  }
  else if (x ≡ 0) goto done;
  if (n < 0) { negate(n);
    negative = ¬negative;
  }
  t = n/d;
  if (t > max_answer/x) goto too_big;
  a = t * x;
  n = n − t * d;
  if (n ≡ 0) goto found;
  t = x/d;
  if (t > (max_answer − a)/n) goto too_big;
  a = a + t * n;
  x = x − t * d;
  if (x ≡ 0) goto found;
  if (x < n) { t = x;
    x = n;
    n = t;
  }    /* now 0 < n ≤ x < d */
  ⟨Compute f = ⌊xn/d + ½⌋ 1482⟩
  if (f > (max_answer − a)) goto too_big;
  a = a + f;
found:
  if (negative) negate(a);
  goto done;
too_big: num_error(a);
done: return a;
}
```

1482. The loop here preserves the following invariant relations between f, x, n, and r: (i) $f + \lfloor (xn + (r + d))/d \rfloor = \lfloor x_0 n_0/d + \frac{1}{2} \rfloor$; (ii) $-d \le r < 0 < n \le x < d$, where x_0, n_0 are the original values of x and n.

Notice that the computation specifies $(x - d) + x$ instead of $(x + x) - d$, because the latter could overflow.

⟨ Compute $f = \lfloor xn/d + \frac{1}{2} \rfloor$ 1482 ⟩ ≡

```
f = 0;
r = (d/2) − d;
h = −r;
loop { if (odd(n)) { r = r + x;
      if (r ≥ 0) { r = r − d;
        incr(f);
      }
    }
    n = n/2;
    if (n ≡ 0) goto found1;
    if (x < h) x = x + x;
    else { t = x − d;
      x = t + x;
      f = f + n;
      if (x < n) { if (x ≡ 0) goto found1;
        t = x;
        x = n;
        n = t;
      }
    }
}
found1:
```

This code is used in section 1481.

1483. The \gluestretch, \glueshrink, \gluestretchorder, and \glueshrinkorder commands return the stretch and shrink components and their orders of "infinity" of a glue specification.

```
#define glue_stretch_order_code (eTeX_int + 6)   /* code for \gluestretchorder */
#define glue_shrink_order_code (eTeX_int + 7)   /* code for \glueshrinkorder */
#define glue_stretch_code (eTeX_dim + 7)   /* code for \gluestretch */
#define glue_shrink_code (eTeX_dim + 8)   /* code for \glueshrink */
```

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡

```
primitive("gluestretchorder", last_item, glue_stretch_order_code);
primitive("glueshrinkorder", last_item, glue_shrink_order_code);
primitive("gluestretch", last_item, glue_stretch_code);
primitive("glueshrink", last_item, glue_shrink_code);
```

1484. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡

```
case glue_stretch_order_code: print_esc("gluestretchorder"); break;
case glue_shrink_order_code: print_esc("glueshrinkorder"); break;
case glue_stretch_code: print_esc("gluestretch"); break;
case glue_shrink_code: print_esc("glueshrink"); break;
```

1485. ⟨ Cases for fetching an integer value 1381 ⟩ +≡

case *glue_stretch_order_code*: **case** *glue_shrink_order_code*:
  { *scan_normal_glue*( );
    $q = cur\_val$;
    if ($m \equiv glue\_stretch\_order\_code$) $cur\_val = stretch\_order(q)$;
    else $cur\_val = shrink\_order(q)$;
    delete_glue_ref(q);
  }

1486. ⟨ Cases for fetching a dimension value 1401 ⟩ +≡

case *glue_stretch_code*: **case** *glue_shrink_code*:
  { *scan_normal_glue*( );
    $q = cur\_val$;
    if ($m \equiv glue\_stretch\_code$) $cur\_val = stretch(q)$;
    else $cur\_val = shrink(q)$;
    delete_glue_ref(q);
  }

1487. The `\mutoglue` and `\gluetomu` commands convert "math" glue into normal glue and vice versa; they allow to manipulate math glue with `\gluestretch` etc.

#**define** *mu_to_glue_code* *eTeX_glue* /* code for `\mutoglue` */
#**define** *glue_to_mu_code* *eTeX_mu* /* code for `\gluetomu` */

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
  primitive("mutoglue", *last_item*, *mu_to_glue_code*);
  primitive("gluetomu", *last_item*, *glue_to_mu_code*);

1488. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡

case *mu_to_glue_code*: *print_esc*("mutoglue"); **break**;
case *glue_to_mu_code*: *print_esc*("gluetomu"); **break**;

1489. ⟨ Cases for fetching a glue value 1489 ⟩ ≡

case *mu_to_glue_code*: *scan_mu_glue*( );

This code is used in section 1462.

1490. ⟨ Cases for fetching a mu value 1490 ⟩ ≡

case *glue_to_mu_code*: *scan_normal_glue*( );

This code is used in section 1462.

1491. ε-TeX (in extended mode) supports 32768 (i.e., 2^{15}) count, dimen, skip, muskip, box, and token registers. As in TeX the first 256 registers of each kind are realized as arrays in the table of equivalents; the additional registers are realized as tree structures built from variable-size nodes with individual registers existing only when needed. Default values are used for nonexistent registers: zero for count and dimen values, *zero_glue* for glue (skip and muskip) values, void for boxes, and *null* for token lists (and current marks discussed below).

Similarly there are 32768 mark classes; the command `\marks`n creates a mark node for a given mark class $0 \le n \le 32767$ (where `\marks0` is synonymous to `\mark`). The page builder (actually the *fire_up* routine) and the *vsplit* routine maintain the current values of *top_mark*, *first_mark*, *bot_mark*, *split_first_mark*, and *split_bot_mark* for each mark class. They are accessed as `\topmarks`n etc., and `\topmarks0` is again synonymous to `\topmark`. As in TeX the five current marks for mark class zero are realized as *cur_mark* array. The additional current marks are again realized as tree structure with individual mark classes existing only when needed.

⟨Generate all ε-TeX primitives 1379⟩ +≡
primitive("marks", *mark*, *marks_code*);
primitive("topmarks", *top_bot_mark*, *top_mark_code* + *marks_code*);
primitive("firstmarks", *top_bot_mark*, *first_mark_code* + *marks_code*);
primitive("botmarks", *top_bot_mark*, *bot_mark_code* + *marks_code*);
primitive("splitfirstmarks", *top_bot_mark*, *split_first_mark_code* + *marks_code*);
primitive("splitbotmarks", *top_bot_mark*, *split_bot_mark_code* + *marks_code*);

1492. The *scan_register_num* procedure scans a register number that must not exceed 255 in compatibility mode resp. 32767 in extended mode.

⟨Declare ε-TeX procedures for expanding 1434⟩ +≡
static void *scan_register_num*(**void**);

1493. ⟨Declare procedures that scan restricted classes of integers 432⟩ +≡
static void *scan_register_num*(**void**)
{ *scan_int*();
  if ((*cur_val* < 0) ∨ (*cur_val* > *max_reg_num*)) { *print_err*("Bad␣register␣code");
    help2(*max_reg_help_line*, "I␣changed␣this␣one␣to␣zero.");
    int_error(*cur_val*);
    cur_val = 0;
  }
}

1494. ⟨Initialize variables for ε-TeX compatibility mode 1494⟩ ≡
max_reg_num = 255;
max_reg_help_line = "A␣register␣number␣must␣be␣between␣0␣and␣255.";

This code is used in sections 1383 and 1385.

1495. ⟨Initialize variables for ε-TeX extended mode 1495⟩ ≡
max_reg_num = 32767;
max_reg_help_line = "A␣register␣number␣must␣be␣between␣0␣and␣32767.";

See also section 1540.

This code is used in sections 1378 and 1385.

1496. ⟨Global variables 13⟩ +≡
static halfword *max_reg_num*; /∗ largest allowed register number ∗/
static char ∗*max_reg_help_line*; /∗ first line of help message ∗/

1497. There are seven almost identical doubly linked trees, one for the sparse array of the up to 32512 additional registers of each kind and one for the sparse array of the up to 32767 additional mark classes. The root of each such tree, if it exists, is an index node containing 16 pointers to subtrees for 4096 consecutive array elements. Similar index nodes are the starting points for all nonempty subtrees for 4096, 256, and 16 consecutive array elements. These four levels of index nodes are followed by a fifth level with nodes for the individual array elements.

Each index node is nine words long. The pointers to the 16 possible subtrees or are kept in the *info* and *link* fields of the last eight words. (It would be both elegant and efficient to declare them as array, unfortunately Pascal doesn't allow this.)

The fields in the first word of each index node and in the nodes for the array elements are closely related. The *link* field points to the next lower index node and the *sa_index* field contains four bits (one hexadecimal digit) of the register number or mark class. For the lowest index node the *link* field is *null* and the *sa_index* field indicates the type of quantity (*int_val*, *dimen_val*, *glue_val*, *mu_val*, *box_val*, *tok_val*, or *mark_val*). The *sa_used* field in the index nodes counts how many of the 16 pointers are non-null.

The *sa_index* field in the nodes for array elements contains the four bits plus 16 times the type. Therefore such a node represents a count or dimen register if and only if $sa\_index < dimen\_val\_limit$; it represents a skip or muskip register if and only if $dimen\_val\_limit \le sa\_index < mu\_val\_limit$; it represents a box register if and only if $mu\_val\_limit \le sa\_index < box\_val\_limit$; it represents a token list register if and only if $box\_val\_limit \le sa\_index < tok\_val\_limit$; finally it represents a mark class if and only if $tok\_val\_limit \le sa\_index$.

The *new_index* procedure creates an index node (returned in *cur_ptr*) having given contents of the *sa_index* and *link* fields.

```
#define box_val 4   /* the additional box registers */
#define mark_val 6   /* the additional mark classes */
#define dimen_val_limit #20   /* 2^4 · (dimen_val + 1) */
#define mu_val_limit #40   /* 2^4 · (mu_val + 1) */
#define box_val_limit #50   /* 2^4 · (box_val + 1) */
#define tok_val_limit #60   /* 2^4 · (tok_val + 1) */
#define index_node_size 9   /* size of an index node */
#define sa_index(A) type(A)   /* a four-bit address or a type or both */
#define sa_used(A) subtype(A)   /* count of non-null pointers */

⟨Declare ε-TEX procedures for expanding 1434⟩ +≡
  static void new_index(quarterword i, pointer q)
  { int k;   /* loop index */
    cur_ptr = get_node(index_node_size);
    sa_index(cur_ptr) = i;
    sa_used(cur_ptr) = 0;
    link(cur_ptr) = q;
    for (k = 1; k ≤ index_node_size − 1; k++)   /* clear all 16 pointers */
      mem[cur_ptr + k] = sa_null;
  }
```

1498. The roots of the seven trees for the additional registers and mark classes are kept in the *sa_root* array. The first six locations must be dumped and undumped; the last one is also known as *sa_mark*.

```
#define sa_mark sa_root[mark_val]   /* root for mark classes */

⟨Global variables 13⟩ +≡
  static pointer sa_root0[mark_val − int_val + 1], *const sa_root = sa_root0 − int_val;
    /* roots of sparse arrays */
  static pointer cur_ptr;   /* value returned by new_index and find_sa_element */
  static memory_word sa_null;   /* two null pointers */
```

1499. ⟨ Set initial values of key variables 21 ⟩ +≡
$sa\_mark = null$;
$sa\_null.hh.lh = null$;
$sa\_null.hh.rh = null$;

1500. ⟨ Initialize table entries (done by `INITEX` only) 163 ⟩ +≡
for $(i = int\_val;\ i \le tok\_val;\ i{+}{+})$ $sa\_root[i] = null$;

1501. Given a type t and a sixteen-bit number n, the *find_sa_element* procedure returns (in *cur_ptr*) a pointer to the node for the corresponding array element, or *null* when no such element exists. The third parameter w is set *true* if the element must exist, e.g., because it is about to be modified. The procedure has two main branches: one follows the existing tree structure, the other (only used when w is *true*) creates the missing nodes.

We use macros to extract the four-bit pieces from a sixteen-bit register number or mark class and to fetch or store one of the 16 pointers from an index node.

```
#define if_cur_ptr_is_null_then_return_or_goto(A)    /* some tree element is missing */
        { if (cur_ptr ≡ null)
            if (w) goto A; else return;
        }
#define hex_dig1(A)  A/4096    /* the fourth lowest hexadecimal digit */
#define hex_dig2(A)  (A/256) % 16    /* the third lowest hexadecimal digit */
#define hex_dig3(A)  (A/16) % 16    /* the second lowest hexadecimal digit */
#define hex_dig4(A)  A % 16    /* the lowest hexadecimal digit */
#define get_sa_ptr
        if (odd(i)) cur_ptr = link(q + (i/2) + 1);
        else cur_ptr = info(q + (i/2) + 1)
          /* set cur_ptr to the pointer indexed by i from index node q */
#define put_sa_ptr(A)
        if (odd(i)) link(q + (i/2) + 1) = A;
        else info(q + (i/2) + 1) = A    /* store the pointer indexed by i in index node q */
#define add_sa_ptr
        { put_sa_ptr(cur_ptr);
          incr(sa_used(q));
        }    /* add cur_ptr as the pointer indexed by i in index node q */
#define delete_sa_ptr
        { put_sa_ptr(null);
          decr(sa_used(q));
        }    /* delete the pointer indexed by i in index node q */
```

```
⟨Declare ε-TeX procedures for expanding 1434⟩ +≡
  static void find_sa_element(small_number t, halfword n, bool w)
      /* sets cur_val to sparse array element location or null */
  { pointer q;    /* for list manipulations */
    small_number i;    /* a four bit index */
    cur_ptr = sa_root[t];
    if_cur_ptr_is_null_then_return_or_goto(not_found);
    q = cur_ptr;
    i = hex_dig1(n);
    get_sa_ptr;
    if_cur_ptr_is_null_then_return_or_goto(not_found1);
    q = cur_ptr;
    i = hex_dig2(n);
    get_sa_ptr;
    if_cur_ptr_is_null_then_return_or_goto(not_found2);
    q = cur_ptr;
    i = hex_dig3(n);
    get_sa_ptr;
    if_cur_ptr_is_null_then_return_or_goto(not_found3);
    q = cur_ptr;
    i = hex_dig4(n);
```

```
    get_sa_ptr;
    if ((cur_ptr ≡ null) ∧ w) goto not_found4;
    return;
  not_found: new_index(t, null);    /* create first level index node */
    sa_root[t] = cur_ptr;
    q = cur_ptr;
    i = hex_dig1(n);
  not_found1: new_index(i, q);    /* create second level index node */
    add_sa_ptr;
    q = cur_ptr;
    i = hex_dig2(n);
  not_found2: new_index(i, q);    /* create third level index node */
    add_sa_ptr;
    q = cur_ptr;
    i = hex_dig3(n);
  not_found3: new_index(i, q);    /* create fourth level index node */
    add_sa_ptr;
    q = cur_ptr;
    i = hex_dig4(n);
  not_found4: ⟨Create a new array element of type t with index i 1502⟩;
    link(cur_ptr) = q;
    add_sa_ptr;
  }
```

1502. The array elements for registers are subject to grouping and have an *sa_lev* field (quite analogous to *eq_level*) instead of *sa_used*. Since saved values as well as shorthand definitions (created by e.g., `\countdef`) refer to the location of the respective array element, we need a reference count that is kept in the *sa_ref* field. An array element can be deleted (together with all references to it) when its *sa_ref* value is *null* and its value is the default value.

Skip, muskip, box, and token registers use two word nodes, their values are stored in the *sa_ptr* field. Count and dimen registers use three word nodes, their values are stored in the *sa_int* resp. *sa_dim* field in the third word; the *sa_ptr* field is used under the name *sa_num* to store the register number. Mark classes use four word nodes. The last three words contain the five types of current marks

```
#define sa_lev sa_used /* grouping level for the current value */
#define pointer_node_size 2 /* size of an element with a pointer value */
#define sa_type(A) (sa_index(A)/16) /* type part of combined type/index */
#define sa_ref(A) info(A + 1) /* reference count of a sparse array element */
#define sa_ptr(A) link(A + 1) /* a pointer value */

#define word_node_size 3 /* size of an element with a word value */
#define sa_num(A) sa_ptr(A) /* the register number */
#define sa_int(A) mem[A + 2].i /* an integer */
#define sa_dim(A) mem[A + 2].sc /* a dimension (a somewhat esotheric distinction) */

#define mark_class_node_size 4 /* size of an element for a mark class */

#define fetch_box(A) /* fetch box(cur_val) */
        if (cur_val < 256) A = box(cur_val);
        else { find_sa_element(box_val, cur_val, false);
          if (cur_ptr ≡ null) A = null; else A = sa_ptr(cur_ptr);
        }

⟨Create a new array element of type t with index i 1502⟩ ≡
  if (t ≡ mark_val) /* a mark class */
  { cur_ptr = get_node(mark_class_node_size);
    mem[cur_ptr + 1] = sa_null;
    mem[cur_ptr + 2] = sa_null;
    mem[cur_ptr + 3] = sa_null;
  }
  else { if (t ≤ dimen_val) /* a count or dimen register */
    { cur_ptr = get_node(word_node_size);
      sa_int(cur_ptr) = 0;
      sa_num(cur_ptr) = n;
    }
    else { cur_ptr = get_node(pointer_node_size);
      if (t ≤ mu_val) /* a skip or muskip register */
      { sa_ptr(cur_ptr) = zero_glue;
        add_glue_ref(zero_glue);
      }
      else sa_ptr(cur_ptr) = null; /* a box or token list register */
    }
    sa_ref(cur_ptr) = null; /* all registers have a reference count */
  }
  sa_index(cur_ptr) = 16 * t + i; sa_lev(cur_ptr) = level_one
```

This code is used in section 1501.

1503. The *delete_sa_ref* procedure is called when a pointer to an array element representing a register is being removed; this means that the reference count should be decreased by one. If the reduced reference count is *null* and the register has been (globally) assigned its default value the array element should disappear, possibly together with some index nodes. This procedure will never be used for mark class nodes.

```
#define add_sa_ref(A) incr(sa_ref(A))   /* increase reference count */
#define change_box(A)   /* change box(cur_val), the eq_level stays the same */
        if (cur_val < 256) box(cur_val) = A; else set_sa_box(A)
#define set_sa_box(X)
        { find_sa_element(box_val, cur_val, false);
          if (cur_ptr ≠ null) { sa_ptr(cur_ptr) = X;
            add_sa_ref(cur_ptr);
            delete_sa_ref(cur_ptr);
          }
        }
```

⟨ Declare ε-TeX procedures for tracing and input 283 ⟩ +≡

```
  static void delete_sa_ref(pointer q)   /* reduce reference count */
  { pointer p;   /* for list manipulations */
    small_number i;   /* a four bit index */
    small_number s;   /* size of a node */
    decr(sa_ref(q));
    if (sa_ref(q) ≠ null) return;
    if (sa_index(q) < dimen_val_limit)
      if (sa_int(q) ≡ 0) s = word_node_size;
      else return;
    else { if (sa_index(q) < mu_val_limit)
        if (sa_ptr(q) ≡ zero_glue) delete_glue_ref(zero_glue);
        else return;
      else if (sa_ptr(q) ≠ null) return;
      s = pointer_node_size;
    }
    do { i = hex_dig4(sa_index(q));
      p = q;
      q = link(p);
      free_node(p, s);
      if (q ≡ null)   /* the whole tree has been freed */
      { sa_root[i] = null;
        return;
      }
      delete_sa_ptr;
      s = index_node_size;   /* node q is an index node */
    } while (¬(sa_used(q) > 0));
  }
```

1504. The *print_sa_num* procedure prints the register number corresponding to an array element.

⟨ Basic printing procedures 55 ⟩ +≡
static void *print_sa_num*(**pointer** q) /* print register number */
{ **halfword** n; /* the register number */
if (*sa_index*(q) < *dimen_val_limit*) n = *sa_num*(q); /* the easy case */
else { n = *hex_dig4*(*sa_index*(q));
q = *link*(q);
$n = n + 16 *$ *sa_index*(q);
q = *link*(q);
$n = n + 256 *$ (*sa_index*(q) $+ 16 *$ *sa_index*(*link*(q)));
}
print_int(n);
}

1505. Here is a procedure that displays the contents of an array element symbolically. It is used under similar circumstances as is *restore_trace* (together with *show_eqtb*) for the quantities kept in the *eqtb* array.

⟨Declare ε-TEX procedures for tracing and input 283⟩ +≡

```
#ifdef STAT
  static void show_sa(pointer p, char *s)
  { small_number t;    /* the type of element */
    begin_diagnostic();
    print_char('{');
    print(s);
    print_char('␣');
    if (p ≡ null) print_char('?');    /* this can't happen */
    else { t = sa_type(p);
      if (t < box_val) print_cmd_chr(internal_register, p);
      else if (t ≡ box_val) { print_esc("box");
        print_sa_num(p);
      }
      else if (t ≡ tok_val) print_cmd_chr(toks_register, p);
      else print_char('?');    /* this can't happen either */
      print_char('=');
      if (t ≡ int_val) print_int(sa_int(p));
      else if (t ≡ dimen_val) { print_scaled(sa_dim(p));
        print("pt");
      }
      else { p = sa_ptr(p);
        if (t ≡ glue_val) print_spec(p, "pt");
        else if (t ≡ mu_val) print_spec(p, "mu");
        else if (t ≡ box_val)
          if (p ≡ null) print("void");
          else { depth_threshold = 0;
            breadth_max = 1;
            show_node_list(p);
          }
        else if (t ≡ tok_val) { if (p ≠ null) show_token_list(link(p), null, 32);
        }
        else print_char('?');    /* this can't happen either */
      }
    }
    print_char('}');
    end_diagnostic(false);
  }
#endif
```

1506. Here we compute the pointer to the current mark of type t and mark class *cur_val*.

⟨Compute the mark pointer for mark type t and class *cur_val* 1506⟩ ≡

```
  { find_sa_element(mark_val, cur_val, false);
    if (cur_ptr ≠ null)
      if (odd(t)) cur_ptr = link(cur_ptr + (t/2) + 1);
      else cur_ptr = info(cur_ptr + (t/2) + 1);
  }
```

This code is used in section 385.

1507. The current marks for all mark classes are maintained by the *vsplit* and *fire_up* routines and are finally destroyed (for `INITEX` only) by the *final_cleanup* routine. Apart from updating the current marks when mark nodes are encountered, these routines perform certain actions on all existing mark classes. The recursive *do_marks* procedure walks through the whole tree or a subtree of existing mark class nodes and preforms certain actions indicted by its first parameter a, the action code. The second parameter l indicates the level of recursion (at most four); the third parameter points to a nonempty tree or subtree. The result is *true* if the complete tree or subtree has been deleted.

```
#define vsplit_init 0    /* action code for vsplit initialization */
#define fire_up_init 1    /* action code for fire_up initialization */
#define fire_up_done 2    /* action code for fire_up completion */
#define destroy_marks 3    /* action code for final_cleanup */
#define sa_top_mark(A) info(A + 1)    /* \topmarksn */
#define sa_first_mark(A) link(A + 1)    /* \firstmarksn */
#define sa_bot_mark(A) info(A + 2)    /* \botmarksn */
#define sa_split_first_mark(A) link(A + 2)    /* \splitfirstmarksn */
#define sa_split_bot_mark(A) info(A + 3)    /* \splitbotmarksn */
```

⟨ Declare the function called *do_marks* 1507 ⟩ ≡

```
  static bool do_marks(small_number a, small_number l, pointer q)
  { int i;    /* a four bit index */
    if (l < 4)    /* q is an index node */
    { for (i = 0; i ≤ 15; i++) { get_sa_ptr;
        if (cur_ptr ≠ null)
          if (do_marks(a, l + 1, cur_ptr)) delete_sa_ptr;
      }
      if (sa_used(q) ≡ 0) { free_node(q, index_node_size);
        q = null;
      }
    }
    else    /* q is the node for a mark class */
    { switch (a) { ⟨ Cases for do_marks 1508 ⟩
      }    /* there are no other cases */
      if (sa_bot_mark(q) ≡ null)
        if (sa_split_bot_mark(q) ≡ null) { free_node(q, mark_class_node_size);
          q = null;
        }
    }
    return (q ≡ null);
  }
```

This code is used in section 976.

1508. At the start of the *vsplit* routine the existing *split_fist_mark* and *split_bot_mark* are discarded.

⟨ Cases for *do_marks* 1508 ⟩ ≡

```
case vsplit_init:
  if (sa_split_first_mark(q) ≠ null) { delete_token_ref(sa_split_first_mark(q));
    sa_split_first_mark(q) = null;
    delete_token_ref(sa_split_bot_mark(q));
    sa_split_bot_mark(q) = null;
  } break;
```

See also sections 1510, 1511, and 1513.

This code is used in section 1507.

1509. We use again the fact that *split_first_mark* $\equiv$ *null* if and only if *split_bot_mark* $\equiv$ *null*.

⟨ Update the current marks for *vsplit* 1509 ⟩ ≡
  { *find_sa_element*(*mark_val*, *mark_class*(*p*), *true*);
    if (*sa_split_first_mark*(*cur_ptr*) ≡ *null*) { *sa_split_first_mark*(*cur_ptr*) = *mark_ptr*(*p*);
      add_token_ref(*mark_ptr*(*p*));
    }
    else *delete_token_ref*(*sa_split_bot_mark*(*cur_ptr*));
    sa_split_bot_mark(*cur_ptr*) = *mark_ptr*(*p*);
    add_token_ref(*mark_ptr*(*p*));
  }

This code is used in section 978.

1510. At the start of the *fire_up* routine the old *top_mark* and *first_mark* are discarded, whereas the old *bot_mark* becomes the new *top_mark*. An empty new *top_mark* token list is, however, discarded as well in order that mark class nodes can eventually be released. We use again the fact that *bot_mark* $\neq$ *null* implies *first_mark* $\neq$ *null*; it also knows that *bot_mark* $\equiv$ *null* implies *top_mark* $\equiv$ *first_mark* $\equiv$ *null*.

⟨ Cases for *do_marks* 1508 ⟩ +≡
case *fire_up_init*:
  if (*sa_bot_mark*(*q*) ≠ *null*) { **if** (*sa_top_mark*(*q*) ≠ *null*) *delete_token_ref*(*sa_top_mark*(*q*));
    delete_token_ref(*sa_first_mark*(*q*));
    sa_first_mark(*q*) = *null*;
    if (*link*(*sa_bot_mark*(*q*)) ≡ *null*)   /∗ an empty token list ∗/
    { *delete_token_ref*(*sa_bot_mark*(*q*));
      sa_bot_mark(*q*) = *null*;
    }
    else *add_token_ref*(*sa_bot_mark*(*q*));
    sa_top_mark(*q*) = *sa_bot_mark*(*q*);
  } **break**;

1511. ⟨ Cases for *do_marks* 1508 ⟩ +≡
case *fire_up_done*:
  if ((*sa_top_mark*(*q*) ≠ *null*) ∧ (*sa_first_mark*(*q*) ≡ *null*)) { *sa_first_mark*(*q*) = *sa_top_mark*(*q*);
    add_token_ref(*sa_top_mark*(*q*));
  } **break**;

1512. ⟨ Update the current marks for *fire_up* 1512 ⟩ ≡
  { *find_sa_element*(*mark_val*, *mark_class*(*p*), *true*);
    if (*sa_first_mark*(*cur_ptr*) ≡ *null*) { *sa_first_mark*(*cur_ptr*) = *mark_ptr*(*p*);
      add_token_ref(*mark_ptr*(*p*));
    }
    if (*sa_bot_mark*(*cur_ptr*) ≠ *null*) *delete_token_ref*(*sa_bot_mark*(*cur_ptr*));
    sa_bot_mark(*cur_ptr*) = *mark_ptr*(*p*);
    add_token_ref(*mark_ptr*(*p*));
  }

This code is used in section 1013.

1513. Here we use the fact that the five current mark pointers in a mark class node occupy the same locations as the the first five pointers of an index node. For systems using a run-time switch to distinguish between `VIRTEX` and `INITEX`, the codewords '**#ifdef** `INIT` ... **#endif**' surrounding the following piece of code should be removed.

⟨ Cases for *do_marks* 1508 ⟩ +≡
#ifdef `INIT`
case *destroy_marks*:
  for ($i = top\_mark\_code$; $i \le split\_bot\_mark\_code$; $i{+}{+}$) { *get_sa_ptr*;
    if ($cur\_ptr \neq null$) { *delete_token_ref*(*cur_ptr*);
      put_sa_ptr(*null*);
    }
  }
#endif

1514. The command code *internal_register* is used for '`\count`', '`\dimen`', etc., as well as for references to sparse array elements defined by '`\countdef`', etc.

⟨ Cases of **register** for *print_cmd_chr* 1514 ⟩ ≡
  { **if** $((chr\_code < mem\_bot) \lor (chr\_code > lo\_mem\_stat\_max))$ $cmd = sa\_type(chr\_code)$;
    else { $cmd = chr\_code - mem\_bot$;
      $chr\_code = null$;
    }
    if ($cmd \equiv int\_val$) *print_esc*("`count`");
    else if ($cmd \equiv dimen\_val$) *print_esc*("`dimen`");
    else if ($cmd \equiv glue\_val$) *print_esc*("`skip`");
    else *print_esc*("`muskip`");
    if ($chr\_code \neq null$) *print_sa_num*(*chr_code*);
  }

This code is used in section 411.

1515. Similarly the command code *toks_register* is used for '`\toks`' as well as for references to sparse array elements defined by '`\toksdef`'.

⟨ Cases of *toks_register* for *print_cmd_chr* 1515 ⟩ ≡
  { *print_esc*("`toks`");
    if ($chr\_code \neq mem\_bot$) *print_sa_num*(*chr_code*);
  }

This code is used in section 265.

1516. When a shorthand definition for an element of one of the sparse arrays is destroyed, we must reduce the reference count.

⟨ Cases for *eq_destroy* 1516 ⟩ ≡
case *toks_register*: **case** *internal_register*:
  if $((equiv\_field(w) < mem\_bot) \lor (equiv\_field(w) > lo\_mem\_stat\_max))$ *delete_sa_ref*(*equiv_field*(*w*));
    break;

This code is used in section 274.

1517. The task to maintain (change, save, and restore) register values is essentially the same when the register is realized as sparse array element or entry in *eqtb*. The global variable *sa_chain* is the head of a linked list of entries saved at the topmost level *sa_level*; the lists for lowel levels are kept in special save stack entries.

⟨Global variables 13⟩ +≡
  static pointer *sa_chain*; /* chain of saved sparse array entries */
  static quarterword *sa_level*; /* group level for *sa_chain* */

1518. ⟨Set initial values of key variables 21⟩ +≡
  sa_chain = *null*;
  sa_level = *level_zero*;

1519. The individual saved items are kept in pointer or word nodes similar to those used for the array elements: a word node with value zero is, however, saved as pointer node with the otherwise impossible *sa_index* value *tok_val_limit*.

#**define** *sa_loc*(*A*) *sa_ref*(*A*) /* location of saved item */

⟨Declare ε-TEX procedures for tracing and input 283⟩ +≡
  static void *sa_save*(**pointer** *p*) /* saves value of *p* */
  { **pointer** *q*; /* the new save node */
    quarterword *i*; /* index field of node */
    if (*cur_level* ≠ *sa_level*) { *check_full_save_stack*;
      save_type(*save_ptr*) = *restore_sa*;
      save_level(*save_ptr*) = *sa_level*;
      save_index(*save_ptr*) = *sa_chain*;
      incr(*save_ptr*);
      sa_chain = *null*;
      sa_level = *cur_level*;
    }
    i = *sa_index*(*p*);
    if (*i* < *dimen_val_limit*) { **if** (*sa_int*(*p*) ≡ 0) { *q* = *get_node*(*pointer_node_size*);
        i = *tok_val_limit*;
      }
      else { *q* = *get_node*(*word_node_size*);
        sa_int(*q*) = *sa_int*(*p*);
      }
      sa_ptr(*q*) = *null*;
    }
    else { *q* = *get_node*(*pointer_node_size*);
      sa_ptr(*q*) = *sa_ptr*(*p*);
    }
    sa_loc(*q*) = *p*;
    sa_index(*q*) = *i*;
    sa_lev(*q*) = *sa_lev*(*p*);
    link(*q*) = *sa_chain*;
    sa_chain = *q*;
    add_sa_ref(*p*);
  }

1520. ⟨Declare ε-TEX procedures for tracing and input 283⟩ +≡

static void *sa_destroy*(**pointer** *p*) /∗ destroy value of *p* ∗/
{ **if** (*sa_index*(*p*) < *mu_val_limit*) *delete_glue_ref*(*sa_ptr*(*p*));
  else if (*sa_ptr*(*p*) ≠ *null*)
    if (*sa_index*(*p*) < *box_val_limit*) *flush_node_list*(*sa_ptr*(*p*));
    else *delete_token_ref*(*sa_ptr*(*p*));
}

1521. The procedure *sa_def* assigns a new value to sparse array elements, and saves the former value if appropriate. This procedure is used only for skip, muskip, box, and token list registers. The counterpart of *sa_def* for count and dimen registers is called *sa_w_def*.

```
#define sa_define(A, B, C, D, E)
        if (e)
          if (global) gsa_def(A, B); else sa_def(A, B);
        else if (global) geq_define(C, D, E); else eq_define(C, D, E)
#define sa_def_box    /* assign cur_box to box(cur_val) */
        { find_sa_element(box_val, cur_val, true);
          if (global) gsa_def(cur_ptr, cur_box); else sa_def(cur_ptr, cur_box);
        }
#define sa_word_define(A, B)
        if (e)
          if (global) gsa_w_def(A, B); else sa_w_def(A, B);
        else word_define(A, B)
```

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡

```
  static void sa_def(pointer p, halfword e)    /* new data for sparse array elements */
  { add_sa_ref(p);
    if (sa_ptr(p) ≡ e) {
#ifdef STAT
      if (tracing_assigns > 0) show_sa(p, "reassigning");
#endif
      sa_destroy(p);
    }
    else {
#ifdef STAT
      if (tracing_assigns > 0) show_sa(p, "changing");
#endif
      if (sa_lev(p) ≡ cur_level) sa_destroy(p); else sa_save(p);
      sa_lev(p) = cur_level;
      sa_ptr(p) = e;
#ifdef STAT
      if (tracing_assigns > 0) show_sa(p, "into");
#endif
    }
    delete_sa_ref(p);
  }
  static void sa_w_def(pointer p, int w)
  { add_sa_ref(p);
    if (sa_int(p) ≡ w) {
#ifdef STAT
      if (tracing_assigns > 0) show_sa(p, "reassigning");
#endif
    }
    else {
#ifdef STAT
      if (tracing_assigns > 0) show_sa(p, "changing");
#endif
      if (sa_lev(p) ≠ cur_level) sa_save(p);
      sa_lev(p) = cur_level;
      sa_int(p) = w;
```

```
#ifdef STAT
    if (tracing_assigns > 0) show_sa(p, "into");
#endif
  }
  delete_sa_ref(p);
}
```

1522. The *sa_def* and *sa_w_def* routines take care of local definitions. Global definitions are done in almost the same way, but there is no need to save old values, and the new value is associated with *level_one*.

```
⟨Declare ε-TEX procedures for tracing and input 283⟩ +≡
static void gsa_def(pointer p, halfword e)    /* global sa_def */
{ add_sa_ref(p);
#ifdef STAT
  if (tracing_assigns > 0) show_sa(p, "globally␣changing");
#endif
  sa_destroy(p);
  sa_lev(p) = level_one;
  sa_ptr(p) = e;
#ifdef STAT
  if (tracing_assigns > 0) show_sa(p, "into");
#endif
  delete_sa_ref(p);
}
static void gsa_w_def(pointer p, int w)    /* global sa_w_def */
{ add_sa_ref(p);
#ifdef STAT
  if (tracing_assigns > 0) show_sa(p, "globally␣changing");
#endif
  sa_lev(p) = level_one;
  sa_int(p) = w;
#ifdef STAT
  if (tracing_assigns > 0) show_sa(p, "into");
#endif
  delete_sa_ref(p);
}
```

1523. The *sa_restore* procedure restores the sparse array entries pointed at by *sa_chain*

⟨ Declare ε-TEX procedures for tracing and input 283 ⟩ +≡

```
static void sa_restore(void)
{ pointer p;     /* sparse array element */
  do { p = sa_loc(sa_chain);
    if (sa_lev(p) ≡ level_one) { if (sa_index(p) ≥ dimen_val_limit) sa_destroy(sa_chain);
#ifdef STAT
      if (tracing_restores > 0) show_sa(p, "retaining");
#endif
    }
    else { if (sa_index(p) < dimen_val_limit)
          if (sa_index(sa_chain) < dimen_val_limit) sa_int(p) = sa_int(sa_chain);
          else sa_int(p) = 0;
      else { sa_destroy(p);
          sa_ptr(p) = sa_ptr(sa_chain);
      }
      sa_lev(p) = sa_lev(sa_chain);
#ifdef STAT
      if (tracing_restores > 0) show_sa(p, "restoring");
#endif
    }
    delete_sa_ref(p);
    p = sa_chain;
    sa_chain = link(p);
    if (sa_index(p) < dimen_val_limit) free_node(p, word_node_size);
    else free_node(p, pointer_node_size);
  } while (¬(sa_chain ≡ null));
}
```

1524. When reading `\patterns` while `\savinghyphcodes` is positive the current *lc_code* values are stored together with the hyphenation patterns for the current language. They will later be used instead of the *lc_code* values for hyphenation purposes.

The *lc_code* values are stored in the linked trie analogous to patterns p_1 of length 1, with *hyph_root* ≡ *trie_r*[0] replacing *trie_root* and *lc_code*(*p_1*) replacing the *trie_op* code. This allows to compress and pack them together with the patterns with minimal changes to the existing code.

```
#define hyph_root  trie_r[0]     /* root of the linked trie for hyph_codes */
```

⟨ Initialize table entries (done by INITEX only) 163 ⟩ +≡

```
  hyph_root = 0;
  hyph_start = 0;
```

1525. ⟨ Store hyphenation codes for current language 1525 ⟩ ≡

{ $c = cur\_lang$;
  $first\_child = false$;
  $p = 0$;
  do { $q = p$;
    $p = trie\_r[q]$;
  } **while** $(\neg((p \equiv 0) \lor (c \le so(trie\_c[p]))))$;
  if $((p \equiv 0) \lor (c < so(trie\_c[p])))$ ⟨ Insert a new trie node between q and p, and make p point to it 963 ⟩;
  $q = p$;   /* now node q represents *cur_lang* */
  ⟨ Store all current *lc_code* values 1526 ⟩;
}

This code is used in section 959.

1526. We store all nonzero *lc_code* values, overwriting any previously stored values (and possibly wasting a few trie nodes that were used previously and are not needed now). We always store at least one *lc_code* value such that *hyph_index* (defined below) will not be zero.

⟨ Store all current *lc_code* values 1526 ⟩ ≡

  $p = trie\_l[q]$;
  $first\_child = true$;
  for $(c = 0;\ c \le 255;\ c{+}{+})$
    if $((lc\_code(c) > 0) \lor ((c \equiv 255) \land first\_child))$ { **if** $(p \equiv 0)$
        ⟨ Insert a new trie node between q and p, and make p point to it 963 ⟩
      else $trie\_c[p] = si(c)$;
      $trie\_o[p] = qi(lc\_code(c))$;
      $q = p$;
      $p = trie\_r[q]$;
      $first\_child = false$;
    }
  if $(first\_child)$ $trie\_l[q] = 0$; **else** $trie\_r[q] = 0$

This code is used in section 1525.

1527. We must avoid to "take" location 1, in order to distinguish between *lc_code* values and patterns.

⟨ Pack all stored *hyph_codes* 1527 ⟩ ≡

{ **if** $(trie\_root \equiv 0)$
    for $(p = 0;\ p \le 255;\ p{+}{+})$ $trie\_min[p] = p + 2$;
  $first\_fit(hyph\_root)$;
  $trie\_pack(hyph\_root)$;
  $hyph\_start = trie\_ref[hyph\_root]$;
}

This code is used in section 965.

1528. The global variable *hyph_index* will point to the hyphenation codes for the current language.

#define *set_hyph_index* /* set *hyph_index* for current language */
  if $(trie\_char(hyph\_start + cur\_lang) \neq qi(cur\_lang))$ $hyph\_index = 0$;
    /* no hyphenation codes for *cur_lang* */
  else $hyph\_index = trie\_link(hyph\_start + cur\_lang)$

#define *set_lc_code*(*A*) /* set *hc*[0] to hyphenation or lc code for *A* */
  if $(hyph\_index \equiv 0)$ $hc[0] = lc\_code(A)$;
  else if $(trie\_char(hyph\_index + A) \neq qi(A))$ $hc[0] = 0$;
  else $hc[0] = qo(trie\_op(hyph\_index + A))$

⟨ Global variables 13 ⟩ +≡
  static trie_pointer *hyph_start*; /* root of the packed trie for *hyph_codes* */
  static trie_pointer *hyph_index*; /* pointer to hyphenation codes for *cur_lang* */

1529. When *saving_vdiscards* is positive then the glue, kern, and penalty nodes removed by the page builder or by `\vsplit` from the top of a vertical list are saved in special lists instead of being discarded.

#define *tail_page_disc* *disc_ptr*[*copy_code*] /* last item removed by page builder */
#define *page_disc* *disc_ptr*[*last_box_code*] /* first item removed by page builder */
#define *split_disc* *disc_ptr*[*vsplit_code*] /* first item removed by `\vsplit` */

⟨ Global variables 13 ⟩ +≡
  static pointer $disc\_ptr0[vsplit\_code - copy\_code + 1]$, ∗**const** $disc\_ptr = disc\_ptr0 - copy\_code$;
    /* list pointers */

1530. ⟨ Set initial values of key variables 21 ⟩ +≡
  $page\_disc = null$;
  $split\_disc = null$;

1531. The `\pagediscards` and `\splitdiscards` commands share the command code *un_vbox* with `\unvbox` and `\unvcopy`, they are distinguished by their *chr_code* values *last_box_code* and *vsplit_code*. These *chr_code* values are larger than *box_code* and *copy_code*.

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
  primitive(`"pagediscards"`, *un_vbox*, *last_box_code*);
  primitive(`"splitdiscards"`, *un_vbox*, *vsplit_code*);

1532. ⟨ Cases of *un_vbox* for *print_cmd_chr* 1532 ⟩ ≡
  ; **else**
  if $(chr\_code \equiv last\_box\_code)$ *print_esc*(`"pagediscards"`);
  else if $(chr\_code \equiv vsplit\_code)$ *print_esc*(`"splitdiscards"`)

This code is used in section 1107.

1533. ⟨ Handle saved items and **goto** *done* 1533 ⟩ ≡
  { $link(tail) = disc\_ptr[cur\_chr]$;
    $disc\_ptr[cur\_chr] = null$;
    goto *done*;
  }

This code is used in section 1109.

1534. The `\interlinepenalties`, `\clubpenalties`, `\widowpenalties`, and `\displaywidowpenalties` commands allow to define arrays of penalty values to be used instead of the corresponding single values.

#define *inter_line_penalties_ptr* *equiv*(*inter_line_penalties_loc*)

⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
primitive("interlinepenalties", *set_shape*, *inter_line_penalties_loc*);
primitive("clubpenalties", *set_shape*, *club_penalties_loc*);
primitive("widowpenalties", *set_shape*, *widow_penalties_loc*);
primitive("displaywidowpenalties", *set_shape*, *display_widow_penalties_loc*);

1535. ⟨ Cases of *set_shape* for *print_cmd_chr* 1535 ⟩ ≡
case *inter_line_penalties_loc*: *print_esc*("interlinepenalties"); **break**;
case *club_penalties_loc*: *print_esc*("clubpenalties"); **break**;
case *widow_penalties_loc*: *print_esc*("widowpenalties"); **break**;
case *display_widow_penalties_loc*: *print_esc*("displaywidowpenalties");
This code is used in section 265.

1536. ⟨ Fetch a penalties array element 1536 ⟩ ≡
{ *scan_int*( );
  if $((equiv(m) \equiv null) \lor (cur\_val < 0))$ $cur\_val = 0$;
  else { **if** $(cur\_val > penalty(equiv(m)))$ $cur\_val = penalty(equiv(m))$;
    $cur\_val = penalty(equiv(m) + cur\_val)$;
  }
}
This code is used in section 422.

1537. *expand_depth* and *expand_depth_count* are used in the ε-TEX code above, but not defined. So we correct this in the following modules, *expand_depth* having been defined by us as an integer paramater (hence there is a new primitive to create in ε-TEX mode), and *expand_depth_count* needing to be a global. Both have to be defined to some sensible value.

⟨ Global variables 13 ⟩ +≡
static int *expand_depth_count*; /∗ current expansion depth ∗/

1538. ⟨ Generate all ε-TEX primitives 1379 ⟩ +≡
primitive("expanddepth", *assign_int*, *int_base* + *expand_depth_code*);

1539. ⟨ Cases for *print_param* 1389 ⟩ +≡
case *expand_depth_code*: *print_esc*("expanddepth"); **break**;

1540. ⟨ Initialize variables for ε-TEX extended mode 1495 ⟩ +≡
$expand\_depth = 10000$; /∗ value taken for compatibility with Web2C ∗/
$expand\_depth\_count = 0$;

1541. The extended features of PRŌTE. PRŌTE extends furthermore ε-TEX i.e. ε-TEX is thus required before adding PRŌTE own extensions. But if ε-TEX mode has not be enabled, the engine is still compatible with TEXwith no added primitive commands and with a modification of code—from ε-TEX exclusively for now—that is sufficiently minor so that the engine still deserves the name TEX.

#define *Prote_ex* (*Prote_mode* ≡ 1) /* is this prote mode? */

⟨ Global variables 13 ⟩ +≡
static int *Prote_mode*; /* to be or not to be; but an int to dump */

1542. We begin in TEX compatibility mode. The state *Prote_mode* will be set to 1 only if activated by the supplementary '*' added to the one activating the ε-TEX extensions (in fact, this means for the user two initial '*' in a row).

⟨ Initialize table entries (done by `INITEX` only) 163 ⟩ +≡
Prote_mode = 0; /* initially we are in compatibility mode */

1543. ⟨ Dump the PRŌTE state 1543 ⟩ ≡
dump_int(*Prote_mode*);

This code is used in section 1306.

1544. ⟨ Undump the PRŌTE state 1544 ⟩ ≡
undump(0, 1, *Prote_mode*);

This code is used in section 1307.

1545. In order to not clobber the global scope with variables that are locally used, the initializations for PRŌTE, if the mode is activated, are done in a dedicated procedure. These are not part of what is dumped.

⟨ Last-minute procedures 1332 ⟩ +≡
static void *Prote_initialize*(**void**)
{ **int** *k*; /* all-purpose index */
⟨ PRŌTE initializations 1567 ⟩;
}

1546. There are commands and command modifiers, these command modifiers maybe encoding too a type. So we must not step on each other toes.

1547. When we are adding primitives that deal intimately with the variables of TEX, in the *eqtb* regions (in our case regions 5 for integers, and 6 for dimensions), the command modifier to the various `assign_*` classes is simply the address. So we have interpolated our added variables above since this is done by the way of WEB pre-processing.

1548. For the conditional primitives, the way is straightforward.

#define *if_incsname_code* (*eTeX_last_if_test_cmd_mod* + 1) /* '`\ifincsname`' */
#define *if_primitive_code* (*eTeX_last_if_test_cmd_mod* + 2) /* '`\ifprimitive`' */

1549. The *last_item* class is for secondary internal values, that can be dereferenced by `\the` but are read-only and are mainly related to the value of a current state or are such values but their assignation shall trigger an action, and we shall not hook in the `assign_*` processing.

The command modifiers for the *last_item* class were, originally, encoding too the type of the item (see m.410). But ε-TEX has added its extensions and we won't try to be smart: the type *cur_val_level* will be set by switching between contiguous ranges of values of the same type.

And we will define here all the instances of *last_item* that we add in order to keep our number assignations gathered.

#define *Prote_version_code* (*eTeX_last_last_item_cmd_mod* + 1) /∗ code for `\Proteversion` ∗/
#define *random_seed_code* (*eTeX_last_last_item_cmd_mod* + 2) /∗ `\randomseed` ∗/
#define *elapsed_time_code* (*eTeX_last_last_item_cmd_mod* + 3) /∗ `\elapsedtime` ∗/
#define *shell_escape_code* (*eTeX_last_last_item_cmd_mod* + 4) /∗ `\shellescape` ∗/
#define *last_xpos_code* (*eTeX_last_last_item_cmd_mod* + 5) /∗ `\lastxpos` ∗/
#define *last_ypos_code* (*eTeX_last_last_item_cmd_mod* + 6) /∗ `\lastypos` ∗/

⟨ Fetch a PRōTE item 1549 ⟩ ≡
{ **switch** (*m*) {
⟨ Cases for fetching a PRōTE int value 1555 ⟩
} /∗ there are no other cases ∗/
cur_val_level = *int_val*;
}

This code is used in section 423.

1550. The `convert` class is for conversion of some external stuff to put it, as a token list, into the scanner. It is not an internal value that could be dereferenced by `\the` and it is obviously not settable: it expands to the token list.

#define *Prote_revision_code* (*eTeX_last_convert_cmd_mod* + 1) /∗ `\Proterevision` ∗/
#define *strcmp_code* (*eTeX_last_convert_cmd_mod* + 2) /∗ `\strcmp` ∗/
#define *set_random_seed_code* (*eTeX_last_convert_cmd_mod* + 3) /∗ `\setrandomseed` ∗/
#define *normal_deviate_code* (*eTeX_last_convert_cmd_mod* + 4) /∗ `\normaldeviate` ∗/
#define *uniform_deviate_code* (*eTeX_last_convert_cmd_mod* + 5) /∗ `\uniformdeviate` ∗/
#define *creation_date_code* (*eTeX_last_convert_cmd_mod* + 6) /∗ `\creationdate` ∗/
#define *file_size_code* (*eTeX_last_convert_cmd_mod* + 7) /∗ `\filesize` ∗/
#define *file_mod_date_code* (*eTeX_last_convert_cmd_mod* + 8) /∗ `\filemodedate` ∗/
#define *file_dump_code* (*eTeX_last_convert_cmd_mod* + 9) /∗ `\filedump` ∗/
#define *mdfive_sum_code* (*eTeX_last_convert_cmd_mod* + 10) /∗ `\mdfivesum` ∗/

1551. When modifying the meaning of something—in this case, for now, switching to the primitive meaning if it exists—or modifying the way expansion is done, it seems that it can be thought as a special case of expansion, hence a variant of *expand_after*.

#define *primitive_code* (*eTeX_last_expand_after_cmd_mod* + 1) /∗ '`\primitive`' ∗/
#define *expanded_code* (*eTeX_last_expand_after_cmd_mod* + 2) /∗ '`\expanded`' ∗/

1552. When the primitive manipulate something really external, whether trying to insert something in the output format—`DVI` for us—or dealing with the system, it doesn't fit in any cmd group and could be called an exception. So it will be a variant of the *extension* cmd group.

ε-TEX didn't add new primitives to the extension command group, so we add a related macro, equal to *TeX_last_extension_cmd_mod*, simply so that it is locally obvious.

#define *eTeX_last_extension_cmd_mod* *TeX_last_extension_cmd_mod* /∗ none added ∗/
#define *reset_timer_code* (*eTeX_last_extension_cmd_mod* + 1) /∗ '`\resettimer`' ∗/
#define *save_pos_code* (*eTeX_last_extension_cmd_mod* + 2) /∗ '`\savepos`' ∗/

1553. Identifying PRoTE.

We will start by giving a mean to test that PRoTE is activated and to identify the version.

⟨ Generate all PRoTE primitives 1553 ⟩ ≡
primitive("Proteversion", *last_item*, *Prote_version_code*);
primitive("Proterevision", *convert*, *Prote_revision_code*);

See also sections 1568, 1571, 1578, 1588, 1591, 1597, 1602, 1609, 1613, 1617, 1621, 1639, 1643, 1650, 1657, 1662, 1666, and 1671.

This code is used in section 1378.

1554. We use the different hooks added to insert our cases.

⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡
case *Prote_version_code*: *print_esc*("Proteversion"); **break**;

1555. ⟨ Cases for fetching a PRoTE int value 1555 ⟩ ≡
case *Prote_version_code*: *cur_val* = *Prote_version*; **break**;

See also sections 1570, 1605, 1641, and 1668.

This code is used in section 1549.

1556. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ ≡
case *Prote_revision_code*: *print_esc*("Proterevision"); **break**;

See also sections 1592, 1598, 1610, 1614, 1618, 1622, 1644, 1651, and 1658.

This code is used in section 468.

1557. ⟨ Cases of 'Scan the argument for command c' 1557 ⟩ ≡
case *Prote_revision_code*: *do_nothing*; **break**;

See also sections 1593, 1599, 1611, 1615, 1619, 1623, 1645, 1652, and 1659.

This code is used in section 470.

1558. ⟨ Cases of 'Print the result of command c' 1558 ⟩ ≡
case *Prote_revision_code*: *print*(*Prote_revision*); **break**;

See also sections 1594, 1600, 1612, 1616, 1620, 1624, 1646, 1653, and 1660.

This code is used in section 471.

1559. PRoTE added token lists routines.

We will, more than once, convert a general normally expanded text to a string. Due to the unfelicity of Pascal about forward declarations of functions, we declare procedures that do their task by defining global variables. In this case, *garbage* is used.

link(*garbage*) will hold the pointer to the head of the token list, *info*(*garbage*) to the tail. If the two are equals, then the list is empty. The routine making a string will take *link*(*garbage*) and put the number in *info*(*garbage*).

1560. The first procedure scan a general text (normally) expanded. The head of the reference count is returned in *link*(*garbage*), the tail in *info*(*garbage*) and if the two are equals, the list is empty. User must keep in mind that this has to be flushed when done with!

⟨Forward declarations 52⟩ +≡

```
static void scan_general_x_text(void);
```

1561. ⟨Declare PRoTE procedures for token lists 1561⟩ ≡

```
static void scan_general_x_text(void)
{ pointer d;    /* to save def_ref */
  d = def_ref;
  info(garbage) = scan_toks(false, true);
  link(garbage) = def_ref;
  def_ref = d;    /* restore whatever */
}
```

See also section 1563.

This code is used in section 472.

1562. The second procedure takes a token list defined in *link*(*garbage*) and converts it to a string number that is returned in *info*(*garbage*). Neither the token list nor the string (obviously) are flushed.

⟨Forward declarations 52⟩ +≡

```
static void toks_to_str(void);
```

1563. Here we are using *token_show* that has to take a reference count.

⟨Declare PRoTE procedures for token lists 1561⟩ +≡

```
static void toks_to_str(void)
{ int old_setting;    /* holds selector setting */
  old_setting = selector;
  selector = new_string;
  token_show(link(garbage));
  selector = old_setting;
  str_room(1);    /* flirting with the limit means probably truncation */
  info(garbage) = make_string();
}
```

1564. PRoTE added strings routines.

The next procedure sets *name_of_file* from the string given as an argument, mimicking the *input* primitive by adding an . *tex* extension if there is none. It silently truncates if the length of the string exceeds the size of the name buffer and doesn't use *cur_area* and *cur_ext*, but *name_length* is set to the real name length (without truncating) so a test about $k \le file\_name\_size$ allows to detect the impossibility of opening the file without having to call external code. The string is not flushed: it is the responsability of the code calling the procedure to flush it if wanted.

⟨ Declare PRoTE procedures for strings 1564 ⟩ ≡

```
static void str_to_name(str_number s)
{ int k;    /* number of positions filled in name_of_file */
  ASCII_code c;    /* character being packed */
  int j;    /* index into str_pool */
  k = 0;
  for (j = str_start[s]; j <= str_start[s + 1] - 1; j++) { c = so(str_pool[j]);
    incr(k);
    if (k <= file_name_size) name_of_file[k] = xchr[c];
  }
  name_length = k;
  name_of_file[name_length + 1] = 0;
}
```

This code is used in section 46.

1565. Exchanging data with external routines.

In order to try to sever external handling from our core, we introduce an all purpose exchange buffer *xchg_buffer*, that will be an array of bytes (these can be interpreted as `text_char` or `ASCII_char` or `eight_bits`).

The data to be used starts at index 1 and ends at index *xchg_buffer_length*.

For the moment, this buffer must accommodate a numerical MD5 hash value, i.e. 16 bytes long; will also be used to exchange 64 bytes chunks to feed MD5 hash generation, and will have to accommodate too the maximal size of the date returned by `\creationdate` or `\filemoddate` that is 23 `text_char`. So at least 64 for now.

⟨ Global variables 13 ⟩ +≡
static eight_bits *xchg_buffer0* [*xchg_buffer_size*], ∗**const** *xchg_buffer* = *xchg_buffer0* − 1;
/∗ exchange buffer for interaction with system routines ∗/
static int *xchg_buffer_length*; /∗ last valid index in this buf; 0 means no data ∗/

1566. ⟨ Check PRoTE "constant" values for consistency 1566 ⟩ ≡
if (*xchg_buffer_size* < 64) *bad* = 51;

This code is used in section 1378.

1567. When there is data in the exchange buffer, the length of the data has to be set. When an external routine has consumed the data, it shall reset the length to 0.

⟨ PRoTE initializations 1567 ⟩ ≡
xchg_buffer_length = 0;

See also sections 1573, 1627, 1642, 1664, and 1670.

This code is used in section 1545.

1568. PRoTE **states.**

\shellescape depends on a pdfTEX feature, namely the ability to escape to shell. There is no such thing in PRoTE. So it expands to 0. Note: this a status primitive; it does not allow to set the status but simply expands to a read-only integer reflecting it. In PRoTE, it is always 0.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
primitive("shellescape", *last_item*, *shell_escape_code*);

1569. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡
case *shell_escape_code*: *print_esc*("shellescape"); **break**;

1570. ⟨ Cases for fetching a PRoTE int value 1555 ⟩ +≡
case *shell_escape_code*: *cur_val* = 0; **break**;

1571. PRoTE **conditionals.**

We add the following conditionals, that are susceptible of the same expansion rules as the other *if_test* ones.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
  primitive(`"ifincsname"`, *if_test*, *if_incsname_code*);
  primitive(`"ifprimitive"`, *if_test*, *if_primitive_code*);

1572. ⟨ Cases of *if_test* for *print_cmd_chr* 1446 ⟩ +≡
case *if_incsname_code*: *print_esc*(`"ifincsname"`); **break**;
case *if_primitive_code*: *print_esc*(`"ifprimitive"`); **break**;

1573. The conditional `\ifincsname` is simple since we increment a global variable *incsname_state* when we enter the `\csname` command and decrement it when we have reached and passed the `\endcsname`—a scope depth index.

⟨ PRoTE initializations 1567 ⟩ +≡
  $incsname\_state = 0$;

1574. ⟨ Cases for *conditional* 1448 ⟩ +≡
case *if_incsname_code*: $b = (incsname\_state > 0)$; **break**;

1575. The conditional `\ifprimitive` is true when the following control sequence is a primitive; false otherwise. *id_lookup* can return *undefined_control_sequence* (for a control sequence not entered in the hash since *no_new_control_sequence* is *true*), but since it has the `eq_type` set to *undefined_cs*, the test of this latter works as for a control sequence entered but never defined.

1576. ⟨ Cases for *conditional* 1448 ⟩ +≡
case *if_primitive_code*:
  { **do** { *get_token*( );
    } **while** $(\neg(cur\_tok \neq space\_token))$;
    if $((cur\_cs \neq 0) \land (cur\_cmd \neq undefined\_cs) \land (cur\_cmd < call))$ $b = true$;
    else $b = false$;
  } **break**;

1577. PRoTE **primitives changing definition or expansion.**
The next primitives, here, are more involved since they are whether changing the definition of a control sequence, or modifying how the tokens will be treated.

1578. Since a user level control sequence can give a new definition to a primitive, the `primitive`... primitive, if the argument is a control sequence whose name is the name of a primitive, will make this primitive meaning the meaning of the control sequence *hic et nunc*. If there was no primitive meaning, no error is raised and nothing is changed. It can be seen as a kind of `expand_after` command since it is in the external handling of the token list creation.

Since we need to redefine the token and hence give a valid control sequence in the *eqtb*, we have defined *frozen_primitive*. This "frozen" is, actually, not quite frozen by itself since we will redefine its values according to the primitive definition we have to restablish momentarily. But it is indeed "permanent" since it only refers to the permanently defined meanings. Hence, the initialization of the *frozen_primitive* address is just to document the code: these values will be overwritten on each actual call.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
primitive("`primitive`", *expand_after*, *primitive_code*);
text(*frozen_primitive*) = *text*(*cur_val*);
eqtb[*frozen_primitive*] = *eqtb*[*cur_val*];

1579. ⟨ Cases of *expandafter* for *print_cmd_chr* 1445 ⟩ +≡
case *primitive_code*: *print_esc*("`primitive`"); **break**;

1580. The problem is that the primitives are added at *level_one* and that a redefinition as a macro at this same level by a user simply overwrites the definition. We need then to keep these definitions.

Primitives are only added by `INITEX`. So we can consider what we will call a `ROM`, since it can be only "flashed" by `INITEX` and is read-only afterwards, a kind of BIOS table holding initial system calls (primitives).

Since primitives are not macros (they don't need to expand or to evaluate parameters since their definition is directly in the code), the definition of a primitive is a couple: the command class (*cur_cmd*) and the modifier (*cur_chr*) to distinguish between the cases—the instances. But since, at the user level, a primitive is identified by its name, and that a redefinition is, mandatorily, a homonym, the location of the macro shadowing the primitive is at the same address as was the primitive in the *eqtb*. So in order to speed-up the check, we should organize things so that the address in the *eqtb* of a control sequence (one character or multiletter) can be readily converted in an address in the `ROM` array.

This array will be an array of memory word, of type **two_halves**, in order to re-use the macro definitions set for the table of equivalents.

The one character primitives are added by direct addressing relative to *single_base*. The multiletter primitives are added starting at *frozen_control_sequence* -1, downwards; but there are only, at the moment, 322 multileter primitives defined by TEX, 78 such primitives defined by ε-TEX, and we are adding 24 more. It is clear that, looking at primitives, region 2 of *eqtb* is really a sparse array and that, when *hash_size* is increased for format needs, there will be a fair amount of space wasted if we simply copy, in fact, second part of region 1 and region 2 in the `ROM`.

Yes, but it is simpler as a first approach—premature optimization is the root of all evil. So a simple translation scheme will be enough.

The index in `ROM` will start at 1 and will go up to $256 + 1 + hash\_size$, that is a simple translation from *single_base* to *ROM_base*, but only for addresses of interest, the other pointing to an *ROM_undefined_primitive* that will allow an easy test.

#define *ROM_base* 1
#define *ROM_size* $(256 + 1 + hash\_size)$ /∗ 256 oc, undefined and ml ∗/
#define *ROM_undefined_primitive* 257
#define *ROM_type_field*(*A*) *A.hh.b0*
#define *ROM_equiv_field*(*X*) *X.hh.rh*
#define *ROM_type*(*A*) *ROM_type_field*(`ROM`[*A*]) /∗ command code for equivalent ∗/
#define *set_ROM_p_from_cs*(*A*)
  if $((A \ge single\_base) \land (A < frozen\_control\_sequence))$ $p = A - single\_base + ROM\_base$;
  else $p = ROM\_undefined\_primitive$

⟨ Global variables 13 ⟩ +≡
 static memory_word `ROM0`[$ROM\_size - ROM\_base + 1$], ∗**const** `ROM` = `ROM0` − *ROM_base*;

1581. Even if it will be unused in TEX or ε-TEX modes, we will initialize it since we add code to the *primitive* procedure and we need TEX and ε-TEX ones to be registered as well, whether `INITEX` switches to PRoTE mode later or not.

⟨ Initialize table entries (done by `INITEX` only) 163 ⟩ +≡
 `ROM`[*ROM_undefined_primitive*] = *eqtb*[*undefined_control_sequence*];
 for ($k = ROM\_base$; $k \le 256$; k++) `ROM`[k] = `ROM`[*ROM_undefined_primitive*];
 for ($k = ROM\_undefined\_primitive + 1$; $k \le ROM\_size$; k++) `ROM`[k] = `ROM`[*ROM_undefined_primitive*];

1582. When a primitive is added—and this only happens in `INITEX`—we have to define the corresponding address in the `ROM`.

1583. *cur_val* has the pointer in second part of region 1 or in region 2 of *eqtb*.

⟨ Add primitive definition to the ROM array 1583 ⟩ ≡

set_ROM_p_from_cs(*cur_val*);
ROM[p] = *eqtb*[*cur_val*];

This code is used in section 263.

1584. This array has to be dumped since it is only defined by INITEX. It is always dumped even if it is unused unless in PRoTE mode.

⟨ Dump the ROM array 1584 ⟩ ≡

for (k = *ROM_base*; $k \le$ *ROM_size*; k++) *dump_wd*(ROM[k]);

This code is used in section 1306.

1585. And what has been dumped shall be undumped.

⟨ Undump the ROM array 1585 ⟩ ≡

for (k = *ROM_base*; $k \le$ *ROM_size*; k++) *undump_wd*(ROM[k]);

This code is used in section 1307.

1586. Once all this is done, the processing of `\primitive` is simple: we read the next token that has to be a control sequence. If this control sequence belongs to region 1 or 2 and is defined in ROM, we redefine the token to be the *frozen_primitive* control sequence, redefining its codes from the ROM and setting the text associated for printing purposes. If not, the token is unchanged. Then we put back the token so that it will be processed again, maybe redefined.

⟨ Cases for *expandafter* 1586 ⟩ ≡

```
case primitive_code:
  { get_token();
    set_ROM_p_from_cs(cur_cs);
    if ((p ≠ ROM_undefined_primitive) ∧ (ROM_type(p) ≠ undefined_cs)) {
      eqtb[frozen_primitive] = ROM[p];
      text(frozen_primitive) = text(cur_cs);
      cur_tok = cs_token_flag + frozen_primitive;
    }
    back_input();
  } break;
```

See also section 1590.
This code is used in section 366.

1587. The next primitive changes the expansion of its argument that is like a general text expanded, except that protected macros (an ε-TEX extension) are not extanded.

1588. ⟨ Generate all PRoTE primitives 1553 ⟩ +≡

primitive("expanded", *expand_after*, *expanded_code*);

1589. ⟨ Cases of *expandafter* for *print_cmd_chr* 1445 ⟩ +≡

case *expanded_code*: *print_esc*("expanded");

1590. This intervenes in *expand* and we must substitute a token list to our current token, putting it back for further reprocessing.

⟨ Cases for *expandafter* 1586 ⟩ +≡

```
case expanded_code:
  { scan_general_x_text();
    back_list(link(link(garbage)));
    free_avail(link(garbage));    /* drop reference count */
  }
```

1591. PRoTE strings related primitives.
The primitive `\strcmp` text two parameters that are general text without expansion. The two token lists created are converted to strings and this couple of strings is then compared, character by character. If the first string is lexicographically sorted before the second, the expansion is −1; if the two strings are equal, the expansion is 0; if the first string is lexicographically sorted after the second, the expansion is 1.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
primitive("`strcmp`", *convert*, *strcmp_code*);

1592. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡
case *strcmp_code*: *print_esc*("`strcmp`"); **break**;

1593. It should be noted that the strings comparison is TEX strings comparison: the arguments are subject to the manipulation done when scanning a general text (squeezing non escaped blanks), and the characters are converted according to the *xord* array. Thus it is an **ASCII_code**—in the TEX sense explained at the very beginning of the web file, part 2—comparison and the result is the same, as long as relative characters are mapped to the same value, whatever the system. Nul strings are valid.

⟨ Cases of 'Scan the argument for command *c*' 1557 ⟩ +≡

```
case strcmp_code:
  { scan_general_x_text();
    toks_to_str();
    s = info(garbage);
    flush_list(link(garbage));
    scan_general_x_text();
    toks_to_str();
    t = info(garbage);
    flush_list(link(garbage));
    if ((length(s) ≡ 0) ∧ (length(t) ≡ 0)) cur_val = 0;
    else if (length(s) ≡ 0) cur_val = −1;
    else if (length(t) ≡ 0) cur_val = 1;
    else { m = str_start[s];
      n = str_start[t];
      r = false;
      while ((¬r) ∧ (m < str_start[s + 1]) ∧ (n < str_start[t + 1])) { cur_val = str_pool[m] − str_pool[n];
        if (cur_val ≠ 0) r = true;
        incr(m);
        incr(n);
      }
      if (cur_val ≡ 0) { if (length(s) ≠ length(t))
          if (m ≠ str_start[s + 1]) cur_val = 1;
          else cur_val = −1;
      }
      else cur_val = cur_val/(double) abs(cur_val);
    }
    flush_string;
    flush_string;
  } break;
```

1594. ⟨ Cases of 'Print the result of command *c*' 1558 ⟩ +≡
case *strcmp_code*: *print_int*(*cur_val*); **break**;

1595. PRoTE date and time related primitives.
The following primitives are related to the time elapsed since a defined moment in time. The creation date is fixed at the moment when *fix_date_and_time* has been called and stays fixed afterwards. This moment is also, by default, the reference moment for computing the time elapsed.

1596. The creation date is retrieved by the `\creationdate` primitive. As explained above, the date corresponds to the moment when *fix_date_and_time* was called taking into account `FORCE_SOURCE_DATE` and `SOURCE_DATE_EPOCH` (see above, m.241). If the creation date is forced, the string will be UTC related.

The format of the string is *D*: *YYYYMMDDHHmmSSOHH* `"mm"`, 'O' being the relationship of local time to UT, that is '-' (minus), '+' or 'Z'; HH followed by a single quote being the absolute value of the offset from UT in hours (00–23), mm followed by a single quote being the absolute value of the offset from UT in minutes (00–59). All fields after the year are optional and default to zero values.

1597. ⟨ Generate all PRoTE primitives 1553 ⟩ +≡
primitive(`"creationdate"`, *convert*, *creation_date_code*);

1598. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡
case *creation_date_code*: *print_esc*(`"creationdate"`); **break**;

1599. *get_creation_date* has to be provided by the system.
⟨ Cases of 'Scan the argument for command *c*' 1557 ⟩ +≡
case *creation_date_code*: *get_creation_date*( ); **break**;

1600. The date is in the *time_str* so we have simply to convert the characters.
⟨ Cases of 'Print the result of command *c*' 1558 ⟩ +≡
case *creation_date_code*:
for ($k = 0$; *time_str*[k] $\neq$ `'\0'`; $k{+}{+}$) *print_char*(*time_str*[k]); **break**;

1601. The time elapsed is a scaled integer the unit being scaled seconds, i.e. 1/65536 of a second. Since our scaled integers have a defined range, the value can not reach or pass, in plain seconds, 32767.

The elapsed time returned is relative to some defined moment. At start, the reference moment is the time the date was set for *fix_date_and_time*. This requires system support and the default implementation here will then fix this moment at noon on 4 July 1776 and what would be returned by the function is here simply defined by a macro: with this reference time and this basic code, *infinity* is the permanent answer.

#define *get_elapsed_time* *infinity* /* a function should be implemented */

1602. ⟨ Generate all PRoTE primitives 1553 ⟩ +≡
primitive(`"resettimer"`, *extension*, *reset_timer_code*);
primitive(`"elapsedtime"`, *last_item*, *elapsed_time_code*);

1603. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡
case *elapsed_time_code*: *print_esc*(`"elapsedtime"`); **break**;

1604. ⟨ Cases of *extension* for *print_cmd_chr* 1604 ⟩ ≡
case *reset_timer_code*: *print_esc*(`"resettimer"`); **break**;
See also sections 1672 and 1738.
This code is used in section 1345.

1605. ⟨ Cases for fetching a PRoTE int value 1555 ⟩ +≡
case *elapsed_time_code*: *cur_val* = *get_elapsed_time*; **break**;

1606. The reference moment can be reset by a call to the primitive `\resettimer`. It simply resets the reference moment to the moment the primitive was called. The counter is not regularily incremented. When asked about the time elapsed what is returned is the difference, in scaled seconds, from the moment of the call to the moment of reference. So there is no persistent variable neither a kind of clock implemented.

Standard Pascal doesn't provide related routines so our syntactically correct but semantically useless routines are implemented here: the *reset_timer* does nothing, while the *get_elapsed_time* simply returns, even when *reset_timer* has been called, the invalid value *infinity*.

#**define** *reset_timer* *do_nothing*

1607. Since to reset the timer a simple call to the routine is necessary, we simply add it to `main_control` by adding it to the cases handled by `do_extension`. It contributes nothing to the token list: it is a "fire and forget", so no need to handle the special `subtype` in the other hooks.

⟨ Cases for *do_extension* 1607 ⟩ ≡
case *reset_timer_code*: *reset_timer*; **break**;

See also sections 1673 and 1739.

This code is used in section 1347.

1608. PRoTE **file related primitives.**
The presence of the following primitives in the engine can be questioned. Since they are very external, and their implementation, for example in C, requires things that are not in the C standard (the date of modification of the file, for example). So these should not be multiplied.

1609. The `\filesize` primitive expands to the size, in bytes, of the file.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
  primitive("`filesize`", *convert*, *file_size_code*);

1610. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡
case *file_size_code*: *print_esc*("`filesize`"); **break**;

1611. In order to be able to treat the problem when trying to open the file, we open here and pass the file pointer, if success, to a dedicated function in order to get its size. In case of problem, nothing is returned.

⟨ Cases of 'Scan the argument for command c' 1557 ⟩ +≡
case *file_size_code*:
  { *scan_general_x_text*( );
    toks_to_str( );
    $s = info(garbage)$;
    flush_list(*link*(*garbage*));
    str_to_name(s);
    $cur\_val = -1$; /* invalid value if error */
    $cur\_val = get\_file\_size()$;
    flush_string;
  } **break**;

1612. ⟨ Cases of 'Print the result of command c' 1558 ⟩ +≡
case *file_size_code*:
  if ($cur\_val \neq -1$) *print_int*(*cur_val*); **break**;

1613. The `\filemoddate` expands to a date with the same format as the creation date (see `\creationdate`).

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
  primitive("`filemoddate`", *convert*, *file_mod_date_code*);

1614. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡
case *file_mod_date_code*: *print_esc*("`filemoddate`"); **break**;

1615. For getting the argument, the treatment resembles that of `\filesize` obviously, since it is only the type of information returned that changes. The availability of this information in system dependent. The information shall be set in *xchg_buffer*.

In this basic implementation, we set the string to the empty one by simply setting *xchg_buffer_length* to 0.

#define *get_file_mtime* *xchg_buffer_length* = 0

⟨ Cases of 'Scan the argument for command *c*' 1557 ⟩ +≡
case *file_mod_date_code*:
  { *scan_general_x_text*( );
    toks_to_str( );
    s = *info*(*garbage*);
    flush_list(*link*(*garbage*));
    str_to_name(*s*);
    get_file_mod_date( );
    flush_string;
  } **break**;

1616. Printing the result consists simply in printing every **text_char** in *time_str*. If the length is 0, nothing is printed.

⟨ Cases of 'Print the result of command *c*' 1558 ⟩ +≡
case *file_mod_date_code*:
  for ($k = 0$; *time_str*[*k*] ≠ `'\0'`; *k*++) *print_char*(*time_str*[*k*]); **break**;

1617. The primitive `\filedump` expands to the dump of the first `length` bytes of the file, starting from `offset`. Offset and length are optional integers given, in that order, introduced resp. by the keywords "offset" and "length". If not specified, they default to 0. A length of 0 expands to nothing (it is not an error). The file name is given as a *general text*.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
  primitive(`"filedump"`, *convert*, *file_dump_code*);

1618. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡
case *file_dump_code*: *print_esc*(`"filedump"`); **break**;

1619. The scanning of the arguments is obvious from the syntax above.

Since "offset" and "length" may be given in that order, we assign the variables `k` and `l`, in alphabetical order. These have to be positive or nul values.

Contrary to other blocks, and for optimization purposes (in order not to clobber the string pool with data that we can read, when necessary, one byte at a time), `k`, `l` and `f` will be defined here and used when printing.

⟨ Cases of 'Scan the argument for command *c*' 1557 ⟩ +≡

```
case file_dump_code:
  { k = 0;
    l = 0;    /* defaults */
    if (scan_keyword("offset")) { scan_int();
      if (cur_val < 0) { print_err("Bad␣");
        print_esc("filedump");
        help2("I␣allow␣only␣nonnegative␣values␣here.",
        "I␣changed␣this␣one␣to␣zero.");
        int_error(cur_val);
      }
      else k = cur_val;
    }
    if (scan_keyword("length")) { scan_int();
      if (cur_val < 0) { print_err("Bad␣");
        print_esc("filedump");
        help2("I␣allow␣only␣nonnegative␣values␣here.",
        "I␣changed␣this␣one␣to␣zero.");
        int_error(cur_val);
      }
      else l = cur_val;
    }
    scan_general_x_text();
    toks_to_str();
    s = info(garbage);
    flush_list(link(garbage));
    str_to_name(s);
    flush_string;    /* this one was the filename argument */
  } break;
```

1620. The variables have been set, and the file name has been defined. We simply print the uppercase hexadecimal transcription of every byte requested before closing the file. Here we deal with bytes (`eight_bits` values) so there is no transcription.

⟨ Cases of 'Print the result of command *c*' 1558 ⟩ +≡
case *file_dump_code*:
  { **FILE** ∗*f* = *fopen*((**char** ∗) *name_of_file0*, "rb");
    if (*f* ≠ Λ) { *fseek*(*f*, *k*, SEEK_SET);
      do { *i* = *fgetc*(*f*);
        if (*i* ≡ EOF) **break**;
        dig[0] = *i* % 16;
        dig[1] = *i*/16;
        print_the_digs(2);
        decr(*l*);
      } **while** (¬(*feof*(*f*) ∨ (*l* ≡ 0)));
      fclose(*f*);
    }
  } **break**;

1621. The `\mdfivesum` is obviously a variant of the `convert` class since it takes values from external and put them as a token list in the stream.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
  primitive("mdfivesum", *convert*, *mdfive_sum_code*);

1622. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡
case *mdfive_sum_code*: *print_esc*("mdfivesum"); **break**;

1623. There is an optional keyword "file" that will tell us if the < *general text* > is to be taken as a filename or just as the string to hash. The < *balanced text* > is expanded in both cases.

Once this is done, we ask to init the MD5 state; then fill the exchange buffer with chunks of data and update the MD5 hash with every chunk until source is exhausted and ask for the final (16 bytes numerical value) result that will be put in the *xchg_buffer*.

Since we are looking for a "general text", that must be enclosed (at least: ended; the opening brace can be implicit) by a *right_brace*, an error will be caught with runaways.

The general text is converted to a string. It is legal to have an empty string if the argument is not a file.

⟨ Cases of 'Scan the argument for command *c*' 1557 ⟩ +≡
case *mdfive_sum_code*:
  { *r* = *scan_keyword*("file");
    scan_general_x_text( );
    toks_to_str( );
    s = *info*(*garbage*);
    flush_list(*link*(*garbage*));
    l = *get_md5_sum*(*s*, *r*);
    flush_string;   /∗ done with the filename or string to hash ∗/
  } **break**;

1624. As a result, there is 16 bytes in the *md5_digest* representing the MD5 hash. We simply print, byte by byte, the uppercase hexadecimal representation of this hash.

⟨ Cases of 'Print the result of command *c*' 1558 ⟩ +≡
case *mdfive_sum_code*:
  for (*k* = 0; *k* < *l*; *k*++) { *dig*[0] = *md5_digest*[*k*] % 16; *dig*[1] = *md5_digest*[*k*]/16;
    print_the_digs(2);
  } **break**;

1625. Pseudo-random number generation.
These routines come from John Hobby's METAPOST and generate pseudo-random numbers with the additive scheme recommended in Section 3.6 of *The Art of Computer Programming*; however, the results are random fractions between 0 and *mpfract_one* − 1, inclusive.

METAPOST uses 28 significant bits of precision and we have kept this in order for the routines to behave the same way as in METAPOST. So the name *mpfract* will be used instead of **scaled**, while the two are integers, in the range defined by TEX.

#**define double**(*A*) $A = A + A$ /∗ multiply a variable by two ∗/
#**define** *halfp*(*A*) $(A)/2$ /∗ when quantity is known to be positive or zero ∗/

1626. The subroutines for logarithm and exponential involve two tables. The first is simple: *two_to_the*[*k*] equals 2^k. The second involves a bit more calculation, which the author claims to have done correctly: *spec_log*[*k*] is 2^{27} times $\ln\bigl(1/(1-2^{-k})\bigr) = 2^{-k} + \frac{1}{2}2^{-2k} + \frac{1}{3}2^{-3k} + \cdots$, rounded to the nearest integer.

⟨ Global variables 13 ⟩ +≡
static int *two_to_the*[31]; /∗ powers of two ∗/
static int *spec_log0*[28], ∗**const** *spec_log* = *spec_log0* − 1; /∗ special logarithms ∗/

1627. ⟨ PRoTE initializations 1567 ⟩ +≡
two_to_the[0] = 1;
for ($k = 1$; $k \le 30$; *k*++) *two_to_the*[*k*] = 2 ∗ *two_to_the*[*k* − 1];
spec_log[1] = 93032640;
spec_log[2] = 38612034;
spec_log[3] = 17922280;
spec_log[4] = 8662214;
spec_log[5] = 4261238;
spec_log[6] = 2113709;
spec_log[7] = 1052693;
spec_log[8] = 525315;
spec_log[9] = 262400;
spec_log[10] = 131136;
spec_log[11] = 65552;
spec_log[12] = 32772;
spec_log[13] = 16385;
for ($k = 14$; $k \le 27$; *k*++) *spec_log*[*k*] = *two_to_the*[27 − *k*];
spec_log[28] = 1;

1628. Here is the routine that calculates 2^8 times the natural logarithm of a **scaled** quantity; it is an integer approximation to $2^{24}\ln(x/2^{16})$, when x is a given positive integer.

The method is based on exercise 1.2.2–25 in *The Art of Computer Programming*: During the main iteration we have $1Ł2^{-30}x < 1/(1-2^{1-k})$, and the logarithm of $2^{30}x$ remains to be added to an accumulator register called y. Three auxiliary bits of accuracy are retained in y during the calculation, and sixteen auxiliary bits to extend y are kept in z during the initial argument reduction. (We add $100\cdot 2^{16} = 6553600$ to z and subtract 100 from y so that z will not become negative; also, the actual amount subtracted from y is 96, not 100, because we want to add 4 for rounding before the final division by 8.)

⟨ Declare PRoTE arithmetic routines 1628 ⟩ ≡

```
static scaled m_log(scaled x)
{ int y, z;    /* auxiliary registers */
  int k;    /* iteration counter */
  if (x ≤ 0) ⟨ Handle non-positive logarithm 1630 ⟩
  else { y = 1302456956 + 4 − 100;    /* 14 × 2^27 ln 2 ≈ 1302456956.421063 */
    z = 27595 + 6553600;    /* and 2^16 × .421063 ≈ 27595 */
    while (x < mpfract_four) { double(x);
      y = y − 93032639;
      z = z − 48782;
    }    /* 2^27 ln 2 ≈ 93032639.74436163 and 2^16 × .74436163 ≈ 48782 */
    y = y + (z/unity);
    k = 2;
    while (x > mpfract_four + 4)
      ⟨ Increase k until x can be multiplied by a factor of 2^−k, and adjust y accordingly 1629 ⟩;
    return y/8;
  }
}
```

See also sections 1632, 1634, 1647, 1648, 1649, 1654, and 1656.

This code is used in section 107.

1629. ⟨ Increase k until x can be multiplied by a factor of 2^{-k}, and adjust y accordingly 1629 ⟩ ≡

```
{ z = ((x − 1)/two_to_the[k]) + 1;    /* z = ⌈x/2^k⌉ */
  while (x < mpfract_four + z) { z = halfp(z + 1);
    k = k + 1;
  }
  y = y + spec_log[k];
  x = x − z;
}
```

This code is used in section 1628.

1630. ⟨ Handle non-positive logarithm 1630 ⟩ ≡

```
{ print_err("Logarithm␣of␣");
  print_scaled(x);
  print("␣has␣been␣replaced␣by␣0");
  help2("Since␣I␣don't␣take␣logs␣of␣non-positive␣numbers,",
  "I'm␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed.");
  error();
  return 0;
}
```

This code is used in section 1628.

1631. Here is introduced the special 28bits significand *mpfract*.

```
#define el_gordo °17777777777   /* 2^31 − 1, the largest value that TeX likes */
#define mpfract_half °1000000000   /* 2^27, represents 0.50000000 */
#define mpfract_one °2000000000   /* 2^28, represents 1.00000000 */
#define mpfract_four °10000000000   /* 2^30, represents 4.00000000 */
```

⟨Types in the outer block 18⟩ +≡

```
  typedef int mpfract;   /* this type is used for pseudo-random numbers */
```

1632. The *make_mpfract* routine produces the **mpfract** equivalent of $p/(\textbf{double})\ q$, given integers p and q; it computes the integer $f = \lfloor 2^{28}p/q + \frac{1}{2}\rfloor$, when p and q are positive. If p and q are both of the same scaled type t, the "type relation" $make\_mpfract(t,t) \equiv$ **mpfract** is valid; and it's also possible to use the subroutine "backwards," using the relation $make\_mpfract(t,\textbf{mpfract}) \equiv t$ between scaled types.

If the result would have magnitude 2^{31} or more, *make_mpfract* sets *arith_error* = *true*. Most of TeX's internal computations have been designed to avoid this sort of error.

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

The present implementation is highly portable, but slow; it avoids multiplication and division except in the initial stage. But since it is not part of TeX inner loop, it doesn't matter.

⟨Declare PRoTE arithmetic routines 1628⟩ +≡

```
  static mpfract make_mpfract(int p, int q)
  { int f;   /* the fraction bits, with a leading 1 bit */
    int n;   /* the integer part of |p/q| */
    bool negative;   /* should the result be negated? */
    int be_careful;   /* disables certain compiler optimizations */

    if (p ≥ 0) negative = false;
    else { negate(p);
      negative = true;
    }
    if (q ≤ 0) {
#ifdef DEBUG
      if (q ≡ 0) confusion("/");
#endif
      negate(q);
      negative = ¬negative;
    }
    n = p/q;
    p = p % q;
    if (n ≥ 8) { arith_error = true;
      if (negative) return −el_gordo; else return el_gordo;
    }
    else { n = (n − 1) * mpfract_one;
      ⟨Compute f = ⌊2^28(1 + p/q) + 1/2⌋ 1633⟩;
      if (negative) return −(f + n); else return f + n;
    }
  }
```

1633. The **do** { loop here preserves the following invariant relations between f, p, and q: (i) $0 \le p < q$; (ii) $fq + p = 2^k(q + p_0)$, where k is an integer and p_0 is the original value of p.

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

⟨ Compute $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$ 1633 ⟩ ≡

```
f = 1;
do { be_careful = p − q;
  p = be_careful + p;
  if (p ≥ 0) f = f + f + 1;
  else { double(f);
    p = p + q;
  }
} while (¬(f ≥ mpfract_one));
be_careful = p − q; if (be_careful + p ≥ 0) incr(f)
```

This code is used in section 1632.

1634. The dual of *make_mpfract* is *take_mpfract*, which multiplies a given integer q by a fraction f. When the operands are positive, it computes $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor$, a symmetric function of q and f.

⟨ Declare PRoTE arithmetic routines 1628 ⟩ +≡

```
static int take_mpfract(int q, mpfract f)
{ int p;    /* the fraction so far */
  bool negative;    /* should the result be negated? */
  int n;    /* additional multiple of q */
  int be_careful;    /* disables certain compiler optimizations */
  ⟨ Reduce to the case that f ≥ 0 and q > 0 1635 ⟩;
  if (f < mpfract_one) n = 0;
  else { n = f/mpfract_one;
    f = f % mpfract_one;
    if (q ≤ el_gordo/n) n = n ∗ q;
    else { arith_error = true;
      n = el_gordo;
    }
  }
  f = f + mpfract_one;
  ⟨ Compute p = ⌊qf/2^28 + 1/2⌋ − q 1636 ⟩;
  be_careful = n − el_gordo;
  if (be_careful + p > 0) { arith_error = true;
    n = el_gordo − p;
  }
  if (negative) return −(n + p);
  else return n + p;
}
```

1635. ⟨ Reduce to the case that $f \ge 0$ and $q > 0$ 1635 ⟩ ≡

if $(f \ge 0)$ *negative* = *false*;
else { *negate*(f);
  negative = *true*;
}
if $(q < 0)$ { *negate*(q);
  negative = ¬*negative*;
}

This code is used in section 1634.

1636. The invariant relations in this case are (i) $\lfloor (qf+p)/2^k \rfloor = \lfloor qf_0/2^{28} + \frac{1}{2} \rfloor$, where k is an integer and f_0 is the original value of f; (ii) $2^k Ł f < 2^{k+1}$.

⟨ Compute $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$ 1636 ⟩ ≡

p = *mpfract_half*;   /* that's 2^{27}; the invariants hold now with $k = 28$ */
if (*q* < *mpfract_four*)
  do {
    if (*odd*(*f*)) *p* = *halfp*(*p* + *q*); **else** *p* = *halfp*(*p*);
    f = *halfp*(*f*);
  } **while** (¬(*f* ≡ 1));
else
do {
  if (*odd*(*f*)) *p* = *p* + *halfp*(*q* − *p*); **else** *p* = *halfp*(*p*);
  f = *halfp*(*f*);
} **while** (¬(*f* ≡ 1))

This code is used in section 1634.

1637. There's an auxiliary array *randoms* that contains 55 pseudo-random fractions. Using the recurrence $x_n = (x_{n-55} - x_{n-31}) \bmod 2^{28}$, we generate batches of 55 new x_n's at a time by calling *new_randoms*. The global variable *j_random* tells which element has most recently been consumed.

⟨ Global variables 13 ⟩ +≡

static mpfract *randoms*[55];   /* the last 55 random values generated */
static int *j_random*;   /* the number of unused *randoms* */

1638. This array of pseudo-random numbers is set starting from a seed value, that is kept in the global integer *random_seed*.

⟨ Global variables 13 ⟩ +≡

static int *random_seed*;   /* seed for pseudo-random number generation */

1639. ⟨ Generate all PRoTE primitives 1553 ⟩ +≡

primitive("randomseed", *last_item*, *random_seed_code*);

1640. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡

case *random_seed_code*: *print_esc*("randomseed"); **break**;

1641. ⟨ Cases for fetching a PRoTE int value 1555 ⟩ +≡

case *random_seed_code*: *cur_val* = *random_seed*; **break**;

1642. We set the initial value from the system time. System integrators could provide a better source of pseudo-randomness.

Every time a new seed value is assigned, the array has to be regenerated for consumption by routines explained a little later.

⟨ PRoTE initializations 1567 ⟩ +≡
  random_seed = *sys_time*;
  init_randoms( );

1643. Since changing the value must trigger the redefinition of the array, a dedicated primitive is defined to take the new seed and call *init_randoms*.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
  primitive(`"setrandomseed"`, *convert*, *set_random_seed_code*);

1644. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡
case *set_random_seed_code*: *print_esc*(`"setrandomseed"`); **break**;

1645. Once we have retrieved and redefined *random_seed*, we must regenerate the *randoms* array.

⟨ Cases of 'Scan the argument for command *c*' 1557 ⟩ +≡
case *set_random_seed_code*:
  { *scan_int*( );
    random_seed = *cur_val*;
    init_randoms( );
  } **break**;

1646. ⟨ Cases of 'Print the result of command *c*' 1558 ⟩ +≡
case *set_random_seed_code*: *print_int*(*random_seed*); **break**;

1647. To consume a random fraction, the program below will say '*next_random*' and then it will fetch *randoms*[*j_random*].

#**define** *next_random*
    if (*j_random* ≡ 0) *new_randoms*( );
    else *decr*(*j_random*)

⟨ Declare PRoTE arithmetic routines 1628 ⟩ +≡
  static void *new_randoms*(**void**)
  { **int** *k*; /∗ index into *randoms* ∗/
    int *x*; /∗ accumulator ∗/
    for ($k = 0$; $k \le 23$; *k*++) { x = *randoms*[*k*] − *randoms*[$k + 31$];
      if ($x < 0$) $x = x$ + *mpfract_one*;
      randoms[*k*] = *x*;
    }
    for ($k = 24$; $k \le 54$; *k*++) { x = *randoms*[*k*] − *randoms*[$k - 24$];
      if ($x < 0$) $x = x$ + *mpfract_one*;
      randoms[*k*] = *x*;
    }
    j_random = 54;
  }

1648. To initialize the *randoms* table, we call the following routine.

⟨ Declare PRoTE arithmetic routines 1628 ⟩ +≡

```
static void init_randoms(void)
{ mpfract j, jj, k;    /* more or less random integers */
  int i;    /* index into randoms */
  j = abs(random_seed);
  while (j ≥ mpfract_one) j = halfp(j);
  k = 1;
  for (i = 0; i ≤ 54; i++) { jj = k;
    k = j − k;
    j = jj;
    if (k < 0) k = k + mpfract_one;
    randoms[(i * 21) % 55] = j;
  }
  new_randoms();
  new_randoms();
  new_randoms();    /* "warm up" the array */
}
```

1649. To produce a uniform random number in the range $0 \le u < x$ or $0 \ge u > x$ or $0 \equiv u \equiv x$, given a **scaled** value x, we proceed as shown here.

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

⟨ Declare PRoTE arithmetic routines 1628 ⟩ +≡

```
static scaled unif_rand(scaled x)
{ scaled y;    /* trial value */
  next_random;
  y = take_mpfract(abs(x), randoms[j_random]);
  if (y ≡ abs(x)) return 0;
  else if (x > 0) return y;
  else return −y;
}
```

1650. This can be used by calling the following primitive.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡

```
primitive("uniformdeviate", convert, uniform_deviate_code);
```

1651. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡

```
case uniform_deviate_code: print_esc("uniformdeviate"); break;
```

1652. It takes one integer argument obviously that will be the argument to the function.

⟨ Cases of 'Scan the argument for command *c*' 1557 ⟩ +≡

```
case uniform_deviate_code:
  { scan_int();
    cur_val = unif_rand(cur_val);
  } break;
```

1653. ⟨ Cases of 'Print the result of command *c*' 1558 ⟩ +≡

```
case uniform_deviate_code: print_int(cur_val); break;
```

1654. The following somewhat different subroutine tests rigorously if ab is greater than, equal to, or less than cd, given integers (a, b, c, d). In most cases a quick decision is reached. The result is $+1$, 0, or -1 in the three respective cases.

```
#define return_sign(A)
        { return A;
        }
⟨Declare PRoTE arithmetic routines 1628⟩ +≡
  static int ab_vs_cd(int a, int b, int c, int d)
  { int q, r;    /* temporary registers */
    ⟨Reduce to the case that a, c ≥ 0, b, d > 0 1655⟩;
    loop { q = a/d;
      r = c/b;
      if (q ≠ r)
        if (q > r) return_sign(1) else return_sign(−1);
      q = a % d;
      r = c % b;
      if (r ≡ 0)
        if (q ≡ 0) return_sign(0) else return_sign(1);
      if (q ≡ 0) return_sign(−1);
      a = b;
      b = q;
      c = d;
      d = r;
    }    /* now a > d > 0 and c > b > 0 */
  }
```

1655. ⟨Reduce to the case that $a, c \geq 0$, $b, d > 0$ 1655⟩ ≡

```
  if (a < 0) { negate(a);
    negate(b);
  }
  if (c < 0) { negate(c);
    negate(d);
  }
  if (d ≤ 0) { if (b ≥ 0)
      if (((a ≡ 0) ∨ (b ≡ 0)) ∧ ((c ≡ 0) ∨ (d ≡ 0))) return_sign(0)
      else return_sign(1);
    if (d ≡ 0)
      if (a ≡ 0) return_sign(0) else return_sign(−1);
    q = a;
    a = c;
    c = q;
    q = −b;
    b = −d;
    d = q;
  }
  else if (b ≤ 0) { if (b < 0)
      if (a > 0) return_sign(−1);
    if (c ≡ 0) return_sign(0)
    else return_sign(−1);
  }
```

This code is used in section 1654.

1656. Finally, a normal deviate with mean zero and unit standard deviation can readily be obtained with the ratio method (Algorithm 3.4.1R in *The Art of Computer Programming*).

⟨ Declare PRoTE arithmetic routines 1628 ⟩ +≡

```
static scaled norm_rand(void)
{ int x, u, l;    /* what the book would call 2^16 X, 2^28 U, and −2^24 ln U */
  do {
    do { next_random;
      x = take_mpfract(112429, randoms[j_random] − mpfract_half);    /* 2^16 √(8/e) ≈ 112428.82793 */
      next_random;
      u = randoms[j_random];
    } while (¬(abs(x) < u));
    x = make_mpfract(x, u);
    l = 139548960 − m_log(u);    /* 2^24 · 12 ln 2 ≈ 139548959.6165 */
  } while (¬(ab_vs_cd(1024, l, x, x) ≥ 0));
  return x;
}
```

1657. This can be used by calling the following primitive.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡

primitive("normaldeviate", *convert*, *normal_deviate_code*);

1658. ⟨ Cases of *convert* for *print_cmd_chr* 1556 ⟩ +≡

case *normal_deviate_code*: *print_esc*("normaldeviate");

1659. ⟨ Cases of 'Scan the argument for command c' 1557 ⟩ +≡

case *normal_deviate_code*: *cur_val* = *norm_rand*( );

1660. ⟨ Cases of 'Print the result of command c' 1558 ⟩ +≡

case *normal_deviate_code*: *print_int*(*cur_val*);

1661. DVI related primitives.

These primitives are related to positions in the DVI output.

The TEX and DVI system coordinates relate to an origin that is at the upper left corner. The TEX coordinates are computed relative to an origin that has $(0,0)$ coordinates. Coordinates grow then rightward and downward. This is the *page* coordinates relative to what is typeset (what TEX is dealing with).

But this typesetting material has to be put on what we will call *paper*. The material put into shape by TEX is put on the paper. On this paper, where will be put the TEX origin? It is considered to be $1in$ at the right and $1in$ down from the upper left corner of the paper (see m.590, alinea 2).

#define *DVI_std_x_offset* 4736286 /∗ 1 inch in sp ∗/
#define *DVI_std_y_offset* 4736286 /∗ 1 inch in sp ∗/

1662. But the paper size is not specified in the DVI file and is not being dealt with by TEX.

In order to have a common reference point, and since the `\lastxpos` and `\lastypos` primitives originated in pdfTEX, these two primitives give positions, in scaled points, relative to the lower left corner of the paper.

Hence the need, for these primitive, to define the paper size, with the (misnamed) `\pagewidth` and `\pageheight`.

`\pagewidth` and `\pageheight` are dimension parameters, initialized to 0 by the generic TEX code.

⟨ Generate all PRoTE primitives 1553 ⟩ +≡
primitive(`"pagewidth"`, *assign_dimen*, *dimen_base* + *page_width_code*);
primitive(`"pageheight"`, *assign_dimen*, *dimen_base* + *page_height_code*);

1663. When instructed to, the `h` and `v` last values are transformed, in the coordinates system defined above and saved in the global variables *last_saved_xpos* and *last_saved_ypos*. They are initialized to 0 and we do not make any verification that a call to the `\savepos` primitive—to come—has been made before retrieving their values.

⟨ Global variables 13 ⟩ +≡
static scaled *last_saved_xpos*, *last_saved_ypos*; /∗ last (x,y) DVI pos saved ∗/

1664. ⟨ PRoTE initializations 1567 ⟩ +≡
last_saved_xpos = 0;
last_saved_ypos = 0;

1665. ⟨ Set *last_saved_xpos* and *last_saved_ypos* with transformed coordinates 1665 ⟩ ≡
last_saved_xpos = *cur_h* + *DVI_std_x_offset*;
last_saved_ypos = *page_height* − (*cur_v* + *DVI_std_y_offset*);

This code is used in section 1678.

1666. ⟨ Generate all PRoTE primitives 1553 ⟩ +≡
primitive(`"lastxpos"`, *last_item*, *last_xpos_code*);
primitive(`"lastypos"`, *last_item*, *last_ypos_code*);

1667. ⟨ Cases of *last_item* for *print_cmd_chr* 1380 ⟩ +≡
case *last_xpos_code*: *print_esc*(`"lastxpos"`); **break**;
case *last_ypos_code*: *print_esc*(`"lastypos"`); **break**;

1668. ⟨ Cases for fetching a PRoTE int value 1555 ⟩ +≡
case *last_xpos_code*: *cur_val* = *last_saved_xpos*; **break**;
case *last_ypos_code*: *cur_val* = *last_saved_ypos*;

1669. *last_saved_xpos* and *last_saved_ypos* are only defined when instructed to by the call the the `\savepos` primitive. Since the real work has to be done at `shipout` time, it is a case to be treated like the `\special` primitive, that is it belongs to the `extension` class.

We will add something more in the handling of the primitive: it will insert a `whatsit` in the DVI file so that one, using the program *dvitype*, could retrieve more than one *hic*. So there is a counter incremented whenever the primitive is called.

⟨Global variables 13⟩ +≡
static int *last_save_pos_number*; /∗ identifying the order of the call ∗/

1670. ⟨PRoTE initializations 1567⟩ +≡
last_save_pos_number = 0; /∗ i.e. none ∗/

1671. ⟨Generate all PRoTE primitives 1553⟩ +≡
primitive("`savepos`", *extension*, *save_pos_code*);

1672. ⟨Cases of *extension* for *print_cmd_chr* 1604⟩ +≡
case *save_pos_code*: *print_esc*("`savepos`"); **break**;

1673. ⟨Cases for *do_extension* 1607⟩ +≡
case *save_pos_code*: ⟨Implement `\savepos` 1674⟩ **break**;

1674. We need the basic two words node, since we don't pass any parameter and it is just an instruction to do something. So the `whatsit` node is just the call.

⟨Implement `\savepos` 1674⟩ ≡
{ *new_whatsit*(*save_pos_code*, *small_node_size*);
write_stream(*tail*) = *null*;
write_tokens(*tail*) = *null*;
}

This code is used in section 1673.

1675. ⟨Cases for displaying the *whatsit* node 1675⟩ ≡
case *save_pos_code*: *print_esc*("`savepos`"); **break**;

This code is used in section 1355.

1676. ⟨Cases for making a partial copy of the whatsit node 1676⟩ ≡
case *save_pos_code*:
{ *r* = *get_node*(*small_node_size*);
words = *small_node_size*;
} **break**;

This code is used in section 1356.

1677. ⟨Cases for wiping out the whatsit node 1677⟩ ≡
case *save_pos_code*: *free_node*(*p*, *small_node_size*); **break**;

This code is used in section 1357.

1678. So, after these trivial initializations, what will we effectively do? When the following procedure will be called, we define *last_saved_xpos*, *last_saved_ypos*, increment *last_save_pos_number*, and a *warning* followed by three *key* ≡ *value* space separated definitions as a `\special`, the first being prefixed by the string `__PROTE_` (shall be considered a reserved prefix) and the string `SAVEPOS_`, equal to the index of the call, and the `XPOS` and `YPOS` definitions.

This is obviously, from the previous description, a variation around *special_out*.

⟨ Declare procedures needed in *out_what* 1678 ⟩ ≡

```
static void save_pos_out(pointer p)
{ int old_setting;   /* holds print selector */
  int k;   /* index into str_pool */

  synch_h;
  synch_v;
  incr(last_save_pos_number);
  ⟨ Set last_saved_xpos and last_saved_ypos with transformed coordinates 1665 ⟩
  old_setting = selector;
  selector = new_string;
  print("warning␣__PROTE_");
  print("SAVEPOS");
  print_char('=');
  print_int(last_save_pos_number);
  print_char('␣');
  print("XPOS");
  print("=");
  print_int(last_saved_xpos);
  print_char('␣');
  print("YPOS");
  print("=");
  print_int(last_saved_ypos);
  selector = old_setting;
  str_room(1);   /* abort if probably overflowed and truncated */
  dvi_out(xxx1);
  dvi_out(cur_length);   /* it's less than 256 */
  for (k = str_start[str_ptr]; k ≤ pool_ptr − 1; k++) dvi_out(so(str_pool[k]));
  pool_ptr = str_start[str_ptr];   /* forget the not commited tentative string */
}
```

This code is used in section 1372.

1679. ⟨ Cases for *out_what* 1679 ⟩ ≡

```
case save_pos_code: save_pos_out(p); break;
```

This code is used in section 1372.

1680. 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.

1681. TEX Live Integration. A TEX engine that aspires to become a member of the TEX Live family of programs must
– respect the TEX Live conventions for command line parameters,
– find its input files using the `kpathsearch` library, and
– implement TEX primitives to support LATEX.
Naturally, the functions that follow here are taken, with small modifications, from the TEX Live sources. What is added here, or rather subtracted here, are the parts that are specific to some of the TEX engines included in TEX Live. New is also that the code is presented in literate programming style.

The code that follows is organized in three parts. Some code for TEX Live must come before the definition of TEX's macros because it uses include files containing identifiers that are in conflict with TEX's macros or modify these macros. For example TEX's *banner* is modified by adding the TEX Live version.

⟨ Header files and function declarations 9 ⟩ +≡
```
#ifdef WEB2CVERSION
#define TL_VERSION "(TeX␣Live␣" WEB2CVERSION ")"
#else
#define TL_VERSION
#endif
```

1682. The remaining two parts are first auxiliary variables and functions and then those functions that are called from the "classic" TEX code.

⟨ TEX Live auxiliary variables 1690 ⟩
⟨ TEX Live auxiliary functions 1686 ⟩
⟨ TEX Live functions 1684 ⟩

1683. Most of the code that we present next comes together in the function *main_init* which is the first function called in the main program of a TEX engine belonging to TEX Live. Before doing so, we make copies of argument count and argument vector putting them in global variables.

⟨ Global variables 13 ⟩ +≡
```
static char **argv;
static int argc;
```

1684. ⟨ TEX Live functions 1684 ⟩ ≡
```
static void main_init(int ac, char *av[])
{ char *main_input_file;
  argc = ac;
  argv = av;
  interaction = error_stop_mode;
  kpse_record_input = recorder_record_input;
  kpse_record_output = recorder_record_output;
  ⟨ parse options 1693 ⟩
  ⟨ set the program and engine name 1715 ⟩
  ⟨ activate configuration lines 1714 ⟩
  ⟨ set the input file name 1717 ⟩
  ⟨ set defaults from the texmf.cfg file 1718 ⟩
  ⟨ set the format name 1722 ⟩
  ⟨ enable the generation of input files 1730 ⟩
}
```
See also sections 1691, 1726, and 1729.
This code is used in section 1682.

1685. ⟨ Forward declarations 52 ⟩ +≡
```
static void main_init(int ac, char *av[]);
```

1686. Command Line. Let's begin with the beginning: the command line. To see how a command line is structured, we first look at the help text that is displayed if the user asks for it (or if TEX decides that the user needs it). The help text is produced by the function *usage_help*.

⟨TEX Live auxiliary functions 1686⟩ ≡

```
static void usage_help(void)
{ ⟨explain the command line 1687⟩
  fprintf(stdout, "Options:\n" ⟨explain the options 1688⟩);
  fprintf(stdout,
      "\nFor further information and reporting bugs see https://hint.userweb.mwn.de/\n");
  exit(0);
}
```

See also sections 1700, 1704, 1707, 1708, 1713, 1716, 1719, 1723, 1724, 1727, 1728, and 1733.

This code is used in section 1682.

1687. The command line comes in three slightly different versions:

⟨explain the command line 1687⟩ ≡

```
fprintf(stdout,
"Usage: %s [OPTION]... [TEXNAME[.tex]] [COMMANDS]\n"
"   or: %s [OPTION]... \\FIRST-LINE\n"
"   or: %s [OPTION]... &FMT ARGS\n\n",
argv[0], argv[0], argv[0]);
fprintf(stdout,
"  Run TeX on TEXNAME, creating TEXNAME.dvi.\n"
"  Any remaining COMMANDS are processed\n"
"  as TeX input after TEXNAME is read.\n"
"  If the first line of TEXNAME starts with %%&FMT, and FMT is\n"
"  an existing .fmt file, use it. Else use 'NAME.fmt', where\n"
"  NAME is the program invocation name.\n"
"\n"
"  Alternatively, if the first non-option argument begins\n"
"  with a backslash, interpret all non-option arguments as\n"
"  a line of TeX input.\n"
"\n"
"  Alternatively, if the first non-option argument begins\n"
"  with a &, the next word is taken as the FMT to read,\n"
"  overriding all else.  Any remaining arguments are\n"
"  processed as above.\n"
"\n"
"  If no arguments or options are specified, prompt for input.\n"
"\n");
```

This code is used in section 1686.

1688. Options. Here is the list of possible options and their explanation:

⟨ explain the options 1688 ⟩ ≡

```
" -help                  "
    "\t display this help and exit\n"
" -version               "
    "\t output version information and exit\n"
" -etex                  "
    "\t enable e-TeX extensions\n"
" -ltx                  "
    "\t enable LaTeX extensions, implies -etex\n"
" -ini                   "
    "\t be initex for dumping formats; this is\n"
    "\t\t\t also true if the program name is 'kinitex'\n"
" -progname=STRING       "
    "\t set program (and fmt) name to STRING\n"
" -fmt=FMTNAME           "
    "\t use FMTNAME instead of program name or a %%& line\n"
" -output-directory=DIR  "
    "\t use existing DIR as the directory to write files to\n"
" -jobname=STRING        "
    "\t set the TeX \\jobname to STRING\n"
" [-no]-mktex=FMT        "
    "\t disable/enable mktexFMT generation (FMT=tex/tfm/fmt)\n"
" -interaction=STRING    "
    "\t set interaction mode (STRING=batchmode/\n"
    "\t\t\t nonstopmode/scrollmode/errorstopmode)\n"
" -kpathsea-debug=NUMBER"
    "\t set path searching debugging flags according\n"
    "\t\t\t to the bits of NUMBER\n"
" -recorder"
    "\t\t enable filename recorder\n"
" [-no]-parse-first-line"
    "\t disable/enable parsing of the first line of\n"
    "\t\t\t the input file\n"
" [-no]-file-line-error"
    "\t disable/enable file:line:error style\n"
" -cnf-line=STRING"
    "\t process STRING like a line in texmf.cnf\n"
```

See also section 1734.

This code is used in section 1686.

1689. The processing of command line options is controlled by the *long_options* array. Each entry in this array contains first the name of the option, then a flag that tells whether the option takes an argument or not. If next the (optional) address of a flag variable is given, it is followed by the value to store in the flag variable. In this case, setting the flag variable is handled by the *getopt_long_only* function.

Besides the flag variables that occur in the table, a few string variables may be set using the options. The following is a complete list of these variables. Variables are initialized with −1 to indicate an undefined value; string variables are initialized with Λ.

⟨ Global variables 13 ⟩ +≡
static int *iniversion* = *false*, *etexp* = *false*, *ltxp* = *false*, *recorder_enabled* = *false*;
static int *parsefirstlinep* = −1, *filelineerrorstylep* = −1, *interaction_option* = −1;
static const char ∗*user_progname* = Λ, ∗*output_directory* = Λ, ∗*c_job_name* = Λ, ∗*dump_name* = Λ;

1690. ⟨ TEX Live auxiliary variables 1690 ⟩ ≡
static struct *option long_options*[ ] =
{
{"help", 0, 0, 0},
{"version", 0, 0, 0},
{"interaction", 1, 0, 0},
{"mktex", 1, 0, 0},
{"no-mktex", 1, 0, 0},
{"kpathsea-debug", 1, 0, 0},
{"progname", 1, 0, 0},
{"fmt", 1, 0, 0},
{"output-directory", 1, 0, 0},
{"jobname", 1, 0, 0},
{"cnf-line", 1, 0, 0},
{"ini", 0, &*iniversion*, 1},
{"etex", 0, &*etexp*, 1},
{"ltx", 0, &*ltxp*, 1},
{"recorder", 0, &*recorder_enabled*, 1},
{"parse-first-line", 0, &*parsefirstlinep*, 1},
{"no-parse-first-line", 0, &*parsefirstlinep*, 0},
{"file-line-error", 0, &*filelineerrorstylep*, 1},
{"no-file-line-error", 0, &*filelineerrorstylep*, 0},
⟨ more options 1735 ⟩
{0, 0, 0, 0} } ;

This code is used in section 1682.

1691. Parsing the command line options is accomplished with the *parse_options* function which in turn uses the *getopt_long_only* function from the C library. This function returns 0 and sets the *option_index* parameter to the option found, or it returns −1 if the end of all options is reached.

⟨ TEX Live functions 1684 ⟩ +≡
```
static void parse_options(int argc, char *argv[])
{ while (true) { int option_index;
    int g = getopt_long_only(argc, argv, "+", long_options, &option_index);
    if (g ≡ 0) { ⟨ handle the options 1695 ⟩ }
    else if (g ≡ '?') { fprintf(stderr, "Try␣'%s␣--help'␣for␣more␣information\n", argv[0]);
      exit(1);
    }
    else if (g ≡ −1) break;
  }
  ⟨ Check the environment for extra settings 1701 ⟩
}
```

1692. ⟨ Forward declarations 52 ⟩ +≡
```
static void parse_options(int argc, char *argv[]);
```

1693. Before we can call the *parse_options* function, we might need some special preparations for Windows.

⟨ parse options 1693 ⟩ ≡
```
#if defined (WIN32)
  { char *enc;
    kpse_set_program_name(argv[0], Λ);
    enc = kpse_var_value("command_line_encoding");
    get_command_line_args_utf8(enc, &argc, &argv);
    parse_options(argc, argv);
    ⟨ record texmf.cnf 1710 ⟩
  }
#else
  parse_options(ac, av);
#endif
```
This code is used in section 1684.

1694. To handle the options, we compare the name at the given *option_index* with the different option names. This is not a very efficient method, but the impact is low and it's simple to write.

Comparing the name of the argument with the *name* field in the *option* structure is done in the auxiliary function *argument_is*. Unfortunately the *name* field is in conflict with the *name* macro defined by TEX. To avoid the conflict, the *argument_is* function goes just after the `kpathsea.h` header file that defines the option structure.

⟨ Header files and function declarations 9 ⟩ +≡
```
#include <kpathsea/kpathsea.h>
  static int argument_is(struct option *opt, char *s)
  { return STREQ(opt→name, s); }
#define ARGUMENT_IS (S) argument_is (long_options + option_index, S)
```

1695. Now we can handle the first two options:

⟨ handle the options 1695 ⟩ ≡
```
if (ARGUMENT_IS("help")) usage_help();
else if (ARGUMENT_IS("version")) { printf(TeX_banner "\n"
    "Prote␣version␣" Prote_version_string "\n");
    exit(0); }
```

See also sections 1696, 1697, 1698, 1699, and 1712.
This code is used in section 1691.

1696. The "interaction" option sets the *interaction_option* variable based on its string argument contained in the *optarg* variable. If defined, the *interaction_option* will be used to set TEX's *interaction* variable in the *initialize* and the *undump* functions.

⟨ handle the options 1695 ⟩ +≡
```
else if (ARGUMENT_IS("interaction")) {
    if (STREQ(optarg, "batchmode")) interaction_option = batch_mode;
    else if (STREQ(optarg, "nonstopmode")) interaction_option = nonstop_mode;
    else if (STREQ(optarg, "scrollmode")) interaction_option = scroll_mode;
    else if (STREQ(optarg, "errorstopmode")) interaction_option = error_stop_mode;
    else WARNING1("Ignoring␣unknown␣argument␣'%s'␣to␣--interaction", optarg);
}
```

1697. The next two options pass the string argument to the `kpathsearch` library.

⟨ handle the options 1695 ⟩ +≡
```
else if (ARGUMENT_IS("mktex")) kpse_maketex_option(optarg, true);
else if (ARGUMENT_IS("no-mktex")) kpse_maketex_option(optarg, false);
```

1698. To debug the searching done by the `kpathsearch` library, the following option can be used. The argument value 3 is a good choice to start with.

⟨ handle the options 1695 ⟩ +≡
```
else if (ARGUMENT_IS("kpathsea-debug")) kpathsea_debug |= atoi(optarg);
```

1699. The next set of options take a string argument and assign it to the corresponding string variable.

⟨ handle the options 1695 ⟩ +≡
```
else if (ARGUMENT_IS("progname")) user_progname = normalize_quotes(optarg, "program␣name");
else if (ARGUMENT_IS("fmt")) dump_name = normalize_quotes(optarg, "format␣name");
else if (ARGUMENT_IS("output-directory"))
    output_directory = normalize_quotes(optarg, "output␣directory");
else if (ARGUMENT_IS("jobname")) c_job_name = normalize_quotes(optarg, "job␣name");
```

1700. When string arguments specify files or directories, special care is needed if arguments are quoted and/or contain spaces. The function *normalize_quotes* makes sure that arguments containing spaces get quotes around them and it checks for unbalanced quotes.

⟨ TEX Live auxiliary functions 1686 ⟩ +≡

```
static char *normalize_quotes(const char *nom, const char *mesg)
{ int quoted = false;
  int must_quote = (strchr(nom, '␣') ≠ Λ);
  char *ret = xmalloc(strlen(nom) + 3);    /* room for two quotes and NUL */
  char *p = ret;
  const char *q;
  if (must_quote) *p++ = '"';
  for (q = nom; *q; q++)
    if (*q ≡ '"') quoted = ¬quoted; else *p++ = *q;
  if (must_quote) *p++ = '"';
  *p = '\0';
  if (quoted) { fprintf(stderr, "!␣Unbalanced␣quotes␣in␣%s␣%s\n", mesg, nom);
    exit(1);
  }
  return ret;
}
```

1701. If the output directory was specified on the command line, we save it in an environment variable so that subbrocesses can get the value. If on the other hand the environment specifies a directory and the command line does not, save the value from the environment to the global variable so that it is used in the rest of the code.

⟨ Check the environment for extra settings 1701 ⟩ ≡

```
if (output_directory) xputenv("TEXMF_OUTPUT_DIRECTORY", output_directory);
else if (getenv("TEXMF_OUTPUT_DIRECTORY")) output_directory = getenv("TEXMF_OUTPUT_DIRECTORY");
```

This code is used in section 1691.

1702. Passing a file name as a general text argument.
scan_file_name uses the following code to parse a file name given as a general text argument. Such an argument can be any token list starting with a left brace and ending with a right brace. This token list is then expanded (without the leading and trailing braces) and printed into the string pool without making it yet an official string. After removing all double quotes, because this is current practice for TEX engines that are part of TEX Live, and setting the area and extension delimiters, all temporary garbage used so far is freed.

Due to the expansion of the token list, this code and hence the *scan_file_name* procedure is recursive. One can provide the name of a file as the content of an other file.

⟨ Define a general text file name and **goto** *done* 1702 ⟩ ≡

```
{ back_input();
  name_in_progress = false;   /* this version is recursive... */
  cur_cs = input_loc;   /* scan_toks will set warning_index from it */
  scan_general_x_text();
  old_setting = selector;
  selector = new_string;
  token_show(link(garbage));
  selector = old_setting;
  ⟨ Suppress double quotes in braced input file name 1703 ⟩
  j = pool_ptr − 1;
  while ((j ≥ str_start[str_ptr]) ∧ (area_delimiter ≡ 0)) { if ((str_pool[j] ≡ '/'))
      area_delimiter = j − str_start[str_ptr];
    if ((ext_delimiter ≡ 0) ∧ (str_pool[j] ≡ '.')) ext_delimiter = j − str_start[str_ptr];
    decr(j);
  }
  flush_list(link(garbage));
  goto done;
}
```

This code is used in section 525.

1703. A simple loop removes the double quotes and adjusts the *pool_ptr*.

⟨ Suppress double quotes in braced input file name 1703 ⟩ ≡

```
for (k = j = str_start[str_ptr]; k < pool_ptr; k++) { if (str_pool[k] ≠ '"') { str_pool[j] = str_pool[k];
    incr(j);
  }
}
pool_ptr = j;
```

This code is used in section 1702.

1704. The `-recorder` Option. The recorder option can be used to enable the file name recorder. It is crucial for getting a reliable list of files used in a given run. Many post-processors use it, and it is used in TEX Live for checking the format building infrastructure.

When we start the file name recorder, we would like to use mkstemp, but it is not portable, and doing the autoconfiscation (and providing fallbacks) is more than we want to cope with. So we have to be content with using a default name. We throw in the pid so at least parallel builds might work. Windows, however, seems to have no *pid_t*, so instead of storing the value returned by *getpid*, we immediately consume it.

⟨ TEX Live auxiliary functions 1686 ⟩ +≡

```
static char *recorder_name = Λ;
static FILE *recorder_file = Λ;

static void recorder_start(void)
{ char *cwd;
  char pid_str[MAX_INT_LENGTH];
  sprintf(pid_str, "%ld", (long) getpid());
  recorder_name = concat3(kpse_program_name, pid_str, ".fls");
  if (output_directory) { char *temp = concat3(output_directory, DIR_SEP_STRING, recorder_name);
    free(recorder_name);
    recorder_name = temp;
  }
  recorder_file = xfopen(recorder_name, FOPEN_W_MODE);
  cwd = xgetcwd();
  fprintf(recorder_file, "PWD␣%s\n", cwd);
  free(cwd);
}
```

1705. After we know the log file name, we have used *recorder_change_filename* to change the name of the recorder file to the usual thing.

⟨ Forward declarations 52 ⟩ +≡

```
static void recorder_change_filename(const char *new_name);
```

1706. Now its time to define this function. Unfortunately, we have to explicitly take the output directory into account, since the new name we are called with does not; it is just the log file name with `.log` replaced by `.fls`.

1707. ⟨ TEX Live auxiliary functions 1686 ⟩ +≡

```
static void recorder_change_filename(const char *new_name)
{ char *temp = Λ;
    if (¬recorder_file) return;
#if defined (_WIN32)
    fclose(recorder_file);    /* An open file cannot be renamed. */
#endif    /* _WIN32 */
    if (output_directory) { temp = concat3(output_directory, DIR_SEP_STRING, new_name);
      new_name = temp;
    }
#if defined (_WIN32)
    remove(new_name);    /* A file with the new_name must not exist. */
#endif    /* _WIN32 */
    rename(recorder_name, new_name);
    free(recorder_name);
    recorder_name = xstrdup(new_name);
#if defined (_WIN32)
    recorder_file = xfopen(recorder_name, FOPEN_A_MODE);    /* A closed file must be opened. */
#endif    /* _WIN32 */
    if (temp) free(temp);
}
```

1708. Now we are ready to record file names. The prefix INPUT is added to an input file and the prefix OUTPUT to an output file. But both functions for recording a file name use the same function otherwise, which on first use will start the recorder.

⟨ TEX Live auxiliary functions 1686 ⟩ +≡

```
static void recorder_record_name(const char *pfx, const char *fname)
{ if (recorder_enabled) {
      if (¬recorder_file) recorder_start();
      fprintf(recorder_file, "%s␣%s\n", pfx, fname);
      fflush(recorder_file);
  }
}
static void recorder_record_input(const char *fname)
{ recorder_record_name("INPUT", fname);
}
static void recorder_record_output(const char *fname)
{ recorder_record_name("OUTPUT", fname);
}
```

1709. Because input files are also recorded when writing the optional sections, we need the following declaration.

⟨ Forward declarations 52 ⟩ +≡

```
static void recorder_record_input(const char *fname);
```

1710. In WIN32, texmf.cnf is not recorded in the case of `-recorder`, because *parse_options* is executed after the start of kpathsea due to special initializations. Therefore we record `texmf.cnf` with the following code:

⟨record `texmf.cnf` 1710⟩ ≡

```
if (recorder_enabled) { char **p = kpse_find_file_generic("texmf.cnf", kpse_cnf_format, 0, 1);
  if (p ∧ *p) { char **pp = p;
    while (*p) { recorder_record_input(*p);
      free(*p);
      p++;
    }
    free(pp);
  }
}
```

This code is used in section 1693.

1711. The `-cnf-line` Option. With the `-cnf-line` option it is possible to specify a line of text as if this line were part of TEX's configuration file—even taking precedence over conflicting lines in the configuration file. For example it is possible to change TEX's `TEXINPUTS` variable by saying `--cnf-line=TEXINPUTS=/foo`. The configuration lines are temporarily stored in the variable *cnf_lines* and counted in *cnf_count* because we can send them to the `kpathsearch` library only after the library has been initialized sufficiently.

⟨ Global variables 13 ⟩ +≡
```
static char **cnf_lines = Λ;
static int cnf_count = 0;
```

1712. ⟨ handle the options 1695 ⟩ +≡
```
else if (ARGUMENT_IS("cnf-line")) add_cnf_line(optarg);
```

1713. The function *add_cnf_line* stores the given command line argument in the variable *cnf_lines*.

⟨ TEX Live auxiliary functions 1686 ⟩ +≡
```
static void add_cnf_line(char *arg)
{ cnf_count++;
  cnf_lines = xrealloc(cnf_lines, sizeof(char *) * cnf_count);
  cnf_lines[cnf_count - 1] = arg;
}
```

1714. To activate the configuration lines they are passed to the `kpathsearch` library.

⟨ activate configuration lines 1714 ⟩ ≡
```
#if 1   /* this function does not exists always */
{ int i;
  for (i = 0; i < cnf_count; i++) kpathsea_cnf_line_env_progname(kpse_def, cnf_lines[i]);
  free(cnf_lines);
}
#endif
```
This code is used in section 1684.

1715. The Input File. After we are done with the options, we inform the `kpathsearch` library about the program name. This is an important piece of information for the library because the library serves quite different programs and its behavior can be customized for each program using configuration files. After the program and engine name is set, the library is ready to use.

⟨ set the program and engine name 1715 ⟩ ≡
  if (¬*user_progname*) *user_progname* = *dump_name*;
#**if defined** (`WIN32`)
  if (*user_progname*) *kpse_reset_program_name*(*user_progname*);
#**else**
  kpse_set_program_name(*argv*[0], *user_progname*);
#**endif**
  xputenv(`"engine"`, `"hitex"`);

This code is used in section 1684.

1716. After the options, the command line usually continues with the name of the input file. Getting a hold of the input file name can be quite complicated, but the `kpathsearch` library will help us to do the job.

We start by looking at the first argument after the options: If it does not start with an "`&`" and neither with a "\", it's a simple file name. Under Windows, however, filenames might start with a drive letter followed by a colon and a "\" which is used to separate directory names. Finally, if the filename is a quoted string, we need to remove the quotes before we use the `kpathsearch` library to find it and reattach the quotes afterward.

⟨TEX Live auxiliary functions 1686⟩ +≡

```
#ifdef WIN32
  static void clean_windows_filename(char *filename)
  { if (strlen(filename) > 2 ∧ isalpha(filename[0]) ∧ filename[1] ≡ ':' ∧ filename[2] ≡ '\\') { char *pp;
        for (pp = filename; *pp; pp++)
          if (*pp ≡ '\\') *pp = '/';
    }
  }
#endif
  static char *find_file(char *fname, kpse_file_format_type t, int mx)
  { char *filename;
    int final_quote = (int) strlen(fname) − 1;
    int quoted = final_quote > 1 ∧ fname[0] ≡ '"' ∧ fname[final_quote] ≡ '"';
    if (quoted) {     /* Overwrite last quote and skip first quote. */
      fname[final_quote] = '\0';
      fname++;
    }
    filename = kpse_find_file(fname, t, mx);
    if (full_name_of_file ≠ Λ) { free(full_name_of_file);
      full_name_of_file = Λ;
    }
    if (filename ≠ Λ) full_name_of_file = strdup(filename);
    if (quoted) {     /* Undo modifications */
      fname−−;
      fname[final_quote] = '"';
    }
    return filename;
  }
  static char *get_input_file_name(void)
  { char *input_file_name = Λ;
    if (argv[optind] ∧ argv[optind][0] ≠ '&' ∧ argv[optind][0] ≠ '\\') {
#ifdef WIN32
      clean_windows_filename(argv[optind]);
#endif
      argv[optind] = normalize_quotes(argv[optind], "input␣file");
      input_file_name = find_file(argv[optind], kpse_tex_format, false);
    }
    return input_file_name;
  }
```

1717. After we called *get_input_file_name*, we might need to look at *argv*[*argc* − 1] in case we run under Windows.

⟨ set the input file name 1717 ⟩ ≡

```
  main_input_file = get_input_file_name();
#ifdef WIN32     /* Were we given a simple filename? */
  if (main_input_file ≡ Λ) { char *file_name = argv[argc − 1];
    if (file_name ∧ file_name[0] ≠ '-' ∧ file_name[0] ≠ '&' ∧ file_name[0] ≠ '\\') {
      clean_windows_filename(file_name);
      file_name = normalize_quotes(file_name, "argument");
      main_input_file = find_file(file_name, kpse_tex_format, false);
      argv[argc − 1] = file_name;
    }
  }
#endif
```

This code is used in section 1684.

1718. After we have an input file, we make an attempt at filling in options from the `texmf.cfg` file.

⟨ set defaults from the `texmf.cfg` file 1718 ⟩ ≡

```
  if (filelineerrorstylep < 0) filelineerrorstylep = texmf_yesno("file_line_error_style");
  if (parsefirstlinep < 0) parsefirstlinep = texmf_yesno("parse_first_line");
```

This code is used in section 1684.

1719. We needed:

⟨ TEX Live auxiliary functions 1686 ⟩ +≡

```
  static int texmf_yesno(const char *var)
  { char *value = kpse_var_value(var);
    return value ∧ (*value ≡ 't' ∨ *value ≡ 'y' ∨ *value ≡ '1');
  }
```

1720. We need a stack, matching the *line_stack* that contains the source file names. For the full source filenames we use poiters to **char** because these names are just used for output.

⟨ Global variables 13 ⟩ +≡

```
  static char *source_filename_stack0[max_in_open] = {Λ}, **const source_filename_stack =
      source_filename_stack0 − 1;
  static char *full_source_filename_stack0[max_in_open] = {Λ},
      **const full_source_filename_stack = full_source_filename_stack0 − 1;
  static char *full_name_of_file = Λ;
```

1721. The function *print_file_line* prints "file:line:error" style messages using the *source_filename_stack*. If it fails to find the file name, it falls back to the "non-file:line:error" style.

⟨ Basic printing procedures 55 ⟩ +≡

```
static void print_file_line(void)
{ int level = in_open;
  while (level > 0 ∧ full_source_filename_stack[level] ≡ Λ) level−−;
  if (level ≡ 0) print_nl("!␣");
  else { print_nl("");
    print(full_source_filename_stack[level]);
    print_char(':');
    if (level ≡ in_open) print_int(line);
    else print_int(line_stack[level]);
    print(":␣");
  }
}
```

1722. The Format File. Most of the time TEX is not running as `initex` or `virtex`, but it runs with a format file preloaded. To set the format name, we first check if the format name was given on the command line with an "`&`" prefix, second we might check the first line of the input file, and last, we check if the program is an initex or virtex program.

If we still don't have a format, we use a plain format if running as a virtex, otherwise the program name is our best guess. There is no need to check for an extension, because the `kpathsearch` library will take care of that. We store the format file name in *dump_name* which is used in the function *w_open_in* below.

⟨ set the format name 1722 ⟩ ≡
```
if (parsefirstlinep ∧ ¬dump_name) parse_first_line(main_input_file);
if (¬main_input_file ∧ argv[1] ∧ argv[1][0] ≡ '&') dump_name = argv[1] + 1;
if (strcmp(kpse_program_name, "kinitex") ≡ 0) iniversion = true;
else if (strcmp(kpse_program_name, "kvirtex") ≡ 0 ∧ ¬dump_name) dump_name = "ktex";
if (¬dump_name) dump_name = kpse_program_name;
if (¬dump_name) { fprintf(stderr, "Unable␣to␣determine␣format␣name\n");
  exit(1);
}
if (ltxp) etexp = 1;
if (etexp ∧ ¬iniversion) { fprintf(stderr, "-etex␣and␣-ltx␣require␣-ini\n");
  exit(1);
}
```
This code is used in section 1684.

1723. Here is the function *parse_first_line*. It searches the first line of the file for a TEX comment of the form "%&format"[1]. If found, we will use the format given there.

⟨ TEX Live auxiliary functions 1686 ⟩ +≡
```
static void parse_first_line(char *filename)
{ FILE *f = Λ;
  if (filename ≡ Λ) return;
  f = open_in(filename, kpse_tex_format, "r");
  if (f ≠ Λ) { char *r, *s, *t = read_line(f);
    xfclose(f, filename);
    if (t ≡ Λ) return;
    s = t;
    if (s[0] ≡ '%' ∧ s[1] ≡ '&') { s = s + 2;
      while (ISBLANK(*s)) ++s;
      r = s;
      while (*s ≠ 0 ∧ *s ≠ '␣' ∧ *s ≠ '\r' ∧ *s ≠ '\n') s++;
      *s = 0;
      if (dump_name ≡ Λ) { char *f_name = concat(r, ".fmt");
        char *d_name = kpse_find_file(f_name, kpse_fmt_format, false);
        if (d_name ∧ kpse_readable_file(d_name)) { dump_name = xstrdup(r);
          kpse_reset_program_name(dump_name);
        }
        free(f_name);
      }
    }
    free(t);
  }
}
```

[1] The idea of using this format came from Wlodzimierz Bzyl.

1724. Commands. In the old days, TEX was a Pascal program, and standard Pascal did say nothing about a command line. So TEX would open the terminal file for input and read all the information from the terminal. If you don't give TEX command line arguments, this is still true today. In our present time, people got so much used to control the behavior of a program using command line arguments—especially when writing scripts—that TEX Live allows the specification of commands on the command line which TEX would normally expect on the first line of its terminal input.

So our next task is writing a function to add the remainder of the command line to TEX's input buffer. The main job is done by the *input_add_str* function which duplicates part of the *input_ln* function. Further it skips initial spaces and replaces trailing spaces and line endings by a single space.

⟨TEX Live auxiliary functions 1686⟩ +≡

```
static void input_add_char(unsigned int c)
{ if (last ≥ max_buf_stack) { max_buf_stack = last + 1;
    if (max_buf_stack ≡ buf_size) ⟨Report overflow of the input buffer, and abort 35⟩;
  }
  buffer[last] = xord[c];
  incr(last);
}
static void input_add_str(const char *str)
{ int prev_last;
  while (*str ≡ '␣') str++;
  prev_last = last;
  while (*str ≠ 0) input_add_char(*str++);
  for (−−last; last ≥ first; −−last) { char c = buffer[last];
    if ((c) ≠ '␣' ∧ (c) ≠ '\r' ∧ (c) ≠ '\n') break;
  }
  last++;
  if (last > prev_last) input_add_char('␣');
}
static int input_command_line(void)
{ last = first;
  while (optind < argc) input_add_str(argv[optind++]);
  loc = first;
  return (loc < last); }
```

1725. ⟨Forward declarations 52⟩ +≡

```
static int input_command_line(void);
```

1726. Opening Files. When we open an output file, there is usually no searching necessary. In the best case, we have an absolute path and can open it. If the path is relative, we try in this order: the *file_name* prefixed by the *output_directory*, the *file_name* as is, and the *file_name* prefixed with the environment variable `TEXMFOUTPUT`.

If we were successful with one of the modified names, we update *name_of_file*.

⟨TEX Live functions 1684⟩ +≡

```
static FILE *open_out(const char *file_name, const char *file_mode)
{ FILE *f = Λ;
  char *new_name = Λ;
  int absolute = kpse_absolute_p(file_name, false);
  if (absolute) { f = fopen(file_name, file_mode);
    if (f ≠ Λ) recorder_record_output(file_name);
    return f;
  }
  if (output_directory) { new_name = concat3(output_directory, DIR_SEP_STRING, file_name);
    f = fopen(new_name, file_mode);
    if (f ≡ Λ) { free(new_name);
      new_name = Λ; }
  }
  if (f ≡ Λ) f = fopen(file_name, file_mode);
  if (f ≡ Λ) { const char *texmfoutput = kpse_var_value("TEXMFOUTPUT");
    if (texmfoutput ≠ Λ ∧ texmfoutput[0] ≠ 0) {
      new_name = concat3(texmfoutput, DIR_SEP_STRING, file_name);
      f = fopen(new_name, file_mode);
      if (f ≡ Λ) { free(new_name);
        new_name = Λ; }
    }
  }
  if (f ≠ Λ ∧ new_name ≠ Λ) update_name_of_file(new_name, (int) strlen(new_name));
  if (f ≠ Λ) recorder_record_output((char *) name_of_file + 1);
  if (new_name ≠ Λ) free(new_name);
  return f;
}
static bool a_open_out(alpha_file *f)    /* open a text file for output */
{ f→f = open_out((char *) name_of_file + 1, "w");
  return f→f ≠ Λ ∧ ferror(f→f) ≡ 0; }
static bool b_open_out(byte_file *f)    /* open a binary file for output */
{ f→f = open_out((char *) name_of_file + 1, "wb");
  return f→f ≠ Λ ∧ ferror(f→f) ≡ 0; }
#ifdef INIT
static bool w_open_out(word_file *f)    /* open a word file for output */
{ f→f = open_out((char *) name_of_file + 1, "wb");
  return f→f ≠ Λ ∧ ferror(f→f) ≡ 0; }
#endif
```

1727. Format file names must be scanned before TEX's string mechanism has been initialized. The function *update_name_of_file* will set *name_of_file* from a C string.

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.

⟨TEX Live auxiliary functions 1686⟩ +≡
  static void *update_name_of_file*(**const char** ∗*s*, **int** *k*)
  { **int** *j*;
    if ($k \le file\_name\_size$) $name\_length = k$; **else** $name\_length = file\_name\_size$;
    for ($j = 0$; $j < name\_length$; $j{+}{+}$) $name\_of\_file[j+1] = xchr[(\mathbf{int})\ s[j]]$;
    $name\_of\_file[name\_length + 1] = 0$;
  }

1728. In standard TEX, the *reset* macro is used to open input files. The `kpathsearch` library uses different search paths for different types of files and therefore different functions are needed to open these files. The common code is in the function *open_in*.

⟨TEX Live auxiliary functions 1686⟩ +≡
  static FILE ∗*open_in*(**char** ∗*filename*, **kpse_file_format_type** *t*, **const char** ∗*rwb*)
  { **char** ∗*fname* = Λ;
    FILE ∗*f* = Λ;
    fname = *find_file*(*filename*, *t*, *true*);
    if (*fname* ≠ Λ) { *f* = *fopen*(*fname*, *rwb*);
      if (*f* ≠ Λ) *recorder_record_input*(*fname*);
      if (*full_name_of_file* ≠ Λ) *free*(*full_name_of_file*);
      full_name_of_file = *fname*; }
    return *f*;
  }
  static bool *a_open_in*(**alpha_file** ∗*f*) /∗ open a text file for input ∗/
  { *f*→*f* = *open_in*((**char** ∗) *name_of_file* + 1, *kpse_tex_format*, "r");
    if (*f*→*f* ≠ Λ) *get*(∗*f*);
    return *f*→*f* ≠ Λ ∧ *ferror*(*f*→*f*) ≡ 0;
  }
  static bool *b_open_in*(**byte_file** ∗*f*) /∗ open a binary file for input ∗/
  { *f*→*f* = *open_in*((**char** ∗) *name_of_file* + 1, *kpse_tfm_format*, "rb");
    if (*f*→*f* ≠ Λ) *get*(∗*f*);
    return *f*→*f* ≠ Λ ∧ *ferror*(*f*→*f*) ≡ 0;
  }
  static bool *w_open_in*(**word_file** ∗*f*) /∗ open a word file for input ∗/
  { *f*→*f* = Λ;
    if (*name_of_file*[1] ≠ 0) *f*→*f* = *open_in*((**char** ∗) *name_of_file* + 1, *kpse_fmt_format*, "rb");
    if (*f*→*f* ≠ Λ) *get*(∗*f*);
    return *f*→*f* ≠ Λ ∧ *ferror*(*f*→*f*) ≡ 0;
  }

1729. TEX's *open_fmt_file* function will call the following function either with the name of a format file as given with an "`&`" prefix in the input or with Λ if no such name was specified. The function will try *dump_name* as a last resort before returning Λ.

⟨TEX Live functions 1684⟩ +≡
```
static bool open_fmt_file(void)
{ int j = loc;
  if (buffer[loc] ≡ '&') { incr(loc);
    j = loc;
    buffer[last] = '␣';
    while (buffer[j] ≠ '␣') incr(j);
    update_name_of_file((char *) buffer + loc, j − loc);
    if (w_open_in(&fmt_file)) goto found;
  }
  update_name_of_file(dump_name, (int) strlen(dump_name));
  if (w_open_in(&fmt_file)) goto found;
  name_of_file[1] = 0;
  wake_up_terminal;
  wterm_ln("I␣can't␣find␣a␣format␣file!");
  return false;
found: loc = j;
  return true;
}
```

1730. The TEX Live infrastructure is able to generate format files, font metric files, and even some tex files, if required.

⟨enable the generation of input files 1730⟩ ≡
```
kpse_set_program_enabled(kpse_tfm_format, MAKE_TEX_TFM_BY_DEFAULT, kpse_src_compile);
kpse_set_program_enabled(kpse_tex_format, MAKE_TEX_TEX_BY_DEFAULT, kpse_src_compile);
kpse_set_program_enabled(kpse_fmt_format, MAKE_TEX_FMT_BY_DEFAULT, kpse_src_compile);
```
This code is used in section 1684.

1731. Date and Time. We conclude this chapter using `time.h` to provide a function that is used to initialize TEX's date and time information. Because *time* is one of TEX's macros, we add the function *tl_now* before including TEX's macros to wrap the call to the *time* function. It sets the variable *start_time* and returns a pointer to a *tm* structure to be used later in *fix_date_and_time*.

To support reproducible output, the environment variable `SOURCE_DATE_EPOCH` needs to be checked. If it is set, it is an ASCII representation of a UNIX timestamp, defined as the number of seconds, excluding leap seconds, since 01 Jan 1970 00:00:00 UTC. Its value is then used to initialize the *start_time* variable.

The TEX Live conventions further require that setting the `FORCE_SOURCE_DATE` environment variable to 1 will cause also TEX's primitives `\year`, `\month`, `\day`, and `\time` to use this value as the current time. Looking at the TEX Live code also reveals that these primitives use the local time instead of the GMT if this variable is not set to 1.

⟨ Header files and function declarations 9 ⟩ +≡

```
#include <time.h>
  static time_t start_time = ((time_t) −1);
  static char *source_date_epoch, *force_source_date;
#if defined (_MSC_VER) ∧ _MSC_VER < 1800
#define strtoull _strtoui64
#endif
  static struct tm *tl_now(void)
  { struct tm *tp;
    time_t t;
    source_date_epoch = getenv("SOURCE_DATE_EPOCH");
    force_source_date = getenv("FORCE_SOURCE_DATE");
    if (force_source_date ≠ Λ ∧ (force_source_date[0] ≠ '1' ∨ force_source_date[1] ≠ 0))
      force_source_date = Λ;
    if (source_date_epoch ≠ Λ) { start_time = (time_t) strtoull(source_date_epoch, Λ, 10);
      if (force_source_date ≠ Λ) t = start_time;
      else t = time(Λ);
    }
    else t = start_time = time(Λ);
    if (force_source_date) tp = gmtime(&t);
    else tp = localtime(&t);
    return tp;
  }
```

1732. Retrieving File Properties. To support LaTeX, a few more time related functions are needed.

⟨ Header files and function declarations 9 ⟩ +≡

```
#define TIME_STR_SIZE 30
  static char time_str[TIME_STR_SIZE];
  static void get_creation_date(void);
  static void get_file_mod_date(void);
  static int get_file_size(void);
#include <md5.h>
#define DIGEST_SIZE 16
#define FILE_BUF_SIZE 1024
  static md5_byte_t md5_digest[DIGEST_SIZE];
  static int get_md5_sum(int s, int file);
```

1733. The code that follows was taken from the `texmfmp.c` file of the TEX Live distribution and slightly modified.

```
⟨ TEX Live auxiliary functions 1686 ⟩ +≡
  static void make_time_str(time_t t, bool utc)
  { struct tm lt, gmt;
    size_t size;
    int off, off_hours, off_mins;    /* get the time */
    if (utc) { lt = *gmtime(&t);
    }
    else { lt = *localtime(&t);
    }
    size = strftime(time_str, TIME_STR_SIZE, "D:%Y%m%d%H%M%S", &lt);
      /* expected format: "D:YYYYmmddHHMMSS" */
    if (size ≡ 0) {    /* unexpected, contents of time_str is undefined */
      time_str[0] = '\0';
      return;
    }    /* correction for seconds: S can be in range 00 to 61, the PDF reference expects 00 to59, therefore
           we map "60" and "61" to "59" */
    if (time_str[14] ≡ '6') { time_str[14] = '5';
      time_str[15] = '9';
      time_str[16] = '\0';    /* for safety */
    }    /* get the time zone offset */
    gmt = *gmtime(&t);    /* this calculation method was found in exim's tod.c */
    off = 60 * (lt.tm_hour − gmt.tm_hour) + lt.tm_min − gmt.tm_min;
    if (lt.tm_year ≠ gmt.tm_year) { off += (lt.tm_year > gmt.tm_year) ? 1440 : −1440;
    }
    else if (lt.tm_yday ≠ gmt.tm_yday) { off += (lt.tm_yday > gmt.tm_yday) ? 1440 : −1440;
    }
    if (off ≡ 0) { time_str[size++] = 'Z';
      time_str[size] = 0;
    }
    else { off_hours = off/60;
      off_mins = abs(off − off_hours * 60);
      snprintf(&time_str[size], TIME_STR_SIZE − size, "%+03d'%02d'", off_hours, off_mins);
    }
  }
  static void get_creation_date(void)
  { make_time_str(start_time, source_date_epoch ≠ Λ);
  }    /* static structure for file status set by find_input_file */
#ifdef WIN32
  static struct _stat file_stat;
#define GET_FILE_STAT _stat (fname, &file_stat)
#else
  static struct stat file_stat;
#define GET_FILE_STAT stat (fname, &file_stat)
#endif
  static char *find_input_file(void)
  { char *fname;
    int r;
    if (output_directory ∧ ¬kpse_absolute_p((char *) name_of_file0, false)) { int r = −1;
```

```
      fname = concat3(output_directory, DIR_SEP_STRING, (char *) name_of_file0);
      r = GET_FILE_STAT;
      if (r ≡ 0) return fname;
      free(fname);
    }
    fname = kpse_find_tex((char *) name_of_file0);
    if (fname ≠ Λ) { r = GET_FILE_STAT;
      if (r ≡ 0) return fname;
      free(fname);
    }
    fname = (char *) name_of_file0;
    r = GET_FILE_STAT;
    if (r ≡ 0) return strdup(fname);
    return Λ;
  }
  static void get_file_mod_date(void)
  { char *fname = Λ;
    fname = find_input_file();
    time_str[0] = 0;
    if (fname ≠ Λ) { make_time_str(file_stat.st_mtime, source_date_epoch ≠ Λ ∧ force_source_date ≠ Λ);
      free(fname); }
  }
  static int get_file_size(void)
  { int s = −1;
    char *fname = Λ;
    fname = find_input_file();
    if (fname ≠ Λ) { s = file_stat.st_size;
      free(fname); }
    return s;
  }
  static int get_md5_sum(int s, int file){ md5_state_t st;
      memset(md5_digest, 0, DIGEST_SIZE);
      if (file) { char *fname;
        pack_file_name(s, empty_string, empty_string, Λ);
        fname = find_input_file();
        if (fname ≠ Λ) { FILE *f;
          f = fopen(fname, "rb");
          if (f ≠ Λ) { int r;
            char file_buf[FILE_BUF_SIZE];
            recorder_record_input(fname);
            md5_init(&st);
            while ((r = fread(&file_buf, 1, FILE_BUF_SIZE, f)) > 0)
              md5_append(&st, (const md5_byte_t*) file_buf, r);
            md5_finish(&st, md5_digest);
            fclose(f);
          }
          free(fname);
        }
        else return 0;
      }
```

else { *md5_init*(&*st*); *md5_append* (&*st*, (*md5_byte_t* ∗) &*str_pool*[*str_start*[*s*]], *str_start*[$s + 1$] − *str_start*[*s*]) ;
md5_finish(&*st*, *md5_digest*);
}
return `DIGEST_SIZE`;
}

1734. Profiling. If we want to find out where and how TEX spends its time, we need a profiler. The common profilers, like `gprof`, examine the executable code of TEX, then take some measurments at run time, and map this information to the functions or source lines of TEX's code.

But TEX is a programming language by itself, and when we program in TEX, we want to map the timing information gathered at run time to the macros and source lines of TEX's input files. The profiler implemented in the follwing code serves this purpose.

By default, profiling is initialy disabled. To enable profiling, you can use the `-prof` option on the command line.

⟨ explain the options 1688 ⟩ +≡
```
"␣-prof␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"
    "\t␣enable␣profiling\n"
```

1735. This option will set the *profile_on* variable, defined below, to true.

⟨ more options 1735 ⟩ ≡
{"prof", 0, &*profile_on*, 1} ,

This code is used in section 1690.

1736. ⟨ Global variables 13 ⟩ +≡
static int *profile_on* = *false*;

1737. To restrict profiling to selected parts of TEX's input, we define two primitives for switching profiling on and off.

#**define** *profile_on_code* (*eTeX_last_extension_cmd_mod* + 3) /∗ '\profileon' ∗/
#**define** *profile_off_code* (*eTeX_last_extension_cmd_mod* + 4) /∗ '\profileoff' ∗/

⟨ Put each of TEX's primitives into the hash table 225 ⟩ +≡
primitive("profileon", *extension*, *profile_on_code*);
primitive("profileoff", *extension*, *profile_off_code*);

1738. ⟨ Cases of *extension* for *print_cmd_chr* 1604 ⟩ +≡
case *profile_on_code*: *print_esc*("profileon"); **break**;
case *profile_off_code*: *print_esc*("profileoff"); **break**;

1739. ⟨ Cases for *do_extension* 1607 ⟩ +≡
case *profile_on_code*:
  if (¬*profile_on*) { *profile_on* = *true*;
    ⟨ record profiling on 1758 ⟩
  }
  break;
case *profile_off_code*:
  if (*profile_on*) { ⟨ record profiling off 1757 ⟩
    profile_on = *false*;
  }
  break;

1740. Files and Lines. The "instructions" that drive the TeX interpreter are called "tokens" and we want to keep track of the files and the lines where the tokens come from. If a token comes from an external file, then TeX's variable *line* will contain the correct line number, and *cur_file* contains the current file. *cur_file* is an abbreviation for *input_file*[*index*] with *index* beeing a short hand for *cur_input.index_field*. We can not use the *index* number to identify input files, because TeX will reuse these numbers, possibly many times, after a file has been closed. Instead we make a new array *input_file_num* that runs in parallel to *input_file* and contains a unique current file number. Further we make a new *read_file_num* array that runs in parallel to *read_file*.

#**define** *cur_file_num* *input_file_num*[*index*] /∗ the current unique **alpha_file** number ∗/

⟨ Global variables 13 ⟩ +≡
  static uint8_t *input_file_num0*[*max_in_open*], ∗**const** *input_file_num* = *input_file_num0* − 1;
  static uint8_t *read_file_num*[20];

1741. To implement the association between unique file numbers and full file names, we use the array *file_num_name*. We will store file numbers with 8 bits and line numbers with 16 bits in a single 32 bit number. We do this because we will need to store this information together with the tokens in a token list. This representation will keep the most significant byte zero, so that we can later use it for the "command" value.

⟨ Global variables 13 ⟩ +≡
#**define** `FILE_NUM_BITS` 8
#**define** `MAX_FILE_NUM` ((1 ≪ `FILE_NUM_BITS`) − 1)
#**define** `LINE_BITS` 16
#**define** `MAX_LINE` ((1 ≪ `LINE_BITS`) − 1)
#**define** `FILE_LINE` (*F*, *L*) (((*F*) ≪ (`LINE_BITS`)) | (*L*))
#**define** `FL_FILE` (`FL`) ((`FL`) ≫ `LINE_BITS`)
#**define** `FL_LINE` (`FL`) ((`FL`) & `MAX_LINE`)
  static char ∗*file_num_name*[`MAX_FILE_NUM` + 1];
  static int *file_from_cmd*[`MAX_FILE_NUM` + 1]; /∗ first command from the file ∗/
  static int *file_to_cmd*[`MAX_FILE_NUM` + 1]; /∗ last command from the file ∗/
  static int *file_num*; /∗ the the last *file_num_name* allocated ∗/
  static uint32_t *cur_file_line* = 0; /∗ the file and line for *cur_tok* ∗/

1742. ⟨ check *line* for overflow 1742 ⟩ ≡
  if (*line* ≥ `MAX_LINE`) *overflow*(`"too␣many␣input␣lines"`, `MAX_LINE`);

This code is used in section 361.

1743. Whenever we have a new *full_name_of_file*, we allocate a new unique file number.

⟨ Allocate a file number 1743 ⟩ ≡
  if (*file_num* ≥ `MAX_FILE_NUM`) *overflow*(`"file␣number"`, *file_num*);
  else *file_num*++;
  file_num_name[*file_num*] = *strdup*(*full_name_of_file*);
  file_from_cmd[*file_num*] = *cmd_count*;

This code is used in sections 1744 and 1745.

1744. Whenever TeX sets *cur_file* to a new value, we need to set *cur_file_num* as well.

⟨ Set new *cur_file_num* 1744 ⟩ ≡
  ⟨ Allocate a file number 1743 ⟩
  cur_file_num = *file_num*;

This code is used in section 536.

1745. Whenever TEX opens *read_file*[n] it needs to set the file count.

⟨ Set new *read_file_num*[n] 1745 ⟩ ≡
  ⟨ Allocate a file number 1743 ⟩
  read_file_num[n] = *file_num*;

This code is used in section 1274.

1746. A few file numbers are predefined. Some tokens, for instance those that are loaded from a format file, have an unknown origin and are classified as comming from an *unknown_file*. We use *system_file* to assign time intervals to the system and use the command field to distinguish between different system events. There might also be tokens from the terminal, either typed in interactively (which will messup your timing values) or from the command line.

For the commands from the *system_file*, we use special line numbers to distinguish various cases.

#define *unknown_file* 0 /∗ tokens with unknown origin, usualy from a format file ∗/
#define *system_file* 1 /∗ tokens generated by the system ∗/
#define *terminal_file* 2 /∗ tokens read from the terminal ∗/
#define *system_unknown* 0 /∗ should not happen ∗/
#define *system_start* 1 /∗ startup of TEX ∗/
#define *system_end* 2 /∗ end of TEX ∗/
#define *system_ship_out* 3 /∗ time spent in *ship_out*( ) ∗/
#define *system_line_break* 4 /∗ time spent in *line_break*( ) ∗/
#define *system_init_trie* 5 /∗ time spent in *line_break*( ) ∗/
#define *system_build_page* 6 /∗ time spent in *line_break*( ) ∗/
#define *system_input_ln* 7 /∗ time spent in *line_break*( ) ∗/
#define *system_insert* 8 /∗ time spent on tokens inserted by TEX ∗/

⟨ Set initial values of key variables 21 ⟩ +≡
  file_num_name[*unknown_file*] = "unknown";
  file_num_name[*system_file*] = "system";
  file_num_name[*terminal_file*] = "terminal";
  file_num = *terminal_file*;

1747. When TEX reads a token from an external file, it has to set *cur_file_line* based on the information in *cur_input*. This code is parallel to the code that is used to read the next line:

⟨ Set *cur_file_line* based on the information in *cur_input* 1747 ⟩ ≡
  if (*terminal_input*) *cur_file_line* = FILE_LINE(*terminal_file*, 0);
  else if (*name* $\le$ 19) *cur_file_line* = FILE_LINE(*read_file_num*[*name* $-$ 1], 0);
    /∗ do I need the line? ∗/
  else *cur_file_line* = FILE_LINE(*cur_file_num*, *line*);

This code is used in section 342.

1748. When we end a `\read` line, we set the current file and line.

⟨ Set *cur_file_line* when a `\read` line ends 1748 ⟩ ≡
  cur_file_line = FILE_LINE(*read_file_num*[*name* $-$ 1], 0);

This code is used in section 359.

1749. As described above, we store file and line numbers as 32 bit integers. When reading input from external files, we know the file and the line; but for token lists we need to store this information together with the tokens, when we create the token list. As a simple solution, we allocate a 32 bit integer for all pointers.

Whenever in TeX's code a new token is generated, we know what file and line is responsible for what is going on. The newly generated token will then point to the same file and line.

⟨ Global variables 13 ⟩ +≡
static uint32_t *fl_mem0* [*mem_max* − *mem_min* + 1], ∗**const** *fl_mem* = *fl_mem0* − *mem_min*;
/∗ the big dynamic file/line storage ∗/

1750. Timing. When the interpreter of TEX executes a command, we need to measure the time needed to execute it and record that information in a file. We call such a measurement a "time stamp" or "stamp" for short. This process should be quick, in order not to disturb the measurements too much by the time needed for the measurement itself and its recording. So we keep the data in a large array, and postpone writing the output file until TEX terminates.

Now let us consider the measurement of such a time stamp. As a general rule, the start of a new time interval is the end of the previous time interval. There is only a single look up of the current time for both. So all time intervals will add up to the total run time.

We keep the start time in two variables *start_sec* and *start_nsec*; the first counts the seconds, the other (roughly) the nano seconds. When we measure the end time, we should know the previous start time, and which command was executed and from which file and line this command was taken. The measured end time is the start time for the next command.

```
⟨ Global variables 13 ⟩ +≡
  static long start_nsec, start_sec, diff_nsec;
  struct timespec ts;
  static int time_error;
```

1751. To get timing information, we use the *clock_gettime* function. Depending on the operating system there are many different variations. The timer used for timing can be selected by defining `GETTIME` at compile time. The default is `GETTIME` ≡ 0 which selects a tread specific clock.

```
⟨ get current time 1751 ⟩ ≡
#if GETTIME ≡ 0
  time_error = clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts);
#elif GETTIME ≡ 1
  time_error = clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
#elif GETTIME ≡ 2
  time_error = clock_gettime(CLOCK_MONOTONIC_RAW, &ts);
#elif GETTIME ≡ 3
  time_error = clock_gettime(CLOCK_MONOTONIC, &ts);
#else
  time_error = clock_gettime(CLOCK_THREAD_CPUTIME_ID, &ts);
#endif
```

This code is used in sections 1754, 1755, 1756, and 1758.

1752. To record the timing information, we compute the time difference in nano seconds, assuming that it fits into unsigned 32 bit which means the difference should be less than about 4 seconds.

```
⟨ compute elapsed time 1752 ⟩ ≡
  diff_nsec = (ts.tv_sec − start_sec) ∗ 1000000000 + (ts.tv_nsec − start_nsec);
#if 0
  printf("time:␣%ld␣%ld␣%ld\n", diff_nsec, (ts.tv_sec), (ts.tv_nsec));
#endif
  start_nsec = ts.tv_nsec;
  start_sec = ts.tv_sec;
```

This code is used in sections 1755 and 1756.

1753. Commands. The fourth piece of information that we store with file, line, and time is the command that TEX is executing. The elapsed time will fit into an four byte unsigned integer, and luckily also the command, the file and the line will do as well. When starting a new time interval, often we will not yet know the next command, its file, and its line. We keep the command, the file, the line, in a set of variables starting with *prof_*. These variables my change within a time interval, and the final values are written when the end time is known. There are also variables related to macro calls which are explainded later but are already defined here because we will initialize them together with the other variables.

⟨ Global variables 13 ⟩ +≡
```
#define MAX_STAMPS 80000000
#define MAX_MACRO_STACK 40000
#define POP_BIT #80
  static uint32_t stamp[MAX_STAMPS];   /* one chunk of profiling information */
  static uint32_t st_count = 0;   /* the next available entry in the array above */
  static uint32_t cmd_count = 0;   /* the number of time stamps in the array above */
  static int prof_cmd = 0;   /* the command to be written */
  static uint32_t prof_file_line = 0;   /* the file and line to be written */
```

1754. The code below initializes the variables. Note that we use the special command value *system_cmd*. The "real" command values are those that occur in the *big_switch* ranging from 0 (*escape*) to 100 (*max_command*), the values that are used when entering the *main_loop*(*letter*, *other_char*, *char_given*, and *char_num*) are a subset of these. The pseudo command *system_cmd* is just outside of this range. A few more pseudo commands are needed below and we define them here.

```
#define system_cmd (max_command + 1)   /* pseudo command value */
#define system_profile_on (system_cmd + 1)
#define system_profile_off (system_cmd + 2)
#define system_macro_push (system_cmd + 3)
#define system_macro_pop (system_cmd + 4)
#define system_macro_pop_small (system_cmd + 5)
#define system_macro_pop_0 (system_cmd + 6)
```

⟨ Initialize profiling 1754 ⟩ ≡
```
  ⟨ Initialize the macro call stack 1769 ⟩
  prof_cmd = system_cmd;
  prof_file_line = FILE_LINE(system_file, system_start);
  ⟨ record macro call information if necessary 1774 ⟩
#if 0
  clock_getres(CLOCK_MONOTONIC, &ts);
  printf("CLOCK_MONOTONIC:␣%ld␣%ld\n", ts.tv_sec, ts.tv_nsec);
  clock_getres(CLOCK_MONOTONIC_RAW, &ts);
  printf("CLOCK_MONOTONIC_RAW:␣%ld␣%ld\n", ts.tv_sec, ts.tv_nsec);
  clock_getres(CLOCK_PROCESS_CPUTIME_ID, &ts);
  printf("CLOCK_PROCESS_CPUTIME_ID:␣%ld␣%ld\n", ts.tv_sec, ts.tv_nsec);
  clock_getres(CLOCK_THREAD_CPUTIME_ID, &ts);
  printf("CLOCK_THREAD_CPUTIME_ID:␣%ld␣%ld\n", ts.tv_sec, ts.tv_nsec);
#endif
  ⟨ get current time 1751 ⟩
  diff_nsec = 0;
  start_nsec = ts.tv_nsec;
  start_sec = ts.tv_sec;
```
This code is used in section 1029.

1755. At the end of TEX we check *last_depth* and clean the remaining macro stack if necessary. We end the record always with recording timing information.

⟨ pop the macro call stack at the end of TEX 1755 ⟩ ≡
  prof_cmd = *cur_cmd*; /* the stop command */
  prof_file_line = *cur_file_line*;
  prof_depth = 0;
  profile_on = *true*;
  ⟨ get current time 1751 ⟩ /* advance the time */
  ⟨ compute elapsed time 1752 ⟩
  ⟨ record timing information 1756 ⟩ /* pop to *prof_depth* */

This code is used in section 1759.

1756. When we record the timing information, we first store command, file, and line, and then store the elapsed time.

#define SHOW_RECORD_TIMING 0

⟨ record timing information 1756 ⟩ ≡
  if (*profile_on*) {
    if (*st_count* + 3 > MAX_STAMPS) *overflow*("profile␣data", MAX_STAMPS);
    ⟨ record poping the macro stack if necessary 1775 ⟩
    stamp[*st_count*++] = (*prof_cmd* ≪ 24) | *prof_file_line*;
    prof_cmd = *prof_cmd* & ∼POP_BIT; /* remove a possible POP_BIT */
#if SHOW_RECORD_TIMING
    print_char('<');
    print_int(*cmd_count*);
    print_char(':');
    if (*prof_cmd* & POP_BIT) *print_char*('-');
    print_int(*prof_cmd* & ∼POP_BIT);
    print_char(':');
    print_int(*prof_depth*);
    print_char('>');
    print_ln( );
#endif
    cmd_count++;
    ⟨ get current time 1751 ⟩
    ⟨ compute elapsed time 1752 ⟩
    stamp[*st_count*++] = *diff_nsec*;
  }

This code is used in sections 1029, 1755, 1757, 1762, 1763, 1764, 1765, 1766, and 1767.

1757. To mark the execution of starting and ending profiling we use the system commands for the start and the end of profiling.

After switching profiling off, no more data is written to the output file. So we have to record the macro call and timing information explicitly before switching the profiling off.

⟨ record profiling off 1757 ⟩ ≡
  prof_file_line = *cur_file_line*;
  prof_cmd = *system_profile_off*;
  prof_depth = *cur_depth*;
  ⟨ record macro call information if necessary 1774 ⟩
  ⟨ record timing information 1756 ⟩

This code is used in section 1739.

1758. When switching profiling on, we have to syncronize the start time with the current time and prepare everything for the recording of the timing before entering the main loop again. Setting the command, file and line information is simple. Since we can not assume that the macro call stack is unchanged, we record the necessary changes.

⟨record profiling on 1758⟩ ≡
  ⟨get current time 1751⟩
  diff_nsec = 0;
  start_nsec = *ts.tv_nsec*;
  start_sec = *ts.tv_sec*;
  prof_file_line = *cur_file_line*;
  prof_cmd = *system_profile_on*;
  prof_depth = *cur_depth*;
  ⟨record macro call information if necessary 1774⟩

This code is used in section 1739.

1759. At the end of TEX we record the timing for the *stop* command and pop the macro call stack.

⟨record the end of TEX 1759⟩ ≡
  ⟨pop the macro call stack at the end of TEX 1755⟩

This code is used in section 1044.

1760. There are two places where we regularly record the next command. The first one is when TEX enters the big switch. Here we record the timing information before reading the next expanded token and set the current file, line, and command after reading the next expanded token. The other place is at the start of the main loop. Here both actions are done one after the other. Whenever we assiciate a command with the current time slot by setting *prof_cmd* and the profile data associated with it, we need to make sure that changes of entries in the macro call are reported to the profiler.

⟨set current file, line, and command for the current time slot 1760⟩ ≡
  if (*profile_on*) { *prof_cmd* = *cur_cmd*;
    prof_depth = *cur_depth*;
    prof_file_line = *cur_file_line*;
    ⟨record macro call information if necessary 1774⟩
  }

This code is used in section 1029.

1761. Some time expensive routines, *line_break* and *ship_out*, consume time that depends entirely on the output generated by TEX but not on how that output is generated. So we charge the time spent in these routines on the system.

⟨Local variables to save the profiling context 1761⟩ ≡
  int *tmp_cmd*;
  int *tmp_file_line*;
  int *tmp_depth*;

This code is used in sections 637, 814, 965, and 993.

1762. ⟨Charge the time used here on *line_break* 1762⟩ ≡

if (*profile_on*) { ⟨record timing information 1756⟩
 tmp_file_line = *prof_file_line*;
 tmp_cmd = *prof_cmd*;
 tmp_depth = *prof_depth*;
 prof_file_line = `FILE_LINE`(*system_file*, *system_line_break*);
 prof_cmd = *system_cmd*;
}

This code is used in section 814.

1763. ⟨restore the previous current file, line, and command 1763⟩ ≡

if (*profile_on*) { ⟨record timing information 1756⟩
 prof_file_line = *tmp_file_line*;
 prof_cmd = *tmp_cmd*;
 prof_depth = *tmp_depth*;
}

This code is used in sections 637, 814, 965, 993, and 1004.

1764. ⟨Charge the time used here on *init_trie* 1764⟩ ≡

if (*profile_on*) { ⟨record timing information 1756⟩
 tmp_file_line = *prof_file_line*;
 tmp_cmd = *prof_cmd*;
 tmp_depth = *prof_depth*;
 prof_file_line = `FILE_LINE`(*system_file*, *system_init_trie*);
 prof_cmd = *system_cmd*;
}

This code is used in section 965.

1765. ⟨Charge the time used here on *build_page* 1765⟩ ≡

if (*profile_on*) { ⟨record timing information 1756⟩
 tmp_file_line = *prof_file_line*;
 tmp_cmd = *prof_cmd*;
 tmp_depth = *prof_depth*;
 prof_file_line = `FILE_LINE`(*system_file*, *system_build_page*);
 prof_cmd = *system_cmd*;
}

This code is used in section 993.

1766. ⟨Charge the time used here on *ship_out* 1766⟩ ≡

if (*profile_on*) { ⟨record timing information 1756⟩
 tmp_file_line = *prof_file_line*;
 tmp_cmd = *prof_cmd*;
 tmp_depth = *prof_depth*;
 prof_file_line = `FILE_LINE`(*system_file*, *system_ship_out*);
 prof_cmd = *system_cmd*;
}

This code is used in section 637.

1767. The following code is currently not used, because input a line is pretty fast.

⟨ Charge the time used here on *input_ln* 1767 ⟩ ≡
  if (*profile_on*) { ⟨ record timing information 1756 ⟩
    tmp_file_line = *prof_file_line*;
    tmp_cmd = *prof_cmd*;
    tmp_depth = *prof_depth*;
    prof_file_line = FILE_LINE(*system_file*, *system_input_ln*);
    prof_cmd = *system_cmd*;
  }

1768. Macro Calls. Let's talk about collecting information about macro calls. Macro calls come in various flavours: There are active characters and single letter control sequences; both have their own region in the *eqtb* table of equivalents. Then there is an empty control sequence, and last there are lots of (currently up to 45000) multiletter control sequences. The plain format defines about 1000 and the LaTeX format about 23000 of them. When *get_next* finds a control sequence, it sets *cur_cs* to the index in the table of equivalents. It will further set $cur\_cmd = eq\_type(cur\_cs)$, $cur\_chr = equiv(cur\_cs)$, and $cur\_tok = cs\_token\_flag + cur\_cs$.

The *get_x_token* or *expand* procedure will then notice the macro call and invoke *macro_call*. *macro_call* will inspect the call site and the parameter section of the macro and put the actual macro parameters on the parameter stack before pushing the macro body on the input stack.

But wait! *macro_call* will apply a technique called last call optimization before doing any of the above: It will remove all entries from the input stack that have no more tokens left for processing. This allows tail recursive macro calls in constant stack space and makes long recursive loops possible. For example building the LaTeX format uses a loop to read the definition of UTF8 codes. This loop has more than 35000 iterations and a nesting depth of the same size. With last call optimization, the loop uses less than 3000 entries on the input stack.

Popping the stack early makes it slightly more difficult to track who is calling whom—an information that we need if we want to give cumulative timing for a macro call. Cumulative timing is the time spent during that macro call including all the calls to sub-macros.

The timing information that we collected in the previous section assigned the time caused by the execution of TeX's commands to the specific line and file that caused the execution of that command. As a consequnce, the time used for the expansion of a macro call is assigned to the file and line that contains the call site of the macro. The time that is spent on executing the commands in the macros body are assigned to the file and line containing the macro definition. But there is no information on the call graph that allows us to classify one command as comming from a lower level of expansion than the other.

In this section, we want to assign the time used for the execution of commands to a macro definition, with its name and the file and line that contains the definition of the macro. While the name of a macro is an important information, it is not sufficient to relate the time to the macros name, because the same name can be used for several different macros at various times.

So we collect for a macro call the following data:

1) The index *cur_cs* into the table of equivalents *eqtb*. Using *cur_cs* it is possible to access the macros name. For active characters and single letter control sequences, we obtain the character by subtracting *active_base* or *single_base* from *cur_cs*. For multi-letter control sequences, we find the index to *str_start* for the macro name in $text(cur\_cs)$, the text field of the corresponding hash table entry. The association between *cur_cs* and the macro name is stored in a separate section of the profilers output file.
2) The line and file of the macro definition. We store this information in the *fl_mem* array using $equiv(cur\_cs)$, the pointer to the reference count token of the macro.
3) The "unoptimized" nesting depth of the macro call.

We store this information in a 64 bit integer and define macros to assemble and disassemble this integer. From this information it is possible to reconstruct the call graph and compute cumulative timing information for macro calls. We keep the information for the whole "unoptimized" macro stack in the *macro_stack*. The varable *prof_depth* tells us the nesting depth of *prof_cmd*. The variable *last_depth* tells us the nesting depth of the last recorded time stamp and the variable *unchanged_depth* tells us which part of the macro stack is unchanged since then.

#**define** `CALL_DEPTH`(A) $((A) \gg 48)$
#**define** `CALL_EQTB`(A) $(((A) \gg 32)$ & #`FFFF`)
#**define** `CALL_FILE`(A) $(((A) \gg 16)$ & #`FF`)
#**define** `CALL_LINE`(A) $((A)$ & #`FFFF`)
#**define** `CALL_DE`(A) $((A) \gg 32)$
#**define** `CALL_CFL`(A) $((A)$ & #`FFFFFFFF`)

#define CALL_INFO(D, C, FL)
$(((uint64\_t)(D) \ll 48) \mid ((uint64\_t)(C) \ll 32) \mid (system\_macro\_push \ll 24) \mid \mathtt{FL})$

⟨ Global variables 13 ⟩ +≡
static *uint64_t macro_stack*[MAX_MACRO_STACK] = {0}; /* the macro calls */
static int *prof_depth*; /* the nesting depth of macros */
static int *last_depth*; /* the nesting depth last report1ed */
static int *unchanged_depth*; /* up to here no change on *macro_stack* */
static int *prof_max_depth*;

1769. ⟨ Initialize the macro call stack 1769 ⟩ ≡
{ **int** *macro_depth* = −1, *macro_cs* = 0;
uint32_t *macro_fl* = FILE_LINE(*terminal_file*, 0);
⟨ update the macro stack 1771 ⟩
cmd_count++;
stamp[*st_count*++] = CALL_CFL(*macro_stack*[0]);
stamp[*st_count*++] = CALL_DE(*macro_stack*[0]);
last_depth = *unchanged_depth* = *prof_depth* = 0;
}

This code is used in section 1754.

1770. We maintain the macro stack in the *macro_call* procedure by adding local variables that capture the information of the call token that cause the invocation of *macro_call*.

⟨ additional local variables for *macro_call* 1770 ⟩ ≡
uint32_t *macro_fl* = *fl_mem*[*cur_chr*];
int *macro_cs* = *cur_cs*;
int *macro_depth* = *cur_depth*;

This code is used in section 388.

1771. At the end of *macro_call*, we use this information to update the *macro_stack* and related variables.

⟨ update the macro stack 1771 ⟩ ≡
macro_depth++;
cur_depth = *macro_depth*;
if (*macro_depth* ≥ MAX_MACRO_STACK) *overflow*("macro␣stack␣size", *macro_depth*);
macro_stack[*macro_depth*] = CALL_INFO(*macro_depth*, *macro_cs*, *macro_fl*);
if (*macro_depth* ≤ *unchanged_depth*) *unchanged_depth* = *macro_depth* − 1;

This code is used in sections 322, 388, 536, and 1769.

1772. We add files as part of the macro stack using the following code:

⟨ additional local variables for *start_input* 1772 ⟩ ≡
uint32_t *macro_fl* = FILE_LINE(*cur_file_num*, 0);
int *macro_cs* = 0;
int *macro_depth* = *cur_depth*;

This code is used in section 536.

1773. We add calls to the output routine and hook macros like `\everypar`, `\everymath`, etc. to the macro stack using tis code:

⟨ additional local variables for *begin_token_list* 1773 ⟩ ≡
  uint32_t *macro_fl* = *fl_mem*[*p*];
  int *macro_cs* = *undefined_control_sequence* + *t*;
    /* *undefined_control_sequence* is the last pointer in the hash table */
  int *macro_depth* = *cur_depth*;

This code is used in section 322.

1774. As we have seen above, recording timing information consists of two steps: In the first step we set global variables like *prof_cmd* and *prof_depth* based on the current command. Then we wait until the command is executed, and in the second step we compute the elapsed time and store the information from the first step together with the time in the *stamp* array. Now we augment this process and store macro call related information.

The key value that guides this process is *prof_depth*: the macro nesting depth associated with *prof_cmd*. Using it we can get from the *macro_stack* information about all currently active macros at lower nesting levels. Two more variables *last_depth* and *unchanged_depth* are related to it: After recording macro call information for a command, we set *last_depth* = *unchanged_depth* = *prof_depth*. The next time we need to record macro call information, *unchanged_depth* and *prof_depth* may have changed, but *last_depth* should be unchanged. Because *unchanged_depth* can only decrease, we know that *unchanged_depth* ≤ *last_depth*. On the other hand, *prof_depth* can increase as well as decrease. This gives us the following cases:

d_depth < *prof_depth* Record the new stack entries from *unchanged_depth* + 1 up to *prof_depth*.

l_depth = *last_depth*: No information on macro calls is necessary.

l_depth = *last_depth*: Record that the stack was popped from *last_depth* down to *prof_depth*.

l_depth < *last_depth*: Record that the stack was popped from *last_depth* down to *prof_depth*.

The common case that *prof_dept* = *last_dept* − 1 asks for a more compact representation and we use the most significant bit of the command value to encode this case. Note that we count this as a separate command.

It would be nice if we could wait with recording the stack changes until the command is completely executed. The command however might read parameters which causes pops and pushes on the macro stack. So by the time the command has finished, the stack might have changed considerably. There is a further complication: the execusion of some commands does not end with a jump to *big_switch* but with a jump to *big_reswitch*. This will set a new value for *prof_cmd* and so will invalidate all our preparations. For this reason we need to make the recording of macro call information idempotent. Recording several pops and pushes explicitly does not harm (except for extra output) but setting the pop bit asks for trouble. Therefore we postpone the recording of poping the macro stack.

⟨ record macro call information if necessary 1774 ⟩ ≡
  if (*unchanged_depth* < *prof_depth*) ⟨ record the new stack entries 1776 ⟩

This code is used in sections 1754, 1757, 1758, and 1760.

1775.

⟨ record poping the macro stack if necessary 1775 ⟩ ≡

```
if (last_depth > prof_depth) {
    if (last_depth ≡ prof_depth + 1) prof_cmd = prof_cmd | POP_BIT;
    else stamp[st_count++] = (system_macro_pop ≪ 24) | (last_depth − prof_depth);
#if SHOW_RECORD_TIMING
    print_char('{');
    print_int(cmd_count);
    print_char(':');
    print_int(system_macro_pop);
    print_char(':');
    print_int(last_depth);
    print_char('>');
    print_int(prof_depth);
    print_char('}');
    print_ln();
#endif
    cmd_count++;
    last_depth = unchanged_depth = prof_depth;
}
```

This code is used in section 1756.

1776. The information about new stack entries are taken from the *macro_stack*.

⟨ record the new stack entries 1776 ⟩ ≡

```
{ int i;
  if (last_depth > unchanged_depth) {
#if SHOW_RECORD_TIMING
    print_char('{');
    print_int(cmd_count);
    print_char(':');
    print_int(system_macro_pop);
    print_char(':');
    print_int(last_depth);
    print_char('>');
    print_int(unchanged_depth);
    print_char('}');
    print_ln();
#endif
    cmd_count++;
  }
  if (st_count + 2 * (prof_depth − unchanged_depth) > MAX_STAMPS)
    overflow("profile␣data", MAX_STAMPS);
  for (i = unchanged_depth + 1; i ≤ prof_depth; i++) {
#if SHOW_RECORD_TIMING
    print_char('[');
    print_int(cmd_count);
    print_char(':');
    print_int(i − 1);
    print_char('<');
    print_int(CALL_DEPTH(macro_stack[i]));
    print_char(':');
    print_cs(CALL_EQTB(macro_stack[i]));
    print_char(']');
    print_ln();
#endif
    cmd_count++;
    stamp[st_count++] = CALL_CFL(macro_stack[i]);
    stamp[st_count++] = CALL_DE(macro_stack[i]);
  }
  last_depth = unchanged_depth = prof_depth;
  if (prof_depth > prof_max_depth) prof_max_depth = prof_depth;
}
```

This code is used in section 1774.

1777. Output the profile data. Here is the code that outputs the results. The format tries to be simple but flexible. The output file has four parts: size data, file names, macro names, and time stamps. To enable some error checking, the three file parts start and end with an output marker.

```
⟨ Finish the extensions 1377 ⟩ +≡
  { FILE *prof;
  pack_job_name(".tprof");
  prof = fopen((char *) name_of_file0, "wb");
  if (prof ≡ Λ) print_err("Unable␣to␣write␣the␣profile␣data\n");
  else { int i;
  prof_file_line = cur_file_line;
  ⟨ output marker 1778 ⟩
  ⟨ output size data 1779 ⟩
  ⟨ output marker 1778 ⟩
  ⟨ output file names 1780 ⟩
  ⟨ output marker 1778 ⟩
  ⟨ output macro names 1781 ⟩
  ⟨ output marker 1778 ⟩
  ⟨ output timing data 1783 ⟩⟨ output marker 1778 ⟩
  fclose(prof);
  }
  }
```

1778. The output marker consists of eight byte: the ASCII codes of "TEX PROF".

```
⟨ output marker 1778 ⟩ ≡
  fputs("TEX␣PROF", prof);
```

This code is used in section 1777.

1779. The information necessary to process the file comes after the first marker. Here and in the following all numbers are written out in "big-endian byte order". We provide the following information:

The number of files (2 byte), the number of byte used for filenames including the zero bytes (2 byte), the number of time stamps (4 byte). The number of macro names (2 byte), the number of bytes used for macro names (4 byte), again including the zero bytes, and the maximum nesting depth used on the *macro_stack*.

We start with the macros to write multibyte integers in bigendian order.

#define PUT1(N) *fputc*((N) & #FF, *prof*)
#define PUT2(N) PUT1((N) ≫ 8), PUT1(N)
#define PUT4(N) PUT2((N) ≫ 16), PUT2(N)

⟨ output size data 1779 ⟩ ≡

```
PUT2(file_num + 1);   /* the number of files */
{ int n, m;
  m = 0;
  for (i = 0; i ≤ file_num; i++) { n = strlen(file_num_name[i]);
    m += n + 1;
  }
  PUT2(m);   /* the number of byte used for file names */
}
PUT4(cmd_count);   /* the number of commands */
{ int k, m;
  m = k = 0;
  for (i = hash_base; i < undefined_control_sequence; i++)
    if (text(i) ≠ 0) { k++;
      m += (str_start[text(i) + 1] − str_start[text(i)]) + 1;
    }
  PUT2(k);   /* the number of macro names */
  PUT4(m);   /* the number of byte for the macro names */
  PUT2(prof_max_depth);   /* the maximum macro nesting depth */
}
```

This code is used in section 1777.

1780. Writing the file names is simple.

⟨ output file names 1780 ⟩ ≡

```
for (i = 0; i ≤ file_num; i++) { fputs(file_num_name[i], prof);
  PUT1(0);
}
```

This code is used in section 1777.

1781. Here is the code to write the macro names to the output file. It traverses the *hash_table* from *hash_base* up to *undefined_control_sequence*. Each macro name is preceeded by its index in the hash table stored in two byte.

⟨ output macro names 1781 ⟩ ≡

```
for (i = hash_base; i < undefined_control_sequence; i++)
  if (text(i) ≠ 0) { int k;
    PUT2(i);
    for (k = str_start[text(i)]; k < str_start[text(i) + 1]; k++) PUT1(str_pool[k]);
    PUT1(0);
  }
```

This code is used in section 1777.

1782. Next we output the time stamps. We use a variable format depending on the command value which is always in the top byte of the stamp. We optimize the output size slightly: For macro calls the *stamp* array contains the depth values. It is of course more efficient to write the relative changes into the output instead of the absolute values. A most common case is poping the stack by one. This is recorded in the *stamp* array by setting the POP_BIT in the command value and we simply keep this bit. Small relative values n in the range $0 \le n \le 10$ are encoded using the command value *system_macro_pop_0* $+ n$. Values in the range $11 \le n \le 255$ encoded as single byte values following the commmand code *system_macro_pop_small*. All other values are encoded using two byte following the command code *system_macro_pop*. Encoding the pops separate from the calls, we need no longer store the depth values with the calls.

⟨encode pop n 1782⟩ ≡

```
#if SHOW_WRITE_TIMING
  printf("!%d:%d:%d>%d\n", j++, i, last_depth, last_depth - n);
#endif
  if (n ≤ 10) PUT1(system_macro_pop_0 + n);
  else if (n ≤ 255) { PUT1(system_macro_pop_small);
    PUT1(n);
  }
  else { PUT1(system_macro_pop);
    PUT2(n);
  }
  last_depth = last_depth - n;
```

This code is used in section 1783.

1783. Now we can output the timing data.

#define SHOW_WRITE_TIMING 0

⟨output timing data 1783⟩ ≡

```
  i = 0;
  { int j = 0;
    last_depth = −1;
    while (i < st_count) { int8_t c = stamp[i] ≫ 24;
      if (c ≡ system_macro_pop) { int n = stamp[i] & #FFFF;
        ⟨encode pop n 1782⟩
        i++;
      }
      else if (c ≡ system_macro_push) { int d = stamp[i + 1] ≫ 16;
        if (d ≤ last_depth) { int n = last_depth − d + 1;
          ⟨encode pop n 1782⟩
            /* here I could optimize the case n = 1 by setting the POP_BIT in system_macro_push */
        }
#if SHOW_WRITE_TIMING
        print_char('!');
        print_int(j++);
        print_char(':');
        print_int(i);
        print_char(':');
        print_int(system_macro_push);
        print_char(':');
        print_int(d);
        print_char(':');
        print_cs(stamp[i + 1] & #ffff);
        print_char(']');
        print_ln();
#endif
        PUT4(stamp[i]);
        i++;
        PUT2(stamp[i]);
        i++;
        last_depth = d;
      }
      else {
        if (c & POP_BIT) { last_depth −−;
#if SHOW_WRITE_TIMING
          printf("!%d:%d:%d>%d}\n", j++, i, last_depth + 1, last_depth);
#endif
        }
#if SHOW_WRITE_TIMING
        printf("!%d:%d:%d:%d>\n", j++, i, stamp[i] ≫ 24, last_depth);
#endif
        PUT4(stamp[i]);
        i++;
        PUT4(stamp[i]);
        i++;
      }
```

```
    }
  }
```

This code is used in section 1777.

1784. 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.